

Diplomarbeit: Abbildung von Bildschirmhalten
auf eine flächige Informationsdarstellung für Blinde
auf taktilen Geräten

Ramona Bunk

29. Dezember 2010

Zusammenfassung

Blinde Computernutzer können nur mit Hilfsmitteln auf dem Bildschirm dargestellte Informationen erfassen. Alternativ zum Bildschirm erhält diese Zielgruppe eine Beschreibung von Ausschnitten des Bildschirms per Sprachausgabe und/oder per Brailleausgabe. Die Größe eines Brailledisplays ist stark begrenzt und kann nur einen Bruchteil der visuellen Informationen wiedergeben. Eine Variante, einen möglichst großen Bildschirmausschnitt zu erfassen, ist die zeilenweise Ausgabe der Inhalte. Die Screenreader Technologie bezeichnet diese Art der Darstellung als Flächenmodus. Ein interessanter Aspekt ist die Erweiterung des herkömmlichen einzeiligen Flächenmodus für mehrzeilige Brailledisplays. Die Entwicklung dieser Displays ist derzeit Gegenstand der Forschung an verschiedenen Instituten weltweit und soll in dieser Arbeit betrachtet werden.

Einzeilige Flächendarstellungen sind unter Windows-Screenreadern bereits weit verbreitet. Für den nicht kommerziellen Bereich entwickelt sich Linux immer mehr zu einer ernst zu nehmenden Alternative. Jedoch bieten Linux-Screenreader derzeit lediglich eine strukturorientierte Ausgabe der Inhalte an. Die Umsetzung eines Flächenmodus unter Linux ist vergleichsweise problematisch, da die vorhandenen Schnittstellen nicht für eine flächendeckende Analyse der Bildschirminhalte ausgelegt sind. Der Schwerpunkt der Arbeit liegt auf der Entwicklung eines entsprechend performanten Algorithmus. Hierfür ist die Analyse der Anordnung der grafischen Elemente auf dem Bildschirm und der Accessibility Schnittstelle unter Linux notwendig. Auf Grundlage dieser Analyse wird schließlich der Algorithmus erarbeitet. Im Anschluss daran wird die Präsentation der ermittelten Inhalte auf dem Brailledisplay betrachtet.

Inhaltsverzeichnis

1	Einleitung	5
2	Stand der Technik	8
2.1	Der Tastsinn	8
2.2	Accessibility Schnittstellen	11
2.2.1	Historische Entwicklung	11
2.2.2	Apple	13
2.2.3	Windows	13
2.2.4	Linux	14
2.3	Aufbau des AT-SPI Baumes	16
2.3.1	Datenstruktur Baum: Terminologie und Eigenschaften	16
2.3.2	Arbeiten mit dem AT-SPI Baum	17
2.4	Flächenmodus bei bestehenden Screenreadern	21
2.5	Forschungsarbeiten zu Flächendisplays	22
2.5.1	NC-State-Display	24
2.5.2	Hyperbraille	24
3	Konzeption des Flächenmodus	29
3.1	Analyse der Grafischen Oberfläche	29
3.1.1	Desktop	29
3.1.2	Anwendungsfenster	30
3.1.3	Spezialfenster	33
3.1.4	Besonderheiten bei einfachen Widgets	36
3.2	Anordnung von Widgets – Probleme und Lösungen	38
3.2.1	Konflikte	38
3.2.2	Regeln	41
3.2.3	Anwenden der Regeln auf die Konflikte	42
3.3	Baumtransformation	44
3.3.1	Zerlegung in Teilbäume	46
3.3.2	Sortierung der Einheiten	53
3.3.3	Bestimmung der aktiven Einheit in einer sortierten Liste von Einheiten	55
3.3.4	Ermittlung aller Widgets in der Einheit	55

3.3.5	Aufbau eines Gewichteten Graphen	57
3.3.6	Auflösen von Konflikten	60
3.4	Darstellung auf dem Brailledisplay	63
4	Umsetzung	66
4.1	Implementierung	66
4.1.1	Integration in den Screenreader SUE	66
4.1.2	Umsetzung der Baumtransformation	68
4.2	Testen und Evaluieren des Algorithmus	69
4.2.1	Systematisches Testen	69
4.2.2	Performance-Tests	71
4.2.3	Regel 6	74
5	Zusammenfassung und Ausblick	78
A	Anordnung der Knoten im AT-SPI Baum – Heuristische Analyse	80
	Glossary	83
	Sachregister	85
	Tabellenverzeichnis	86
	Abbildungsverzeichnis	87
	Literaturverzeichnis	88

Kapitel 1

Einleitung

Seit Jahrzehnten arbeiten blinde Menschen mit Computern. Damit bieten sich ihnen viele Möglichkeiten, die den Umgang mit ihrer Behinderung in einigen Bereichen ihres Lebens einfacher gestalten. Blinden wird hiermit ein Stück mehr Unabhängigkeit in ihrem Alltag ermöglicht. So können sie Fahrpläne eigenständig in Erfahrung bringen, einkaufen und online Zeitung lesen.

Um einen Computer nutzen zu können, benötigen blinde Anwender alternative Ausgabemedien wie Sprache oder Braille. Das traditionelle Medium, der Bildschirm, ist ein rein visuelles Gerät und somit für diese Zielgruppe nicht geeignet. Sprache wird mit Hilfe von Text-to-Speech-Systemen aus einem digitalen Text erzeugt und über Lautsprecher wiedergegeben. Für die Brailleausgabe wird spezielle Hardware, das sogenannten Brailledisplay, benötigt. Durch das Setzen von piezoelektronischen Stiften werden Braillezeichen aus jeweils 8 Stiften gebildet. Je nach Setzen der Stifte entsteht ein bestimmtes Zeichen. (siehe auch [Bun-10]) Derzeit können die auf dem Markt befindlichen Braillegeräte gerade mal eine Zeile darstellen, die aus 20 bis 80 einzelnen Zeichen besteht.

Die Wahl des Ausgabemediums ist individuell sehr unterschiedlich. In Deutschland sind Braillezeilen weit verbreitet und viele Blinde arbeiten mit einer Kombination aus Braille und Sprache. Es existieren aber auch eine Reihe von Anwendern, die nur eine der beiden Techniken nutzen. Dementsprechend müssen die Programme, die die Aufbereitung des Bildschirminhaltes für die alternativen Ausgabemedien übernehmen, die Screenreader, diese unterschiedlichen Arbeitsweisen berücksichtigen. Zu den Aufgaben eines Screenreaders gehören das Auslesen der Bildschirminformationen, die Bereitstellung von bestimmten Arbeitsabläufen und die Ausgabe der Inhalte über alternative Ausgabemedien.

Die Arbeitsabläufe werden in verschiedene Modi eingestuft. Der Strukturmodus (siehe auch [Bun-10]) stellt genau *ein* grafisches Element dar. Dies kann ein Textfeld, eine Auswahlbox, ein Schalter oder ein ähnliches Element

sein und wird mit möglichst vielen Strukturinformationen präsentiert. Solche Informationen sind unter anderem Rollenbezeichnungen, Namen, Beschreibungen, Indizes oder Tastaturkürzel. Für blinde Anwender ist immer nur ein kleiner Bildschirmausschnitt sichtbar. Aus diesem Grund wurde unter Windows zusätzlich der Flächenmodus eingeführt. Anders als der Strukturmodus versucht dieser nicht, strukturelle Informationen und logische Zusammenhänge zwischen den einzelnen Objekten auf dem Bildschirm zu erfassen. Der Bildschirminhalt wird vielmehr Zeile für Zeile auf dem Brailledisplay angezeigt. Im Flächenmodus werden mehrere Elemente gleichzeitig auf der Braillezeile präsentiert. Damit erhält der blinde Nutzer die Möglichkeit, sich leichter einen Überblick über die dargestellten Fenster und deren Inhalte zu verschaffen. Aufgrund der Hardware ist es bisher nur möglich gewesen, einen Flächenmodus anzubieten, der jeweils eine Zeile anzeigen kann. Derzeit laufen Forschungsarbeiten mit dem Ziel, mehrzeilige Brailledisplays zu entwickeln. Das wird zukünftig auch Auswirkungen auf den Flächenmodus haben.

Viele blinde Anwender arbeiten vorzugsweise in einem der beiden vorgestellten Modi und schalten in den anderen um, wenn die angezeigten Informationen für das Verständnis nicht ausreichen.

Beim Arbeiten, dem Interagieren mit den einzelnen grafischen Elementen, befindet sich der Nutzer im sogenannten Navigationsmodus. Hierbei sind nur Elemente anzeigbar, die auch den Fokus erhalten können. Oftmals finden sich interessante Informationen in Objekten, die nicht den Fokus erhalten können. Um diese ebenfalls abfragen zu können, gibt es den Erkundungsmodus. Dieser erlaubt das „Erkunden“ des Bildschirms und damit die Anzeige von Titelleisten, Statusleisten und inaktiven Elementen. Der Anwender kann sich beim normalen Arbeiten zwischen dem Struktur- und dem Flächenmodus entscheiden. Vor der gleichen Wahl steht er beim Erkunden des Bildschirms.

In dieser Arbeit soll sich ausschließlich mit dem Flächenmodus auseinandergesetzt werden. Zunächst wird der aktuelle Stand der Technik betrachtet, um im Anschluss ein Konzept mit einer entsprechenden Umsetzung für einen Linux-Screenreader zu erarbeiten.

Abschnitt 2.1 untersucht die Leistungsfähigkeit des Tastsinnes. Diese Betrachtungen dienen dem Verständnis der Art von Informationen, die auf einem Brailledisplay angezeigt werden können.

Die Grundlage der Arbeit eines Screenreaders ist die Abfrage der auf dem Bildschirm dargestellten Informationen. Wie dies konkret erfolgt und welche Schnittstellen dabei unter den verschiedenen Betriebssystemen zum Einsatz kommen ist Gegenstand von Abschnitt 2.2. Ein gesonderter Abschnitt (2.3) geht im Detail auf den Aufbau der Accessibility Datenstruktur unter Linux ein.

Teure Screenreader unter Windows bieten bereits einen Flächenmodus an. Abschnitt 2.4 beschreibt diesen und untersucht die Ursachen, warum

solch ein Flächenmodus eines Windows-Screenreaders nicht einfach auf einen Linux-Screenreader übertragen werden kann.

Interessant sind Forschungsprojekte zur Entwicklung größerer und günstigerer Brailledisplays. Mit diesen Geräten können mehrere Zeilen Text, Bilder, Karten oder Schemata abgebildet werden. Dies ist Gegenstand von Abschnitt 2.5.

Anschließend stellt Kapitel 3 die Konzeption eines Flächenmodus unter Linux vor. Hilfstechnologien unter Linux verwenden eine Schnittstelle, die die Bildschirminhalte in einem Baum nachbildet. Durch die Art der Kommunikation zwischen der Anwendung, der Accessibility Schnittstelle und dem Screenreader enthält der Screenreader beim Auftreten eines Ereignisses immer genau einen Knoten des Baumes. Dieser repräsentiert ein grafisches Element, wie etwa einen Druckknopf. Der Screenreader hat zunächst keine Kenntnis darüber, an welcher Stelle des Baumes sich der Knoten befindet. Im Strukturmodus ist dies nicht weiter problematisch, da nur ein einzelnes Element dargestellt wird. Im schlechtesten Fall müssen noch ein paar Nachbarknoten im Baum erforscht werden, um strukturelle Zusammenhänge zu klären. Beim Flächenmodus sieht der Fall etwas anders aus. Hier ergeben sich gleich mehrere Probleme. Abschnitt 3.2 zeigt, dass die Einteilung in Bildschirmzeilen nicht immer trivial ist. So gibt es Fälle, bei denen grafische Elemente mehreren imaginären Zeilen zugeordnet werden können. Ziel ist es, ein Verfahren zu entwickeln, welches die eindeutige Einteilung in Zeilen zum Ergebnis hat.

Nachdem die Zuordnung der grafischen Elemente zu Zeilen erfolgt ist, müssen die einzelnen Zeilen aus dem Baum extrahiert werden. Da der Accessibility Baum die Elemente aller geöffneten Fenster enthält, kann er unter Umständen recht groß werden. Die Abfrage aller Elemente im Baum und deren anschließende Auswertung würde den Computer für mehrere Sekunden blockieren und damit ein Arbeiten in dieser Zeit nicht gestatten. Kein blinder Anwender würde einen Screenreader benutzen, der nach jeder Aktion ein paar Sekunden das Arbeiten unmöglich macht. Es muss also eine Lösung gefunden werden, die eine gewisse Performance bietet. Abschnitt 3.3 zeigt ein Konzept, bei dem möglichst wenige Knoten des Baumes analysiert werden, um das gewünschte Ergebnis zu liefern.

Zum Abschluss des Kapitels „Konzeption“ wird betrachtet, wie die ermittelten Inhalte auf dem Brailledisplay dargestellt werden sollen. Neu bei dieser Betrachtung ist die Berücksichtigung von mehreren Zeilen Text auf einem Braillegerät.

Nachdem das Konzept erarbeitet ist, soll es für den Linux-Screenreader SUE umgesetzt werden. Kapitel 4 beschreibt die Details dieser Verwirklichung.

Der abschließende Ausblick geht auf offene Probleme und Fragestellungen ein.

Kapitel 2

Stand der Technik

2.1 Der Tastsinn

Diese Arbeit beschäftigt sich mit Darstellungsformen auf dem Brailledisplay, die über die Präsentation einzelner Bildschirm-Elemente hinausgehen. Die Sinneswahrnehmung, die bei diesem Ausgabemedium gefordert ist, ist der Tastsinn. Die grundlegende Fragestellung in diesem Zusammenhang ist zunächst, wie leistungsfähig dieser Sinn ist. Wie komplex dürfen die dargestellten Inhalte sein? Können Grafiken haptisch erfasst werden? Ist es sinnvoll, die Bildschirminhalte auch grafisch auf einem Brailledisplay wiederzugeben? Solche und ähnliche Fragestellungen sind die Motivation zu den nachfolgenden Betrachtungen.

Der Tastsinn ist ein aktiver Sinn. Gegenstände, Formen und Oberflächen werden mit den Händen erkundet. Das ist ein bewusster, gewollter Prozess. Im Gegensatz dazu wird selten die Entscheidung getroffen, bewusst etwas sehen oder hören zu wollen. Der taktile Sinn leistet gute Dienste beim Lesen von Brailleschrift und taktilen Bildern, aber das volle Potential entfaltet sich erst beim Erkunden von räumlichen Gegenständen und Texturen.

In den Forschungsfeldern der Psychologie und Neurowissenschaften werden derzeit verschiedene Ansätze zum Thema „Taktile Wahrnehmung“ verfolgt. Die zentrale Frage ist bei diesen Forschungen: Können wir äquivalente Informationen über den Tast- und Sehsinn aufnehmen?

Die erste wichtige Erkenntnis ist, dass unsere Wahrnehmung multimodal ist. Wir nehmen unsere Umwelt über mehrere Sinne gleichzeitig wahr. Wir sehen die Welt, fühlen, hören und riechen sie. Wir sind selten auf einen Sinn beschränkt, wenn wir Informationen aus unserer Umgebung aufnehmen. Für einen Menschen, der alle seine Sinne nutzen kann, sind Sehen und Tasten zwei sich ergänzende Eindrücke. Meistens liefern unterschiedliche Sinne redundante Informationen, die sich gegenseitig bestätigen. Wenn sie konträr sind, lernen wir, die Wahrheit zu erkennen. Heller [Hel-06a] zeigt in seinen Forschungen, dass Fehlinterpretationen eines Sinnes durch den Einsatz eines

weiteren erkannt werden können.

Millar [Mil-06] belegt in ihrer Arbeit, dass für die Interpretation der aufgenommenen Informationen ein Bezugsrahmen von entscheidender Bedeutung ist. Das Auge versucht stets das gesamte Bild zu erfassen und hat damit die Umwelt als Bezugsrahmen. Fehlt dieser, kommt es zu Illusionen und wir sind unter Umständen nicht mehr in der Lage, den Bildausschnitt richtig zu interpretieren. Bei der Übertragung dieser Erkenntnis auf den Tastsinn ist es offensichtlich, dass es nicht ausreicht, Formen und Bilder mit nur einem Finger erfühlen zu wollen. Das Gesamtbild, der Bezugsrahmen, fehlt. Es kommt häufiger zu Fehlinterpretationen. Millar identifiziert drei mögliche Bezugsrahmen: *body-centered*, *external-environmental* und *object-based*. *Body-centered* oder auch egozentrisch beschreibt körpereigene Anhaltspunkte wie die Körperhaltung. Unter *external-environmental* werden vom ertasteten Gegenstand unabhängige, externe, räumliche Informationen verstanden und *object-based* bezieht sich auf den Gegenstand selbst.

Millar führte eine Versuchsreihe durch, bei der der Einfluss eines externen im Gegensatz zu einem egozentrischen Bezugsrahmen untersucht wurde. Die Probanden mussten unter verschiedenen Bedingungen Landmassen auf einer taktilen Karte ertasten und finden. Das Ergebnis war überraschend. Externe Bezugspunkte können genauso effektiv sein wie egozentrische. Nimmt man beide Bezugsrahmen zugleich zur Hilfe, waren die Ergebnisse doppelt so akkurat, wie einzeln genommen. Dabei kann ein externer Bezugspunkt einfach ein Rahmen um die Karte sein. Die Ausnutzung von vorhandenen externen Bezugspunkten bei haptischen Informationen geschieht nicht automatisch, sondern muss erlernt werden. Während bei visuellen Daten fast immer ein Hintergrund vorhanden ist, der entsprechend ausgenutzt wird, sind solche Einzelheiten dem Tastsinn meist nicht zugänglich. Dementsprechend sind wir es nicht gewohnt, solche haptischen Hintergrundinformationen mit einzubeziehen. Aufbauend auf diesen Ergebnissen schlägt Millar vor, beim ertasten von haptischen Formen zwei Hände einzusetzen. Eine um den äußeren Rahmen abzustecken und eine, um die Form an sich zu erfassen. Versuche zeigen, dass diese Taktik durchaus zu besseren Ergebnissen führt.

Bei einem Versuch, einfache 2-D Formen zu ertasten, stellt Heller die These auf, dass Probleme bei der Schwierigkeit taktile Bilder zu benennen mit dem Zugriff auf das semantische Gedächtnis zusammenhängen und nicht auf eine Begrenzung der haptischen Fähigkeiten zurückzuführen sind. Wenn Probanden eine grobe Kategorisierung zu einer zu findenden Form erhalten, fiel es ihnen sehr viel leichter, diese aus einer Reihe von möglichen Formen zu erkennen.

Blinde und Pädagogen, die diese Zielgruppe unterrichten, unterliegen auch heute noch weit verbreiteten Vorurteilen über die Fähigkeiten taktile Bilder zu erfassen. Heller beschreibt diese Ansichten folgendermaßen:

„[...] it was stupid and idiotic to expect blind people to make

sense of tangible pictures [...]“

„[...] it was perfectly reasonable to teach blind people about maps and graphs. However, it was not sensible to ask blind people to „think sighted,“[...]“ [Hel-06a]

Heller ist der Meinung, dass der Sehsinn ein nützlicher und überlegener Sinn für das Erfassen von räumlichen Informationen ist. Jedoch sind die Konsequenzen eines fehlenden Sehsinnes nicht so eindeutig. Der Tastsinn kann alternative Informationen liefern, wenn der Sehsinn verloren wird oder von Geburt an fehlt. Der haptische Sinn ist allerdings langsamer beim Erkennen von Formen, als der Sehsinn. Durch Geschick und Übung kann diese reduzierte Effizienz etwas kompensiert werden. Kennedy (1993) geht sogar noch einen Schritt weiter und sagt, dass der Tastsinn äquivalente Informationen zum Sehsinn liefert.

Die oben beschriebenen Vorurteile entstammen der Tatsache, dass Geburtsblinde sich Objekte oftmals anders vorstellen als Sehende. Sehende betrachten einen Gegenstand aus einem bestimmten Blickwinkel. Sie erfassen meist nur 1 bis 2 Seiten oder Teile des Objektes zu einem bestimmten Zeitpunkt. Geburtsblinde hingegen betrachten den gesamten Gegenstand. Sie können diesen auch aus einem Blickwinkel betrachten, aber anscheinend ist das nicht die Normalität für Menschen, die keinerlei visuelle Erfahrungen haben. Heller schreibt hierzu:

„[...] It also may be a relatively unfamiliar skill, and this could influence the reactions of blind people to pictures. Moreover, it is not known if there are preferred viewpoints in haptic pictures of many familiar objects, nor is it known if this would change with practise and skill.“[Hel-06a]

Er kommt zu dem Schluss, dass visuelle Erfahrungen nicht notwendig sind, um Perspektiven zu erkennen. Es scheint vielmehr ein Lernprozess zu sein.

Kennedy und Juricevic [Ken-06] gehen hierauf näher ein. Schon im Säuglingsalter entwickelt sich die Tiefenwahrnehmung und die Bewegungserfassung in Bezug zur Umwelt. Die visuelle Entwicklung erfolgt graduell mit der Verbesserung der Seheindrücke in den ersten Jahren des Lebens. Analog dazu bildet sich die haptische Räumlichkeitswahrnehmung heraus. Auch diese entwickelt sich graduell. Sowohl das visuelle, als auch das haptische Wahrnehmungssystem erfassen den drei-dimensionalen Raum. Die Ergebnisse sollten miteinander in Beziehung stehen, so dass zum Beispiel Linien in Zeichnungen ähnliche Vorstellungen in Vision und Tastempfinden erzeugen. Unser Gehirn nutzt beim Verarbeiten von Seh- und Tastinformationen die selben Hirn-Ariale zum Erkennen und Interpretieren von räumlichen For-

men und Grenzen. Dies hat Auswirkungen auf die Zeichenfähigkeit. Kennedy und Juricevic beschreiben dies wie folgt:

„If the self-organizing aspects of perspective in vision and touch are alike, they could support a drawing development trajectory that makes specific spatial skills [...] and projection systems [...] available in the same order to the blind and sighted. The evidence in our figures here lends itself to that speculation.“ [Ken-06]

Dies bestätigte die Analyse von taktilen Bildern, die von Geburtsblinden angefertigt wurden. Je mehr Erfahrung im Zeichnen die blinden Probanden hatten, desto mehr spielten auch Perspektiven, wie wir sie in Zeichnungen von Sehenden kennen, eine Rolle. Auch Sehende lernen erst in der Schule, richtige perspektivische Formen im zwei-dimensionalen Raum zu erstellen.

Die Forschungsergebnisse zeigen, dass es gar nicht so abwegig ist, blinden Menschen Grafiken zu präsentieren. Es muss jedoch berücksichtigt werden, dass das Erfassen und Erstellen solcher Grafiken erlernt werden muss, bevor effektiv damit gearbeitet werden kann. Weitere Untersuchungen sind notwendig, um die Grenzen der Komplexität solcher Grafiken zu finden. Eines haben Heller und seine Kollegen bereits gezeigt: Die Bilder können weit komplexer sein, als bisher gemeinhin angenommen.

Es macht also Sinn, blinden Computernutzern auch grafische Informationen auf einem Brailledisplay zu präsentieren. Dies führt zu neuen Herausforderungen bei der Abbildung von Bildschirminhalten in taktiler Form. Damit sind taktile Darstellungen und Arbeitsweisen denkbar, die den visuellen stark ähneln. Wie solche aussehen können, wurde in einem Teilprojekt des Hyperbraille Projektes untersucht. Hierauf wird in Abschnitt 2.5.2 dieser Arbeit näher eingegangen.

2.2 Accessibility Schnittstellen

Im folgenden Abschnitt sollen die notwendigen Schnittstellen betrachtet werden, mit denen ein Screenreader die Bildschirminformationen abrufen kann. In diesem Zusammenhang wird ein kurzer Rückblick auf die Entwicklung der Accessibility Schnittstellen gegeben. Anschließend erfolgt eine Beschreibung konkreter Schnittstellen für die Betriebssysteme Mac OS X, Windows und Linux.

2.2.1 Historische Entwicklung

Peter Korn, der derzeitige Accessibility Principal bei Oracle, teilt die Entwicklung der Accessibility im Bereich der Informationstechnik in drei Phasen ein. [Kor-1]

Die erste Phase ist der „1st Generation Access“. In den späten 1960er Jahren wurden erste Techniken entwickelt, mit denen auch Menschen mit Behinderungen Computer nutzen konnten. Zu der Zeit waren die Bildschirmausgaben noch zeichenbasiert und die Hilfsmittel-Software erhielt ihre Inhalte aus dem Textbuffer. Bis in die 1980er Jahre hinein wurden erste Screenreader erschaffen, die auf diese Art die notwendigen Informationen vom System bekamen.

Um in den ersten Jahrzehnten das Arbeiten mit einem Computer für sehgeschädigte Menschen überhaupt zu ermöglichen, war der Einsatz von spezieller Hardware notwendig. So wurden Eingaben über Spezialtastaturen getätigt und war man auf eine vergrößerte Darstellung der Bildschirminhalte angewiesen, so wurden eigens für diesen Zweck entwickelte Grafikkarten benötigt. Blinde Anwender nutzten unter anderem den Optacon (OPTical to TActile CONverter) [Tel-03] [Tel-05]. Dies war ein elektromechanisches Gerät, mit dessen Hilfe Blinde gedruckte Textseiten lesen konnten. Über eine kleine Kamera wurde der Text eingelesen und dann entsprechend taktil für die Anwender dargestellt. Dieses Gerät erzeugte allerdings kein Braille, sondern erschuf ein taktilles Abbild der Schwarzschriftbuchstaben mit denen Sehende arbeiten. Die blinden Anwender waren es nicht gewohnt, mit Schwarzschriftzeichen zu arbeiten und einige hatten nicht einmal eine Vorstellung davon, wie diese aussahen. Sie mussten das Erkennen dieser Zeichen neu erlernen und es bedurfte einiger Übung, um eine gewisse Lesegeschwindigkeit zu erreichen.

In den späten 1980er Jahren veränderte sich die Arbeitsweise am PC von zeichenbasierten Bildschirmausgaben hin zur grafischen Benutzeroberfläche (GUI – Graphical User Interface). Damit begann der „2nd Generation Access“. Aufgrund dieses Wandels konnten und können sehgeschädigte Nutzer nun zum überwiegenden Teil die gleiche Hardware verwenden, die auch Sehende einsetzen. Die Erfüllung von speziellen Bedürfnissen der Sehgeschädigten bei der Computernutzung wurde nun auf die Software verlagert. Allerdings wurde herkömmliche Software für Sehende erstellt, ohne die Bedürfnisse von Randgruppen zu berücksichtigen. Deshalb musste die Accessibility Software meist in die Schnittstellen des Systems hacken, um an die dargestellten Inhalte zu gelangen und diese entsprechend für die sehgeschädigten Nutzer aufzubereiten.

Seit 1997 kristallisiert sich schrittweise der „3rd Generation Access“ heraus. Hierbei liegt der Schwerpunkt vor allem auf der Integration der Softwarelösung in das System, um damit Systemhacks, wie sie bisher nötig waren, zu vermeiden. Alle Informationen, die die Assistive Technology (AT) benötigt, werden beim 3rd Generation Access über eine API (Application Programming Interface) zur Verfügung gestellt. Eine der Ersten dieser Art war die Java Accessibility API. Accessibility APIs arbeiten nach einem Client-Server basiertem System. Die AT-Software ist der Client und die Anwendungen, welche die Schnittstellen bedienen treten als Server auf. Der

jeweilige Server muss die Anfragen des Clients beantworten. Die Kommunikation zwischen den beiden Parteien wird über einen Teil der API spezifiziert. Parallel zu diesen Schnittstellen wurden eine Reihe von Accessibility Standards entwickelt:

- Die Web Content Accessibility Guidelines (WCAG) wurden vom World Wide Web Consortium (W3C) erarbeitet und beinhalten Richtlinien für die Erstellung von barrierefreien Webseiten.
- Die User Agent Accessibility Guidelines (UAAG), ebenfalls aus der Hand des W3C, definieren, wie Browser, Media Player und andere sogenannte „User Agents“ die barrierefreie Nutzung von Webinhalten durch Menschen mit Behinderungen ermöglichen sollen.
- Die Authoring Tools Accessibility Guidelines beschreiben die Richtlinien für barrierefreie Autorensysteme, sprich Content Management Systeme. Auch diese entstanden unter der Obhut des W3C.
- Derzeit wird ISO 13066 erarbeitet. Dieser Standard hat die Kompatibilität von Informationstechnologien mit Assistive Technology zum Inhalt.

2.2.2 Apple

Bereits seit 20 Jahren bietet Apple innovative Lösungen, die es Menschen mit Behinderungen erlauben, ihre Produkte zu nutzen. iPhone, iPad, iPod und Mac OS X enthalten standardmäßig Vergrößerungssoftware und den haus-eigenen Screenreader VoiceOver. Apple versucht auf die Bedürfnisse seiner Kunden einzugehen und bietet neue Lösungen, um Probleme anzugehen. So gibt es zum Beispiel einen Braille-Spiegel, der es blinden und gehörlosen Kindern erlaubt, gemeinsam an einem Rechner zu arbeiten. Zusätzlich bietet Apple den weltweit ersten Screenreader, der über Gesten gesteuert werden kann.

Seit dem Mac OS X Version 10.2 (Erschienen 2002) gibt es bei Apple das Accessibility Framework, eine „3rd Generation Access“ Schnittstelle. Dieses Framework besteht zum einen aus einem Accessibility Protokoll, welches von den objektorientierten API's Carbon und Cocoa implementiert wird. Damit können Anwendungen ihre Informationen der AT-Software offenbaren. Zum anderen enthält das Framework eine API, mit deren Hilfe AT-Software in der Lage ist, in den Ablauf von Anwendungen einzugreifen.

2.2.3 Windows

Unter Windows ist der „2nd Generation Access“ noch weit verbreitet. Die Screenreader für die Microsoft Betriebssysteme arbeiten mit einem Off-

Screen-Modell, um die Informationen der grafischen Oberfläche für ihre Zwecke nachzubilden.

Für GUI basierte Anwendungen werden Pixel und nicht mehr, wie noch im reinen Textmodus, Character/Attribut Paare im Grafikspeicher abgelegt. Dies führt zur sogenannten „Pixel Barriere“. Zur Überwindung dieser Pixel Barriere wird eine virtuelle Bildschirm Kopie (VISC, virtual screen copy) verwendet. In dieser VISC werden die Pixel den dargestellten Character-Zeichen zugeordnet. Diese Zuordnung kann auf zwei Wegen erlangt werden. Zum einen über die Modifizierung des Grafiktreibers, um die Information zu erhalten, sobald das Zeichen in Pixel umgewandelt wird. Zum anderen über die Umkehrung des Pixel Prozesses durch das Anwenden von OCR (Optical Character Recognition, Texterkennung) Techniken. Diese Techniken können entweder direkt auf des Video Signal oder auf den Grafikspeicher angewendet werden. Die OCR ist unabhängig vom Betriebssystemen und der verwendeten GUI. Allerdings ist die Erkennung nicht immer akkurat und leidet oftmals unter einer relativ schlechten Performance.

Die VISC enthält nur textliche Informationen, aber keine über den Aufbau des Fensters und seiner Widgets. Damit kann nicht einmal zugeordnet werden, zu welcher Anwendung der Text gehört, da Fensterdimensionen im VISC nicht bekannt sind. Der Screenreader benötigt ein Modell, welches die Fenster- und Widget-Strukturen abbildet. Dies macht das Off-Screen-Modell (OSM).

1998 wurde mit MSAA (Microsoft Active Accessibility) ein erster Versuch in Richtung „3rd Generation Access“ veröffentlicht. Allerdings bot MSAA nicht genügend Funktionalität, um alle Bedürfnisse von AT-Software abzudecken. Diese waren zum überwiegenden Teil weiterhin auf die bisher verwendeten Hacks angewiesen. 2006 begannen dann die Arbeiten zu IAccessible2. IAccessible2 erweitert das Interface von MSAA, welches den Namen IAccessible trägt, um Funktionen, die bisher fehlten. Als Vorbild nahmen sich die Entwickler die Java Accessibility API. Peter Korn schreibt hierzu in seinem Weblog vom 14.12.2006 [Kor-2]:

„[...] in fact, if you look at the IAccessibleRelation header file or the other IAccessible2 header files, you'll see they bear a Sun Copyright from 2000 and 2006 because IAccessible2 was derived directly from the OpenOffice.org UNO Accessibility implementation for use in both the Java platform and UNIX environments.“

2.2.4 Linux

Linux bietet vielseitige Lösungen im Bereich Accessibility. Diese reichen von rein zeichenbasierten Distributionen wie Ariadne bis zu komplexen grafischen Oberflächen wie bei Blinux oder Ubuntu. Die modernste Linux Accessibility Architektur bietet die Desktop-Umgebung GNOME. Sie baut auf

der Arbeitsweise des „3rd Generation Access“ auf.

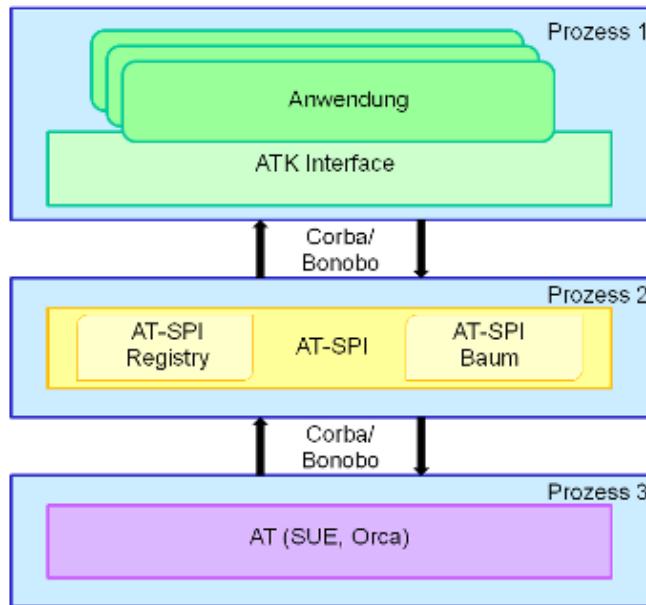


Abbildung 2.1: Linux: GNOME Accessibility Architektur

Diese Architektur ging aus dem GNOME Accessibility Projekt (GAP) hervor und wird ständig weiterentwickelt. Abbildung 2.1 zeigt die Zusammenhänge der einzelnen Schichten. Jede Schicht arbeitet unabhängig von den anderen in einem eigenständigen Prozess. Für die Interprozesskommunikation kommen unter GNOME die Programme Corba und Bonobo zum Einsatz. Diese übernehmen den Datenaustausch zwischen den einzelnen Bibliotheken und Programmen der Accessibility Technologie.

Assistive Technologie (AT), wie zum Beispiel Screenreader, erhalten Informationen von den Applikationen über die Accessibility Toolkit/Assistive Technology Service Provider Interface (ATK/AT-SPI).

Die Unterstützung für ATK ist in den GNOME Widgets integriert. Die AT-Software enthält ihre Informationen also direkt von der Anwendung. Der Vorteil dabei ist, dass die Anwendung den Aufbau und die Zusammenhänge ihrer Widgets kennt und diese direkt über ATK und AT-SPI der AT-Software bekannt geben kann.

Das Accessibility Toolkit (ATK) beschreibt eine Menge von Schnittstellen, die GUI abhängig implementiert werden müssen. Für das GNOME eigene Toolkit GTK wird ATK in GAIL (GNOME Accessibility Implementation Library) implementiert. GAIL wird dynamisch zur Laufzeit einer GTK Anwendung geladen und bei der Verwendung von Standard GTK Objekten stehen bereits Basisinformationen der Widgets in der AT zur Verfügung. Wenn GAIL nicht geladen wird, haben die GTK Widgets eine Default Accessibili-

ty Implementierung, die keine Informationen liefert, aber konform zur ATK API ist. Weitere ATK Implementationen gibt es für Java-Anwendungen, das OpenOffice.org Projekt und Mozilla.

Der Austausch der Informationen zwischen den Anwendungen und der AT-Software wird in Form von Corba-Objekten vorgenommen. Dieser Prozess wird von der AT-SPI Registry gesteuert.

2.3 Aufbau des AT-SPI Baumes

Im folgenden Abschnitt wird die Datenstruktur der AT-SPI Schnittstelle näher betrachtet. Beim Abfragen von Inhalten einer Anwendung über die Accessibility Architektur wird zunächst einmal ein ATK Objekt von dem entsprechenden Inhalt erstellt. Die so entstehenden Objekte werden mit Hilfe eines Baumes zueinander in Beziehung gesetzt. Dieser Baum bildet die komplette grafische Oberfläche nach und wird auch als AT-SPI Baum bezeichnet. Bevor es zur näheren Betrachtung des AT-SPI Baumes kommt, sollen zunächst Begriffsdefinitionen zu der Datenstruktur Baum gegeben werden.

2.3.1 Datenstruktur Baum: Terminologie und Eigenschaften

Bäume sind verallgemeinerte Listenstrukturen. Die Elemente eines Baumes werden als Knoten bezeichnet und können, im Gegensatz zu Listen, mehr als nur einen Nachfolger haben. Die Anzahl der Nachfolger eines Knotens ist endlich. Nachfolger werden üblicherweise als Söhne oder Children bezeichnet. Analog dazu werden unmittelbare Vorgänger mit Vater oder Parent benannt. Alle Vorgänger eines Knotens sind die Vorfahren oder auch Ancestors und alle Nachfolger die Nachfahren bzw. Descendants.

In einem Baum gibt es genau einen Knoten, der keinen Vater-Knoten hat. Diesen Knoten nennt man Wurzel des Baumes. Alle Knoten, die keine Children haben heißen Blätter des Baumes. Die restlichen Elemente, die sowohl einen Vater-Knoten, als auch Söhne haben, sind innere Knoten.

Ein Pfad wird bei Ottmann [Ott-02] folgendermaßen definiert:

„Eine Folge p_0, \dots, p_k von Knoten eines Baumes, die die Bedingung erfüllt, dass p_{i+1} Sohn von p_i ist für $0 \leq i < k$, heißt Pfad mit Länge k , der p_0 mit p_k verbindet. Jeder von der Wurzel verschiedene Knoten eines Baumes ist durch genau einen Pfad mit der Wurzel verbunden.“

Ist unter den Söhnen eines Knotens eine Reihenfolge definiert, so dass man vom ersten, zweiten, dritten, usw. Sohn sprechen kann, so nennt man den Baum geordnet. Dies ist nicht zu verwechseln mit der Ordnung eines Baumes, die die maximale Anzahl von Söhnen zu einem Knoten angibt. Die

Anzahl von Child-Knoten zu einem Knoten p nennt man auch den Rang von p . Mit der Höhe eines Baumes wird der maximal Abstand eines Blattes von der Wurzel angegeben und die Tiefe eines Knotens gibt den Abstand dieses Knotens zur Wurzel an. Alle Knoten eines Baumes, die die gleiche Tiefe haben, werden zu einer Ebene zusammengefasst.

Ottmann definiert einen Baum als vollständig, „wenn er auf jeder Ebene die maximal mögliche Knotenzahl hat und sämtliche Blätter dieselbe Tiefe haben.“[Ott-02]

Um Suchanfragen auf Bäume effizienter zu gestalten, ist es sinnvoll, die Höhendifferenz zwischen Teilbäumen eines Knotens zu begrenzen. Gibt es eine solche Forderung, so spricht man von höhenbalancierten Bäumen.

Die eben definierten Begrifflichkeiten werden im Folgenden bei der Beschreibung des AT-SPI Baumes angewendet.

2.3.2 Arbeiten mit dem AT-SPI Baum

Die Wurzel des Baumes ist der abstrakte Desktop der GNOME GUI. Die Söhne der Wurzel repräsentieren jeweils eine der Anwendungen, die gerade aktiv sind. Standardmäßig sind dies zum Beispiel folgende Knoten:

- `gnome-panel` – enthält die Child-Knoten für das obere und das untere Kantenpanel
- `nautilus` – enthält den Child-Knoten `x-nautilus-desktop`, die Repräsentation des Nutzerdesktops
- alle Anwendungen, die durch ein Fenster repräsentiert werden, unter anderem der Texteditor GEdit oder die Office Suite OpenOffice.org

Auf der 3. Ebene des Baumes – unterhalb der Anwendungsknoten – befinden sich unter anderem die Knoten, die die geöffneten Fenster der jeweiligen Anwendung repräsentieren. Geht man tiefer in den Baum hinein findet man nun eine Repräsentation für jedes Widget und jeden Rahmen um eine Gruppe von Widgets in Form eines Knotens. Das komplette Fenster wird auf diese Weise im Baum abgebildet. Häufig werden verschiedene zusammenhängende Bereiche durch Rahmen-Elemente wie Panel und Filler von anderen Bereichen abgekapselt.

Abbildung 2.2 zeigt die Verteilung der Knoten im AT-SPI Baum. Der Zugriff auf die Inhalte des Baumes kann auf zwei verschiedenen Wegen erfolgen. Zum einen besteht die Möglichkeit direkt eine Anfrage nach der Wurzel des Baumes zu tätigen. Ein Teil der AT-SPI Schnittstelle, die AT-SPI Registry, bearbeitet derartige Anfragen und liefert dann ein Corba-Objekt an die anfragende AT-Software zurück. Dieses Objekt repräsentiert die Wurzel des Baumes, den abstrakten Desktop. Die AT-SPI Schnittstelle stellt

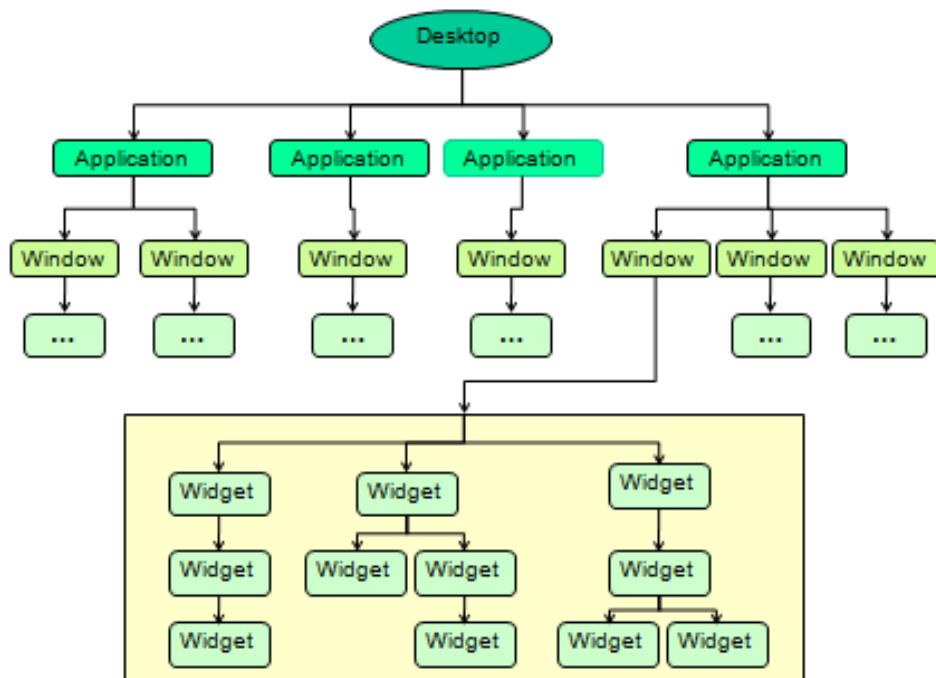


Abbildung 2.2: Schematische Darstellung des AT-SPI Baumes

nun verschiedene Methoden bereit, mit denen die Eigenschaften des Corba-Objektes erkundet werden können. Außerdem kann mit Hilfe dieser Methoden der Baum erforscht werden, indem über Child- und Parent-Beziehungen weitere Knoten des Baumes erfragt werden. Jeder abgefragte Knoten wird in Form eines Corba-Objektes zurückgeliefert. Dieses Objekt wird innerhalb des Accessibility Frameworks als Accessible bezeichnet.

Die zweite Möglichkeit, den Baum zu erkunden, besteht über das Beobachten von auftretenden Ereignissen in der Anwendung. Hierzu meldet die AT-Software Interesse an bestimmten Ereignissen bei der AT-SPI Registry an. Sobald in einer Anwendung ein Ereignis ausgelöst wird, erfolgt eine Meldung an die AT-SPI Registry. Diese schaut nun, ob sich eine AT-Software für dieses Ereignis von dieser Anwendung interessiert. Ist das der Fall, so übermittelt die AT-SPI Registry das Ereignis, zusammen mit dem Widget, welches von dem Ereignis manipuliert wurde, in Form eines Corba-Objektes. Dieses Corba-Objekt repräsentiert gleichzeitig einen Knoten im AT-SPI Baum und damit einen Ausgangspunkt, um den Baum zu erkunden. Dies ist die Standardvariante für eine AT-Software, um Objekte aus dem AT-SPI Baum zu verarbeiten. Der Nachteil besteht darin, dass die AT-Software zunächst nicht weiß, wo genau sich dieser Knoten im Baum befindet. Wird nur ein Widget, also nur ein Knoten im Baum betrachtet, ist dies nicht weiter hinderlich. Das ist zum Beispiel im Strukturmodus eines Screenreaders der

Fall. Bei einer flächendeckenden Darstellung wie sie im Flächenmodus eines Screenreaders zu erwarten ist, kann dies allerdings problematisch werden. Denn zunächst muss festgestellt werden, wo der Knoten im Baum lokalisiert ist und welche Knoten den benachbarten Widgets in der Bildschirmdarstellung entsprechen. Für solche Arten von Anfragen ist der Baum allerdings nicht konzipiert. Je mehr Knoten im Baum besucht werden müssen, um die notwendigen Informationen zusammen zu stellen, desto schwerfälliger wird das Arbeiten auf dem Baum. Jede Anfrage von der AT-Software zum AT-SPI Baum geht über die Interprozess-Kommunikations-Schnittstelle und das kostet Zeit.

Bei dem AT-SPI Baum handelt es sich nicht um einen geordneten Baum in dem Sinne, dass der erste Child-Knoten auch wirklich links vom zweiten Child-Knoten angeordnet ist. Die Reihenfolge der Anordnung von Geschwisterknoten im Baum ist vielmehr von der jeweiligen Programmierung der Anwendung abhängig. Ging der Entwickler systematisch beim Erstellen der GUI vor, so ist die Chance recht hoch, dass auch die Knoten im AT-SPI Baum einer gewissen Ordnung unterliegen. Umgekehrt kann die Anordnung von Geschwisterknoten im Baum auch recht willkürlich sein, wenn die GUI der Anwendung nicht so sorgfältig erschaffen wurde.

Für die Navigation im Baum ist es jedoch notwendig, jeden Child-Knoten eindeutig ansprechen zu können. Aus diesem Grund gibt es einen Index über die Child-Knoten eines Accessibles und man kann von einem geordneten Baum sprechen.

Für den AT-SPI Baum kann keine Ordnung festgelegt werden, da nicht vorhersagbar ist, wie viele Children ein Knoten hat. So hat zum Beispiel ein Spread Sheet in der OpenOffice.org Tabellenkalkulation 1.073.741.824 Child-Knoten. Da der AT-SPI Baum die grafische Oberfläche widerspiegelt, ist er auch nicht balanciert oder vollständig.

Eigenschaften eines Knotens

Im Folgenden sollen die für diese Arbeit wichtigen Eigenschaften eines Knotens im AT-SPI Baum betrachtet werden. Dafür ist es notwendig die Schnittstelle genauer zu untersuchen. AT-SPI stellt für das Accessible Objekt verschiedene Attribute und Methoden bereit.

Zunächst einmal hat das Accessible eine Rolle und einen Namen. Die Rolle gibt die Art des Widgets an, zum Beispiel `push button`, `menu bar` oder `text`. Sie kann über eine der Methoden `getRole()`, `getRoleName()` oder `getLocalizedRoleName()` abgefragt werden. Die Methode `getRole()` liefert ein Objekt vom Typ `Role` zurück. `getRoleName()` hingegen holt die englische Rollenbezeichnung und `getLocalizedRoleName()` bietet die übersetzte Form der Rolle an. Der Name des Accessibles sollte kurz die Funktionalität des grafischen Elementes beschreiben. Für eine längere, konkretere Beschreibung gibt es die Eigenschaft `description`, die oftmals den Tooltip

zu einem Widget widerspiegelt. Diese Eigenschaften erhält man jeweils über die Attribute `name` und `description`. Sie werden immer in ihrer lokalisierten Form zurück gegeben.

Weiterhin besitzt ein `Accessible` eine Reihe von Zuständen, wie `visible`, `enabled` und `focused`. Die Methode `getState()` liefert ein Objekt `StateSet` zurück, welches die aktuellen Zustände des `Accessibles` enthält. Über Beziehungen zwischen `Accessibles` können zusätzliche Verknüpfungen zwischen den Knoten des Baumes aufgebaut werden, die über normale Baumstrukturen hinaus gehen. Dies wird vorwiegend dafür verwendet, um inhaltlich zusammengehörige Widgets besser kennzeichnen zu können. So zum Beispiel um ein Label einem Eingabefeld eindeutig zuzuordnen. Beide Elemente sind normalerweise Geschwisterelemente. Wenn es auf der Ebene weitere Eingabefelder und Label gibt, ist ohne die Beziehung nicht eindeutig, welches Label zu welchem Eingabefeld gehört. Solche Beziehungen werden in einem Objekt `RelationSet` gespeichert, welches über die Methode `getRelationSet()` für das jeweilig `Accessible` zugänglich ist.

Zu den wichtigsten Eigenschaften eines Baumes zählen die Parent-Child-Beziehungen zwischen den einzelnen Knoten. Über das Attribut `parent` kann jedes `Accessible` seinen Vater-Knoten erfragen. Das Attribut `childCount` gibt Auskunft über die Anzahl der eigenen Child-Knoten. Zum direkten und eindeutigen Ansprechen jedes Child-Knotens existiert ein Index über diese und die Methode `getChildAtIndex` gibt das entsprechende Child-`Accessible` zurück. Mit `getIndexInParent()` kann jedes `Accessible` seinen eigenen Index in Bezug zu seinem Parent erfragen.

Abhängig von dem Widget, welches das `Accessible` repräsentiert, können zusätzlich verschiedene Interfaces für das `Accessible` implementiert sein. Konkret sind dies `Text`, `Table`, `Image`, `Value`, `EditableText`, `Hypertext`, `Desktop`, `Application`, `Selection` und `StreamableContent`. Über jedes dieser Interfaces werden weitere Eigenschaften und Methoden für das `Accessible` zur Verfügung gestellt.

Für diese Arbeit spielt das Interface `Component` eine wichtige Rolle. Es ist für fast alle `Accessibles` implementiert und bietet Informationen zur räumlichen Anordnung auf dem Bildschirm und der Größe des Widgets. Über die Methode `getExtents()` kann ein Objekt vom Typ `BoundingBox` erfragt werden, welches die Koordinaten der linken, oberen Ecke des Widget enthält und seine Abmaße. Die gleichen Informationen können separat über die Methoden `getPosition()` und `getSize()` erlangt werden.

Es gibt eine Reihe weiterer Attribute und Methoden zu einem `Accessible` (siehe [SPI-1] und [SPI-2]). Die hier beschriebenen sind die für diese Arbeit relevanten Eigenschaften.

2.4 Flächenmodus bei bestehenden Screenreadern

Windows-Screenreader bieten schon seit Jahren einen Flächenmodus zum Arbeiten an. Es ist also keine neue Erfindung und viele Anwender erwarten von ihrem Screenreader eine Zeilenorientierte Darstellungsform. Bei kommerziellen Screenreadern wie JAWS, Blindows, COBRA und SuperNova gehört ein Flächenmodus zum Standard-Repertoire.

In Handbüchern zur Bedienung der einzelnen Screenreader wird der Flächenmodus wie folgt beschrieben:

COBRA – Handbuch

„Alternativ zu Elementdarstellung, in der immer nur das eine Element angezeigt wird, gibt es mit der Zeilendarstellung eine Ansicht, in der Sie immer alle Elemente auf der Braillezeile angezeigt bekommen, die sich am Bildschirm in der gleichen Zeile befinden. Statt der oben beschriebenen Kürzel werden die Elemente mit kurzen Zeichen ‚geschmückt‘, sodass Sie sofort erkennen können, um was für eine Art es sich handelt. Im folgenden zeigen wir anhand von Beispielen, die verschiedenen Elementarten:

- E[Document1.doc]: ein Eingabefeld
- <OK>: eine Schaltfläche
- [X] Sprache aktiv: ein aktiviertes Kontrollfeld
- [] Sprache aktiv: ein nicht aktiviertes Kontrollfeld
- (x) Montag: ein aktivierter Auswahlschalter
- () Montag: ein nicht aktivierter Auswahlschalter
- [Allgemein]: eine Registerkarte

“[COB-01]

JAWS – Bedienschulung

„Im Flächenmodus repräsentiert die Braillezeile die Zeile auf dem Bildschirm, die den Focus besitzt. In diesem Modus führen die Routingtasten einen Mausklick an der Stelle aus, an der sie gedrückt werden. Standardmäßig kann der Braille Cursor im Flächenmodus über den ganzen Bildschirm bewegt werden, ohne dabei die Position des aktiven Cursors zu verändern. Dieses Feature kann umgeschaltet werden, so daß die Braillezeile den aktiven Cursor bewegt. Sie können Mausklicks mit Hilfe der Routingtasten überall dort ausführen, wo dies von Microsoft Windows erlaubt wird.“[JAW-01]

Ogleich diese Arbeitsweise unter Windows weit verbreitet ist, gibt es noch nichts vergleichbares unter Linux. Die Ursachen für diesen Sachverhalt sind vielfältig.

Windows-Screenreader arbeiten mit einem Off-Screen-Modell (OMS), welches die Bildschirminhalte nachbildet und für den Screenreader zur Abfrage bereitstellt. Das OMS ist Teil des Screenreaders und geht damit uneingeschränkt auf die Bedürfnisse der Hilfsmittelsoftware ein. Mit diesen Voraussetzungen ist es relativ einfach möglich, einen Flächenmodus zu entwickeln. Im Gegensatz dazu kommen Screenreader unter Linux ohne OMS aus. Sie bekommen die notwendigen Informationen von der Accessibility Schnittstelle, die die Bildschirminhalte in einem Accessibility Baum nachbildet. Diese Art der Informationskette ist nicht dafür ausgelegt, dass der Screenreader große Mengen an Daten selber verwaltet. Er ist darauf angewiesen, diese zur Laufzeit aus dem Baum auszulesen. Der Baum wiederum ist so aufgebaut, dass er eine zeilenweise Anzeige von Bildschirmausschnitten nicht unterstützt.

Das Konzept des Flächenmodus kann also nicht einfach von Windows-Screenreader übernommen werden. Es muss eine Lösung gefunden werden, die den Gegebenheiten unter Linux entspricht.

Bisher wurde der Flächenmodus immer nur für die Anzeige einer einzelnen Zeile umgesetzt. Das zu erarbeitende Konzept für Linux-Screenreader soll so allgemein aufgestellt werden, dass es die Darstellung mehrerer Zeilen gleichzeitig unterstützt.

2.5 Forschungsarbeiten zu Flächendisplays

Der Zugang zu grafischen Informationen für Blinde mittels eines Screenreaders ist noch stark eingeschränkt. Momentan sind Brailledisplays, die lediglich eine Textzeile darstellen können, der Stand der Technik. Für die Abbildung von grafischen Konstrukten sind jedoch große taktile Displays notwendig. Die bekanntesten Forschungsaktivitäten hierzu sind der DMD 120060 von Metec, die Dot View Serie von KGS, das NIST Display und das GWP von Handytech. Diese Projekte gingen allerdings nicht über den Status eines Prototypens hinaus. Die Ursachen sind vor allem in den Kosten und der fabrikanten Herstellung solcher Displays zu suchen. Ein Brailledisplay besteht aus vielen einzelnen Pins, die in einer Matrix angeordnet sind. Alle Pins des Displays müssen eine gewisse Resistenz gegen Druck aufweisen, damit die gesetzten Stifte ertastbar sind. Dies ist eine Herausforderung für vertikale Braille-Module, wie sie in größeren Displays verarbeitet werden müssen.

Außerdem sollte die Aktualisierungsrate der Matrix nicht langsamer sein, als bei herkömmlichen Braillezeilen. Dies ist notwendig, um ein angenehmes Arbeiten ohne störende Verzögerungen zu ermöglichen. Für einen flüssigen

Arbeitsablauf sollten des weiteren Bewegungen zwischen dem Brailledisplay und der Tastatur gering gehalten werden. Dies kann unter anderem dadurch erreicht werden, dass Steuertasten nicht außerhalb des tatkilen Bereiches angeordnet werden. Routing-Tasten, wie sie bei aktuellen Braillezeilen eingesetzt werden, sind für größere Displays unpraktikabel. Die Entfernung zwischen Taste und Inhalt ist zu groß und führt zu Unterbrechungen in der Arbeit.

Der Bezug zwischen dem taktil angezeigten Objekt und den Steuertasten muss aus der Anwenderperspektive heraus ermittelt werden. Mehrere Technologien könnten eingesetzt werden, um die Finger-Positionierung des Anwenders zu erkennen:

1. Der Anwender trägt ein Steuergerät am Finger.
2. Die Finger werden über eine Videokamera von oben beobachtet.
3. Berührungen der Oberfläche werden durch eine Videokamera von unten beobachtet.
4. Berührungen an den erhobenen Pins werden ermittelt.
5. Es wird eine berührungsempfindliche Oberfläche eingesetzt.

Diese Methoden können folgendermaßen beurteilt werden:

1. Ein Extragerät am Finger ist nicht sehr komfortabel und anwenderfreundlich.
2. Eine Videokamera kann die Stärke des ausgeübten Druckes auf die Oberfläche nicht feststellen.
3. Eine Kamera von unten ist aufgrund von mechanischen Grenzen nicht möglich.
4. Die Fingerposition ist nicht erkennbar, wenn keine Pins gesetzt sind.
5. Die bevorzugte Eingabemethode ist ein berührungsempfindliche Oberfläche. Diese muss die Positionierung der Eingabe mindestens mit einer Auflösung entsprechend der Fingergröße oder der Größe eines Braillezeichens erkennen können. Dadurch kann der dargestellte Inhalt direkt manipuliert werden. Außerdem wird die Eingabe von Kommandos mit Hilfe von Gesten möglich. Bei der Erkundung des Displays können die Handflächen von mehreren Nutzern involviert sein.

Diese Analyse zeigt, dass eine berührungsempfindliche Oberfläche zur Manipulation der taktilen Objekte zu bevorzugen ist.

Nachfolgend werden zwei Forschungsprojekte zum Thema Brailledisplay beschrieben, die derzeit durchgeführt werden.

2.5.1 NC-State-Display

An der North Carolina State University wird derzeit ein Vollbild-Display für Blinde entwickelt. Das Ziel ist ein kostengünstiges Display zu erschaffen, welches die Darstellung von Blindenschrift erlaubt. Bei diesem Projekt werden elektroaktive Polymere als Material verwendet. Die Bildpunkte werden hydraulisch angehoben und über ein mechanisches Einrasten wird das Absinken beim Druck mit dem Finger verhindert. Das Material reagiert sehr schnell und ermöglicht damit zum Beispiel das schnelle Scrollen durch ein Dokument. Die Größe des Displays erlaubt es außerdem, taktile Bilder zu präsentieren.

Das hydraulische Heben der Punkte funktioniert bereits. Im nächsten Schritt soll eine Lösung für das mechanische Einrasten erforscht werden. Bis zum Frühjahr 2011 soll ein erster Prototyp für den Gesamtmechanismus entstehen.

Die Prognose für das NC-State-Display geht von einem kostengünstigen und einfach zu bedienenden Display aus. [NCS-01]

2.5.2 Hyperbraille

Das Hyperbraille-Projekt startete Mitte 2007 und basierte auf den vielversprechenden Resultaten des BrailleDis Projektes von 2001. Die immer noch existierende Kluft zwischen den Inhalten, die visuell zur Verfügung stehen und denen, die taktil präsentiert werden, war eine starke Motivation für die Arbeit in diesem Projekt. Finanziert wurde das Vorhaben vom deutschen Bundeswirtschaftsministerium. Das Projekt lief 2010 aus und wurde bereits auf mehreren Messen und Konferenzen vorgestellt. Die beteiligten Projektpartner sind: Metec AG, FH Papaenmeier GmbH & Co. KG, IMS-Chips, Institut für Informatik an der TU Dresden und Institut für Informatik an der Universität Potsdam.

Das Projekt-Ergebnis ist ein Prototyp mit 7200 brührungsempfindlichen Stiften, die in einer Matrix von 60x120 angeordnet sind. Zusätzlich zur Entwicklung der Hardware wurde Software zur Ansteuerung des Displays und Filter für die Darstellung von verschiedenen Anwendungsprogrammen entwickelt. Dabei lag der Schwerpunkt auf in der Arbeitswelt gängiger Office- und Internet-Anwendungen.

Nach Aussage der Projekt-Beteiligten ersetzt dieses Brailledisplay 12 konventionelle Braillezeilen und vergrößert die Menge der beidhändig wahrnehmbaren Informationen. Außerdem wird der Zugang zu grafischen Informationen ermöglicht. Im Idealfall können vollständige Textabsätze, Tabellen, Menüs und andere Elemente angezeigt werden. In grafischen Darstellungen sind zum Beispiel Wegepläne, technische Zeichnungen, elektrische Schaltpläne und UML-Diagramme denkbar. Damit wird Sehgeschädigten der Zugang zu weiteren Berufen ermöglicht. Ein Schulungskonzept für die

Bedienung des Brailledisplays wird von der Deutschen Blindenstudienanstalt e.V. entwickelt.

Hardware

Der entwickelte Prototyp trägt den Namen BrailleDis 9000 besteht aus 720 Braillemodulen mit je 10 piezoelektrisch aktivierten Stiften. Diese Module sind vertikal und nicht wie bei Braillezeilen bisher üblich horizontal angeordnet. Das Brailledisplay wird über einen USB 2.0 Anschluss mit dem PC verbunden. Die Stifte verfügen über sensitive Eigenschaften, die Interaktionen zwischen Anwender und Software per Fingerklick ermöglichen. Denkbar sind zum Beispiel das Anfertigen einfacher Zeichnungen oder Drag-and-Drop Techniken in blindengerechten Arbeitsweisen.

Im Hyperbraille-Projekt wurde das im BrailleDis-Projekt entwickelte Flächendisplay überarbeitet. Der Schwerpunkt lag dabei in der Entwicklung von vertikal angeordneten Braillemodulen. Die Verwendung von 2x5 Pin großen Modulen ermöglicht einen leichten Austausch von beschädigten und defekten Teilen. Jedes Modul hat einen separaten Sensor, eine unabhängige Antriebs-Elektronik und ist gesondert mit dem Daten-Bus verbunden. Dadurch beeinflussen defekte Zellen nicht die restlichen Module, es sei denn, sie erzeugen einen Kurzschluss im Daten-Bus. Über den Daten-Bus ist eine Auffrischung der Anzeige alle 50ms durchführbar. Die Trägheit der piezoelektronischen Elemente begrenzt die Auffrischungsrate allerdings auf 150ms. Der modulare Aufbau ermöglicht verschiedene Dimensionen für das Display und die erhobenen Stifte besitzen eine hohe Druckresistenz.

Off-Screen-Modell

Herkömmliche Off-Screen-Modelle sind nicht ausreichend für taktile grafische Displays. Aus diesem Grund wurde im Hyperbraille-Projekt ein erweitertes Off-Screen-Modell für Flächendisplays entwickelt. Dieses OSM besteht aus einem azyklischen Graphen, der eine baumähnliche Formatierung hat (OSM-Baum). Für die Darstellung von GUI Komponenten standen einheitliche Objekte mit verschiedenen Parametern oder individuelle Objekte für jede Komponente zur Diskussion. Die vielversprechendste Herangehensweise ist die Verwendung von individuellen Objekten. Diese werden in mehrere Typen eingeteilt:

- einfache Widgets
- komplexe Widgets
- Kontainer

Einfache Widgets können nicht weiter untergliedert werden und sind sichtbare GUI Komponenten. Dies sind vorwiegend Elemente, die die Präsentation

von atomaren Werten wie Strings, Datumsangaben, Zeitangaben, numerischen und booleschen Werten erlauben. Zu solchen Widgets gehören unter anderem Checkboxes, Radiobuttons und Textfelder. Komplexe Widgets sind sichtbare GUI Komponenten, die die Darstellung von multiplen Werten erlauben. Dazu gehören Tabellen, Listen und Bäume. Komplexe Widgets setzen sich aus einfachen und anderen komplexen Widgets zusammen. Container wiederum werden zur Gruppierung von Widgets verwendet. Dies kann zum Beispiel eine Menüleiste sein, die mehrere Menüs umschließt.

Für die Unterstützung von verschiedenen Ausgabegeräten implementieren die einzelnen Widgets eine Reihe von Schnittstellen, die die entsprechende Präsentation auf dem Ausgabegerät vorgeben. Über diese Schnittstellen werden Ausgabe-Operationen wie Speech-Rendering und Brailledarstellung definiert. Derzeit werden drei Schnittstellen von den Widgets implementiert: `IVisual`, `IAudible` und `ITactual`.

Der OSM-Baum wird durch den OSM-Manager administriert. Dieser bietet Schnittstellen zum Speichern und Abfragen von Daten. Der OSM-Baum kann erweitert werden, indem Knoten zum Baum hinzugefügt werden. Auf diese Art können existierende Teilbäume eingefügt werden. Dies ist entscheidend, wenn Zweige von ihrem Parent-Knoten abgekoppelt und dann einem anderen Knoten zugeordnet werden sollen. Wird ein Knoten entfernt, so werden auch alle seine Child-Knoten aus dem Baum entfernt. Das Bearbeiten eines Knotens hat im Gegensatz dazu keinerlei Auswirkungen auf den Aufbau des Baumes. Hierbei werden lediglich die Eigenschaften verändert.

Der Zugang zum OSM kann auf verschiedenen Wegen erlangt werden. Zum einen können sich Programme beim OSM als Listeners registrieren. Diesen wird automatisch jede Änderung am OSM mitgeteilt. Zum anderen generiert der OSM-Manager Ereignisse bei jeder Modifikation des OSM. Diese Ereignisse enthalten Informationen zu den betroffenen Knoten und über die Art der Änderung (addition, deletion, editing). Interessierte Anwendungen können diese Ereignisse abfangen und verarbeiten.

Weiterhin muss es Anpassungsmöglichkeiten für Software geben, die nicht durch die Accessibility Schnittstellen unterstützt werden. Es sind anwendungsspezifische Filter zu entwickeln, um die entsprechenden Daten dem OSM zur Verfügung zu stellen. Ein konfigurierbares Wildcard-Widget soll Entwicklern von nicht-standardisierten grafischen Elementen ermöglichen, diese accessible zu gestalten.

Taktile Abbildung von GUIs

In Abschnitt 2.1 wurde gezeigt, dass Blinde in der Lage sind komplexere grafische Darstellungen zu entziffern. Als nächstes erfolgt eine Beschreibung der Konvertierung gebräuchlicher visueller Darstellungen zu taktilen interaktiven Präsentationsformen.

Ein direktes mapping der visuellen 2-dimensionalen Informationen auf

taktile Displays ist trivial. Dies ist aber für Blinde nicht nutzbar, da das visuelle Konzept nicht mit taktilen Ausgaben vereinbar ist. Zusätzlich kommen Beschränkungen durch die Hardware hinzu. Ein Flächendisplay hat eine geringere Auflösung, als ein Bildschirm. Diese entspricht gerade einmal 10dpi bei der Pin-Matrix. Deshalb können nur Fragmente des Bildschirms angezeigt werden. Die Konsequenz ist eine extrem geringe Informationsdichte auf den Flächendisplays.

Für die Erstellung von visuellen Designs gibt es umfangreiche Studien, die beschreiben, wie bestimmte Aktionen aussehen sollten. Shneiderman's information-seeking mantra gibt zum Beispiel an, wie eine Zoom-Funktion aufgebaut und wie bestimmte Informationen bei Bedarf angezeigt werden sollten. Ähnliche Studien müssen für taktile Anzeigen erstellt werden. Bei solchen Anzeigen müssen verschiedenen Nutzergruppen berücksichtigt werden. Zum einen gibt es Anwender, die stark vertraut mit den bisherigen Arbeitstechniken sind. Zum anderen gibt es Anfänger, die oftmals mit nur einer Hand arbeiten, anstatt wie bei den neuen Displays angebracht mit zwei. Weiterhin wird es Nutzer geben, die sich schnell mit dem neuen Medium und entsprechenden Arbeitstechniken vertraut machen werden. Um diese heterogene Nutzergruppen zu berücksichtigen muss eine Balance zwischen der Menge der angezeigten Informationen und dem freien Platz auf dem Display gefunden werden. Das Ziel dabei ist, eine Informationsflut zu vermeiden.

Taktiler Erfassen von Inhalten ist sequentiell und erlaubt daher nur eine minimale Vorverarbeitung der Daten. Desweiteren behindert die Arbeit mit zwei Geräten, dem Flächendisplay und der Tastatur, die Performance des Anwenders, da der Wechsel zwischen den Geräten den Arbeitsfluss unterbricht. Es ist notwendig, ergonomische Operationen zu erstellen, die zum Beispiel Eingaben über das Display erlauben.

Zur taktilen Orientierung auf dem Flächendisplay wird dieses in vier Bereiche eingeteilt: Header-, Body-, Struktur- und Detailregion. Diese Regionen werden durch eine Linie mit gesetzten Pins von einander getrennt und können auf Wunsch ausgeblendet werden. Der Header zeigt die Eigenschaften und den Status einer Anwendung an. Er nimmt ein oder zwei Zeilen am oberen Rand des Displays ein und entspricht dem visuellen Konzept von Titel, Menü und Statuszeile. Der Body ist die primäre Region und kann nicht ausgeblendet werden. Er wird zur Anzeige der Inhalte einer Anwendung verwendet und beansprucht den meisten Platz auf dem Display. In der Detail-Region werden Einzelheiten zum fokussierten Element gegeben. Diese können Tastaturkürzel, Statusinformationen, Indizes, Schriftfarben und andere Details enthalten. Die Detail-Region ist am unteren Rand des Displays angeordnet und kann ebenfalls 1 bis 2 Zeilen enthalten. Der letzte Bereich ist die Struktur-Region. Diese befindet sich entweder auf der linken oder rechten Seite des Bodys und hat eine Breite von zwei Braillezeichen. Hier werden strukturelle Informationen kodiert, die Aussagen zur Forma-

tierung oder Hierarchie der abgebildeten Elemente erlauben. Damit besteht die Möglichkeit, Überschriften, Kommentare oder Rechtschreibfehler effizient lokalisieren zu können.

Zusätzlich zu der Aufteilung des Flächendisplays werden vier Sichten definiert. Damit ist es möglich, die Inhalte auf verschiedene Weise zu präsentieren. Folgende Sichten wurden festgelegt: Layout-, Umriss-, Symbol- und Arbeitssicht.

In der Layoutdarstellung werden die Pixel-Informationen erhalten und auf die geringe Auflösung des Flächendisplays angepasst. Text erscheint nicht in Brailleschrift, sondern in einer taktilen Version der Schwarzschrift. Damit kann der originale Bildschirminhalt möglichst Detailgetreu erkundet werden. Die Umriss-Sicht hingegen liefert eine abstrakte Präsentation der Bildschirm-Objekte. Über die Darstellung einfacher geometrischer Formen wird die Struktur der Elemente reflektiert. Textuelle Informationen werden ausgeblendet. Damit kann sich der Anwender einen Überblick über komplexe Objekte wie Anwendungen oder Dokumente verschaffen, ohne zunächst auf die Details zu achten. Die Symbolsicht ist der Layoutdarstellung sehr ähnlich. Auch hier wird versucht die relative Positionierung der Elemente beizubehalten. Allerdings erscheint Text in Brailleschrift und Grafiken werden in Form von vordefinierten Symbolen und Formen angezeigt. Für einen schnellen Arbeitsfluss ist die Arbeitssicht geeignet. Diese spiegelt am ehesten den Strukturmodus bisheriger Screenreader wieder. Das fokussierte Element wird mit so vielen Details wie möglich abgebildet und grafische Details werden weggelassen. Es wird eine logische und strukturierte Präsentation der Bildschirminhalte auf reiner Textbasis gegeben.

Mit den Sichten werden neue Arbeitsmethoden für das Flächendisplay eingeführt. Dies kann problematisch bei der Akzeptanz des neuen Displays in der Blinden-Kommunity sein. Intensive Schulungen müssen diesem Problem entgegensteuern. Das Flächendisplay ermöglicht endlich den Zugang zu grafischen Informationen auf dem Computer. Das diese von Blinden interpretiert werden können wurde bereits in mehreren psychologischen und neurowissenschaftlichen Studien belegt (siehe Abschnitt 2.1).

Kapitel 3

Konzeption des Flächenmodus

3.1 Analyse der Grafischen Oberfläche

Basierend auf dem GNOME Design Guide [GNO-1] soll hier der grundlegende Aufbau der GNOME Oberfläche analysiert werden. Diese Analyse dient dem Verständnis aller vorkommenden grafischen Elemente, ihrer Anordnung und möglicher Probleme für die Präsentation in einem Braille-Flächenmodus.

Zum besseren Verständnis und zur Wahrung des Überblicks im Flächenmodus werden zunächst folgende Festlegungen getroffen:

1. Die GNOME GUI wird in logische Einheiten eingeteilt.
2. Im Flächenmodus wird jeweils eine logische Einheit zeilenweise präsentiert.

3.1.1 Desktop

Der GNOME Desktop besteht aus vier logischen Einheiten: dem Desktop, dem oberen Kantenpanel, dem unteren Kantenpanel und verschiedenen Fenstern.

Der Desktop kann zur Ablage verschiedenster Items genutzt werden. Da GNOME versucht diese entsprechend ihrem Inhalt zu präsentieren, haben diese Items die unterschiedlichsten Größen. Außerdem können sie beliebig angeordnet sein. Dies macht die Einteilung in Zeilen nicht immer eindeutig. Der Nutzerdesktop ist im AT-SPI Baum unter dem Application-Knoten (1. Ebene) mit dem Namen `nautilus` zu finden. Unterhalb dieses Knotens gibt es mehrere Child-Knoten, unter anderem einen mit dem Namen `x-nautilus-desktop` und der Rolle `frame`. In diesem Teilbaum ist in der 8. Ebene ein Knoten mit der Rolle `layered pane` zu finden. Dies ist das

direkte Parent-Element zu allen Symbolen, die auf dem Desktop abgelegt sind. Alle Child-Knoten zu diesem `layered pane` haben die Rolle `icon` und sind Blätter des Baumes.

Die Kantenpanel befinden sich, wie schon im Namen angedeutet, am oberen und unteren Fensterrand. Standardmäßig bilden sie jeweils eine Zeile. Über verschiedene Einstellmöglichkeiten können sie allerdings auch an den linken und rechten Rand verschoben werden. Dann werden die Inhalte nicht mehr in einer Zeile, sondern in einer Spalte präsentiert. Dies bedeutet, das ein Kantenpanel, welches sich am linken oder rechten Bildschirmrand befindet, durch viele Zeilen widergespiegelt wird. Die Kantenpanel sind im AT-SPI Baum unter dem Application-Knoten mit dem Namen `gnome-panel` angeordnet. Dieser Knoten hat zwei Children mit der Rolle `frame`, die jeweils den Namen des Kantenpanels tragen. Drei Ebenen unter diesen Frames befindet sich dann jeweils ein weiterer Knoten mit dem gleichen Namen, diesmal allerdings mit der Rolle `panel`. Dies ist das Parent-Element zu weiteren Panels, die die Kantenpanel in Abschnitte einteilen. Die Teilbäume dieser Abschnitte enthalten dann die Repräsentation der Widgets, die auf den Kantenpanels dargestellt sind.

Fenster sind wohl die vielseitigsten Gebilde auf dem GUI. Sie können alle Arten von grafischen Elementen (Widgets) enthalten. In den nachfolgenden Ausführungen werden Fenster in zwei Kategorien eingeteilt: in Anwendungs- und Spezialfenster. Trotz ihrer Vielseitigkeit wird sich zeigen, dass alle Fenster einen gewissen Grundaufbau haben, der sich noch einmal in verschiedene logische Einheiten unterteilen lässt.

3.1.2 Anwendungsfenster

Ein Anwendungsfenster dient meist dem direkten Arbeiten mit der jeweiligen Anwendung. In dem Fenster werden häufig komplexe Dokumente angezeigt, auf denen das Hauptaugenmerk der Nutzer liegt.

In der Abbildung 3.1 wird der Aufbau eines solchen Fensters schematisch wiedergegeben. Im Einzelnen besteht es aus folgenden Bestandteilen:

Titelleiste

Die Titelleiste besteht aus genau einer Zeile. Allerdings kann diese Zeile nur im Erkundungsmodus eines Screenreaders (siehe Kapitel 1) angezeigt werden, da die Titelleiste nie direkt den Fokus erhalten kann. Je nach Nutzervorlieben enthält die Leiste auf der linken oder rechten Seite eine kleine Reihe von Buttons, die zum Schließen, Maximieren, Minimieren und Roll-up des Fensters gedacht sind. Leider sind diese Buttons nicht über die Accessibility Schnittstelle sichtbar und damit für die AT-Software nicht vorhanden. Die Titelzeile besteht für den Screenreader also lediglich aus dem Fensternamen. Diesen erhält man, indem man den Namen des Vorfahren-Knotens auf

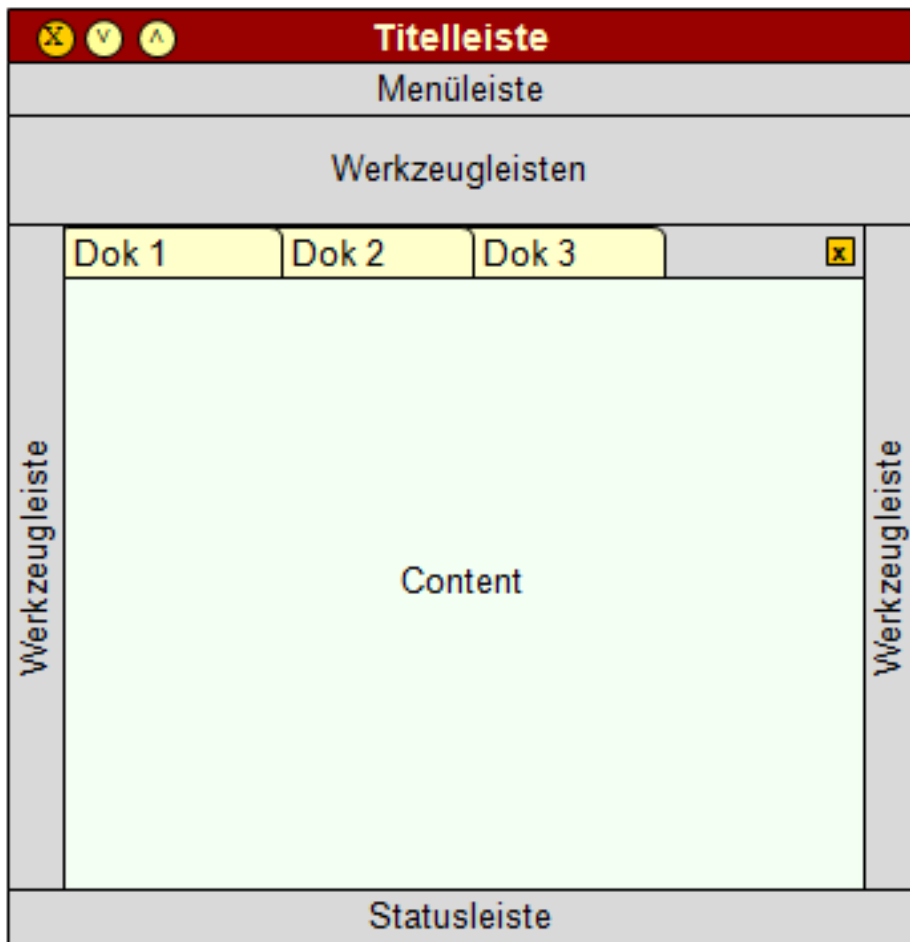


Abbildung 3.1: Schematische Darstellung eines Anwendungsfensters

der 2. Ebene ermittelt, da dieser Knoten das Fenster an sich repräsentiert.

Menüleiste

Die Menüleiste ist im Allgemeinen die Zeile direkt unter der Titelleiste. Bei umfangreichen Menüs und schmalen Fenstern kann es passieren, dass die Menüleiste mehrere Zeilen umspannt. Eine Menüleiste besteht aus einer Reihe Menütiteln, die bei Aktivierung in ein vertikal angeordnetes Untermenü aufklappen. Solch ein Untermenü kann wiederum andere Untermenüs, einfache Menüpunkte und Auswahlmenüs (radio button, check box) enthalten. Durch die vertikale Anordnung in einem Untermenü entspricht jeder Menüeintrag einer einzelnen Zeile. Eine Ausnahme bildet dabei ein Untermenü in einem Untermenü. Wird dieses ausgewählt, klappt zur Seite ein neues Menü auf. Dadurch wird die Zeile noch durch den ersten Menüeintrag

des neuen Menüs ergänzt. Getrennt werden die beiden in der Zeile angezeigten Menüeinträge durch einen Pfeil: Untermenü → Menüpunkt

Ein Menüeintrag besteht oftmals aus einem grafischen Symbol, seinem Namen und einem Tastaturkürzel. Für die Braille-Darstellung wird im Flächenmodus der Name und das Tastaturkürzel ausgewählt. Vertikale Trennlinien werden als eigenständige Zeile behandelt.

Eine andere Art von Menü ist das Kontextmenü, welches ein zu einem Objekt gehörendes, vertikal angeordnetes Menü darstellt. Es unterliegt den gleichen Gesetzmäßigkeiten wie ein zu einer Menüleiste gehörendes Menü.

Der Teilbaum, der eine Menüleiste repräsentiert beginnt immer mit dem Accessible mit der Rolle `menu bar`. Alle Child-Elemente von diesem Knoten haben die Rolle `menu` und den Namen des jeweiligen Menütitels. Unterhalb dieses Knotens sind die Menüeinträge angeordnet. Diese können folgende Rollen haben: `menu item`, `menu separator` oder `check menu item`. Der Name des jeweiligen Accessible's ist auch der Name des Menüeintrages. Lediglich die Knoten mit der Rolle `menu` können wiederum Child-Knoten haben. Alle anderen sind Blätter des Baumes. Ein Kontextmenü hat als Start-Knoten statt der Rolle `menu bar` die Rolle `popup menu`.

Werkzeugleisten

Werkzeugleisten können horizontal oder vertikal angeordnet sein. Die GNOME Human Interface Guidelines [GNO-1] empfehlen eine horizontale Anordnung, da diese eher der Arbeitsweise von sehenden Nutzern entspricht. Eine Anwendung kann mehrere Werkzeugleisten haben, die eine Reihe von teils unterschiedlichen Elementen enthalten kann. Das am häufigsten vorkommende Element ist ein Push Button (GNOME Übersetzung: Druckknopf). Visuell kann ein Element in der Werkzeugleiste durch ein Symbol und/oder Text angezeigt werden. Aber auch Pfeile (zum Anzeigen versteckter Funktionen) und Eingabe- und Auswahlfelder sind durchaus üblich. Für die Braille-Darstellung ist es unerheblich, ob nur ein Symbol oder ein Symbol in Kombination mit Text angezeigt wird. In Braille wird immer der Text verwendet. Die Abbildung der in Werkzeugleisten gebräuchlichen Pfeile ist nicht möglich, da diese Information in der AT-SPI Schnittstelle fehlt.

Der Knoten, bei dem der Teilbaum beginnt, hat die Rolle `tool bar`. Es kann mehrere solcher Knoten unter einem Fenster-Knoten geben, für jede vorhandene Werkzeugleiste einen. Elemente mit der Rolle `tool bar` sind Geschwister-Knoten zu einander. Abhängig von der Anwendung kann der Teilbaum nur noch eine weitere Ebene umfassen (zum Beispiel bei OpenOffice.org) oder weit verzweigt sein (zum Beispiel bei Evolution).

Leiste mit geöffneten Dokumenten

Eine Reihe von Anwendungen enthalten eine Leiste, in denen in Form von Seitenreitern die geöffneten Dokumente angezeigt werden. Diese sind meist in einer Zeile angeordnet. Nur wenn zu viele Dokumente geöffnet sind, kann es in einzelnen Anwendungen dazu führen, dass mehrere Zeilen verwendet werden.

Der interne Aufbau einer Dokumentenleiste ist abhängig von der Anwendung und kann nicht verallgemeinert werden. Aus diesem Grund wird es als Teil des Inhaltsbereiches betrachtet und nicht als Spezialfall.

Inhaltsbereich

Der Begriff Inhaltsbereich wird für einen Teil der Anwendung mit beliebigem Inhalt verwendet. Dieser wird in den unterschiedlichsten Mustern angeordnet sein.

Statusleiste

Die Statusleiste schließt das Fenster an der unteren Kante ab. Sie besteht aus genau einer Zeile und kann unterschiedliche Elemente enthalten.

Eine Statusleiste wird im AT-SPI Baum durch ein Element mit der Rolle `statusbar` repräsentiert. Abhängig von der Anwendung kann der Teilbaum unterhalb dieses Knoten unterschiedlich aufgebaut sein.

Festlegung der logischen Einheiten

Als logische Einheiten eines Anwendungsfensters werden folgende Bestandteile betrachtet:

- Titelleiste
- Menüleiste, Kontextmenü
- Werkzeugleisten
- Content
- Statusleiste

3.1.3 Spezialfenster

Spezialfenster besitzen häufig nur für kurze Zeit die Aufmerksamkeit des Nutzers. Sie dienen dazu Warnungen und Entscheidungsfragen an den Anwender heranzutragen, Einstellungen vorzunehmen, über den Fortschritt von ausgeführten Aktionen zu berichten und Hilfestellung beim Arbeiten zu leisten. Auch bei diesen Fenstern kann ein gewisser Grundaufbau festgestellt werden.

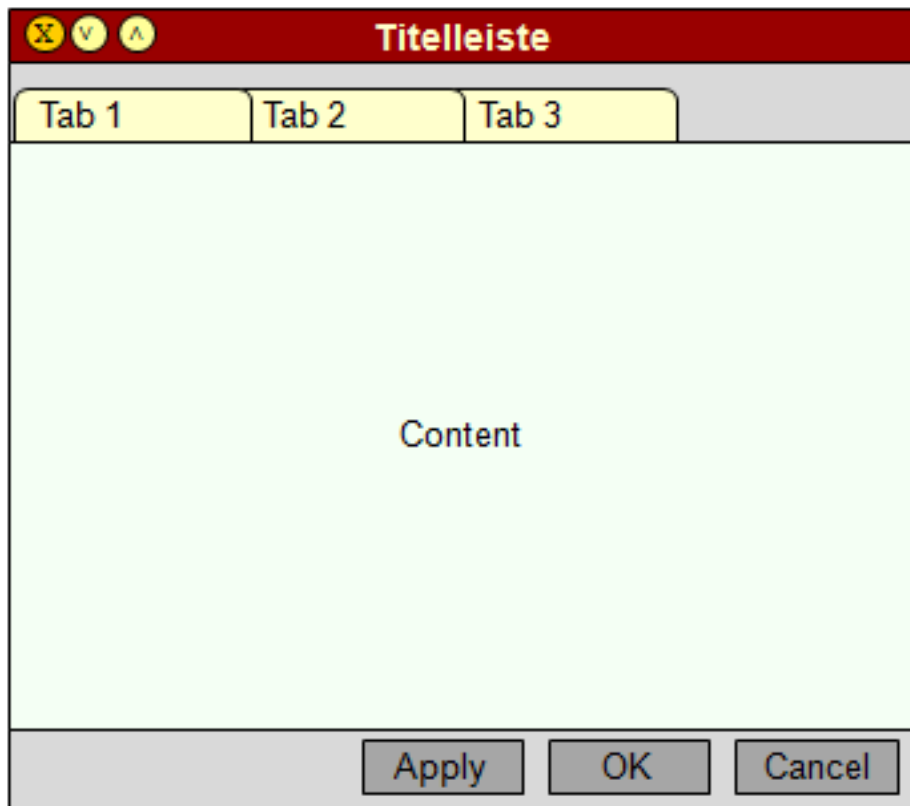


Abbildung 3.2: Schematische Darstellung eines Spezialfensters

Abbildung 3.2 zeigt den schematischen Aufbau eines solchen Fenster. Spezialfenster haben keine Menü- Werkzeug- und Statusleisten. Im Folgenden soll kurz der Aufbau der einzelnen Fenster betrachtet werden.

Utility Windows (Einstellungsfenster)

Utility Windows sind Einstellungs- oder Eigenschaftsfenster. Ihr Aufbau ist meist komplex und vielseitig. Diese Art von Fenster hat eine Titelleiste einen Inhaltsbereich und zum Abschluss eine Reihe mit Push Buttons. Im Inhaltsbereich kann sich eine Leiste mit Seitenreitern befinden, oder eine baumartige Struktur auf der linken Seite des Fensters. Diese Elemente dienen zum Umschalten zwischen den verschiedenen Seiten des Dialoges.

Toolbox (Werkzeugkasten)

Ein Werkzeugkasten entspricht im Wesentlichen einer Werkzeugleiste. Allerdings ist der Kasten vom Anwendungsfenster separiert. Werkzeugkästen haben keine Titelleiste. Ihre Elemente bestehen meist aus Push Buttons,

können aber auch andere Widgets enthalten. Diese sind in einem Gitter angeordnet.

Alerts (Warnungen)

Alerts sind kleine Fenster, die dem Anwender eine Nachricht überbringen. Sie erscheinen immer im Vordergrund, da sie sich meist auf wichtige Systemereignisse beziehen. Diese Art von Fenster haben keine Titelleiste. Auf der linken Seite ist ein Symbol platziert, welches die Art der Warnung angibt (Frage, Kritisch, Information, ...). Rechts daneben sind mehrere Labels angeordnet, die die Nachricht enthalten. Das erste Label ist der primäre Text, der eine kurze Zusammenfassung enthält. Alle weiteren Labels gehören zum sekundären Text, der die Nachricht ausführlich beschreibt. Das Fenster wird durch eine Reihe von Buttons am unteren Rand abgeschlossen.

Progress Bar Window (Fortschrittsbalken)

Fortschrittsbalken zeigen den Verlauf einer Aktion wie das Kopieren von Dateien an. Sie können Teil eines Anwendungsfensters sein oder in einem eigenen Dialog angezeigt werden. Ein Dialog zur Anzeige eines Fortschrittsbalkens besteht aus einer Titelleiste, einem primären und verschiedenen sekundären Texten, dem Fortschrittsbalken und einer Reihe mit Buttons.

Dialog (Dialoge)

Dialoge können die unterschiedlichsten Aufgaben erfüllen. Sie bestehen aus einer Titelleiste, beliebigem Inhaltsbereich und einer Leiste mit Buttons.

Spezialfenster haben eine der folgenden Rollen: `dialog`, `alert` oder `file chooser`. Wenn eine Titelleiste vorhanden ist, hat sie die gleichen Eigenschaften wie bei einem Anwendungsfenster. Die Zeile mit den Push Buttons befindet sich direkt unterhalb des Fenster-Knotens. Eventuell ist sie noch einmal in ein `Filler` Element gekapselt. Alle Buttons der Zeile sind direkte Geschwister-Knoten zueinander. Auf gleicher Ebene mit dieser Zeile befindet sich meist auch der Einstiegs-Knoten zum Teilbaum, der den Inhaltsbereich repräsentiert. Enthält der Inhalt Seitenreiter, so werden diese durch einen Knoten mit der Rolle `Page Tab List` angezeigt. Die Children dieses `Accessibles` haben die Rolle `Page Tab`. Mit jedem `Page Tab` Element startet ein Teilbaum, der beliebige Inhalte enthalten kann. Nur ein Child-Element der Seitenreiterliste ist jeweils aktiv.

Alternativ zu Seitenreitern kann am linken Fensterrand eine Art Auswahlliste vorhanden sein, die das Umschalten zwischen verschiedenen Inhaltsseiten ermöglicht. Diese Liste kann entweder ein `Tree`, ein `Tree Table`, ein Element mit der Rolle `List` oder eine einspaltige Tabelle sein.

Festlegung logischer Einheiten

Aufgrund der vorangegangenen Analyse werden folgende logische Einheiten in einem Spezialfenster festgelegt:

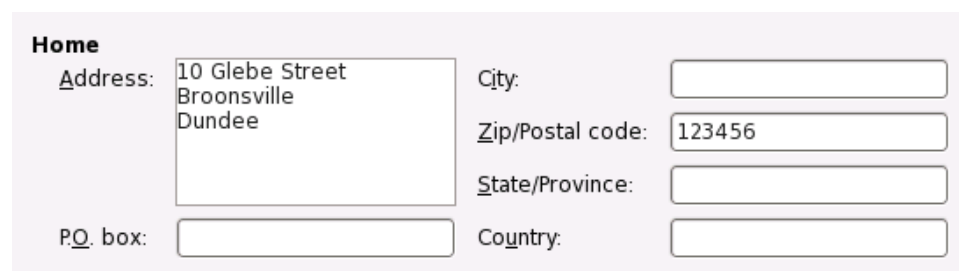
- Titelleiste
- Leiste mit Tabs
- Baum, Baumtabelle, Liste oder einspaltige Tabelle am linken Fenster-
rand. Diese sind durch ihre Positionierung von anderen Elementen mit
gleicher Rolle zu unterscheiden.
- Inhaltsbereich
- Leiste mit Buttons

3.1.4 Besonderheiten bei einfachen Widgets

Die GNOME Human Interface Guidelines [GNO-1] geben folgende Empfehlung: „Make sure that dialog elements are clearly-aligned“. Leider sind trotzdem nicht immer die Zeilen, in denen die Elemente angeordnet sind, eindeutig zu identifizieren. Die Ursachen können zum Einen in einem nachlässigen Design der Anwendung liegen, zum Anderen aber auch durch die Art der verwendeten Widgets bedingt sein. Die im Folgenden vorgestellten Elemente weisen Besonderheiten auf, die bei der Einordnung in Zeilen berücksichtigt werden müssen.

Mehrzeilige Textfelder

Mehrzeilige Textfelder haben die Rolle `text` und den Zustand `multi line`. Für die einzelnen Zeilen in diesem Widget gibt es keine eigenständigen Koordinaten und Größenangaben. Diese existieren nur für das gesamte Text-Element. Deshalb ist es sinnvoll, die Größe des Widgets auf die Anzahl der angezeigten Zeilen aufzuteilen und somit aus einem Blatt-Knoten mehrere Zeilen zu gewinnen.



The image shows a form titled "Home" with several input fields. The "Address" field is a multi-line text area containing "10 Glebe Street", "Broonsville", and "Dundee". The "City" field is a single-line text box. The "Zip/Postal code" field is a single-line text box containing "123456". The "State/Province" field is a single-line text box. The "P.O. box" field is a single-line text box. The "Country" field is a single-line text box.

Abbildung 3.3: Beispiel eines Inhaltsbereiches mit Textfeldern

Abbildung 3.3 zeigt beispielhaft ein Formular mit einem mehrzeiligen Textfeld. Das Feld entspricht genau einem Knoten des AT-SPI Baumes und hat damit auch nur eine Größenangabe. Da das mehrzeilige Textfeld allerdings drei Zeilen enthält, wird die Größe auf alle drei Zeilen gleichmäßig aufgeteilt. Dies erlaubt eine ausgewogene Darstellung der Inhalte im Flächenmodus.

Tabellen

In Tabellen können mehrere Fälle auftreten, die bei der Darstellung auf dem Brailledisplay zu berücksichtigen sind.

Start	Ziel	Abfahrt	Ankunft	Dauer
Berlin	Dresden	13:30	15:55	2h 25 min
Stralsund	Rostock	08:13	09:06	53 min
Greifswald	Stralsund	11:15	11:39	24 min
Sassnitz	Greifswald	17:23	18:59	1h 36 min

Tabelle 3.1: Kleine Beispiel-Tabelle

In Tabelle 3.1 ist eine kleine einfache Tabelle abgebildet. Alle Zellen enthalten kurze Einträge und die Zeilen sind klar zu erkennen.

Autor	Titel	Genre	Preis
J.K. Rowling	Harry Potter und der Stein der Weisen	Kinderbuch	11,95 Euro
Ian Irvine	Die Geomantin	Fantasy	14,00 Euro
Bram Stoker	Dracula	Horror	9,00 Euro

Tabelle 3.2: Beispiel-Tabelle mit mehreren Zeilen in einer Zeile

Anders liegt der Fall bei Tabelle 3.2. In diesem Beispiel gibt es einen Zeilenumbruch innerhalb einer Tabellenzelle, so dass eine Zelle aus zwei Zeilen besteht. Abhängig von der Anwendung kann der Zeilenumbruch in der Tabellenzelle erkannt werden oder auch unentdeckt bleiben. Im OpenOffice.org Writer, der Standard-Textverarbeitung unter GNOME, kann diese Information aus der AT-SPI Schnittstelle extrahiert werden und die Tabellenzelle als zwei Zeilen auf dem Brailledisplay abgebildet werden. Im Gegensatz dazu, bietet OpenOffice.org Calculator, die Standard-Tabellenkalkulation, diese Information nicht. Hier wird die Tabellenzelle, die eigentlich einen Zeilenumbruch enthält, als eine Zeile dargestellt.

Einen Schritt weiter geht das Beispiel in Tabelle 3.3. Hier werden mehrere Zellen vertikal miteinander verbunden, so dass eine Tabellenzelle mehrere andere Zellen umspannt. Dies kann in der Tabellenkalkulation nur auf eine rein visuelle Art geschehen und ist damit nicht für die Accessibility

Autor	Titel	Genre	Preis
J.K. Rowling	Harry Potter und der Stein der Weisen	Kinderbuch	11,95 Euro
	Harry Potter und die Kammer des Schreckens	Kinderbuch	11,95 Euro
	Harry Potter und der Gefangene von Askaban	Kinderbuch	11,95 Euro
Ian Irvine	Die Geomantin	Fantasy	14,00 Euro
Bram Stoker	Dracula	Horror	9,00 Euro

Tabelle 3.3: Beispiel-Tabelle vertikal verbundener Zellen

Schnittstelle sichtbar. In der Textverarbeitung hingegen kann dieser Fall ohne Schwierigkeiten abgebildet werden. Dieses Problem kann nach den im nächsten Abschnitt aufgestellten Regeln zur Konfliktbeseitigung gelöst werden. Zusätzlich muss dem Anwender allerdings noch die Information gegeben werden, dass hier mehrere Zellen miteinander verbunden sind. Wie das konkret auf dem Brailledisplay aussehen kann, wird in Abschnitt 3.4 gezeigt.

3.2 Anordnung von Widgets – Probleme und Lösungen

3.2.1 Konflikte

Nachdem die logischen Einheiten im vorherigen Abschnitt festgelegt worden sind, sollen nun Untersuchungen zur Anordnung von beliebigen Widgets im Inhaltsbereiche erfolgen.

Am einfachsten ist es sicherlich, wenn alle Elemente die gleiche Höhe haben und in einem Gitter angeordnet sind (Abbildung 3.4). In solch einem Fall ist die Einteilung in Zeilen simpel.

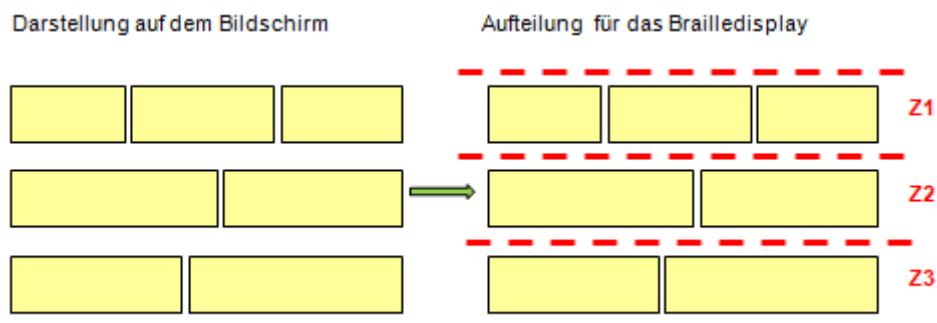


Abbildung 3.4: Anordnung von Widgets mit eindeutiger Zeilenzuordnung

Wie sieht es aber aus, wenn die Widgets unterschiedlich hoch sind? (Abbildung 3.5) Auch in diesem Fall ist die Einteilung in Zeilen eindeutig. Die kleineren Widgets passen sich den Größeren an.

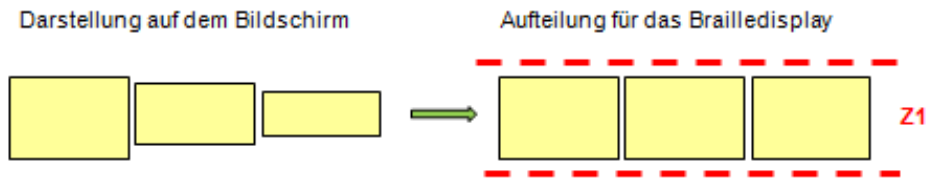


Abbildung 3.5: Anordnung von Widgets mit unterschiedlichen Höhen

Die nachfolgenden Abbildungen 3.6 und Abbildung 3.7 zeigen Beispiele, bei dem die Einteilung in Zeilen nicht mehr eindeutig ist. Eine Möglichkeit wäre, die Widgets so zu verschieben, dass sich eindeutige Zeilen herauskristallisieren. Aber wie sollte solch eine Verschiebung dann aussehen?

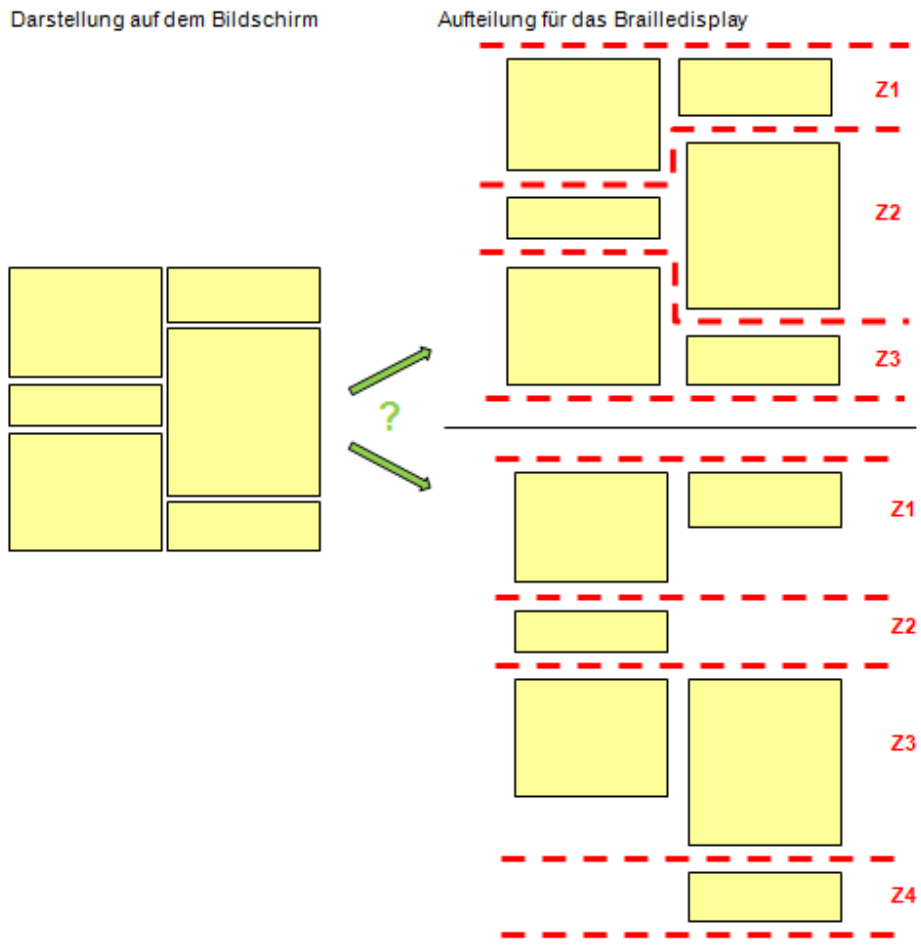


Abbildung 3.6: Anordnung von Widgets über mehrere Zeilen – Beispiel 1

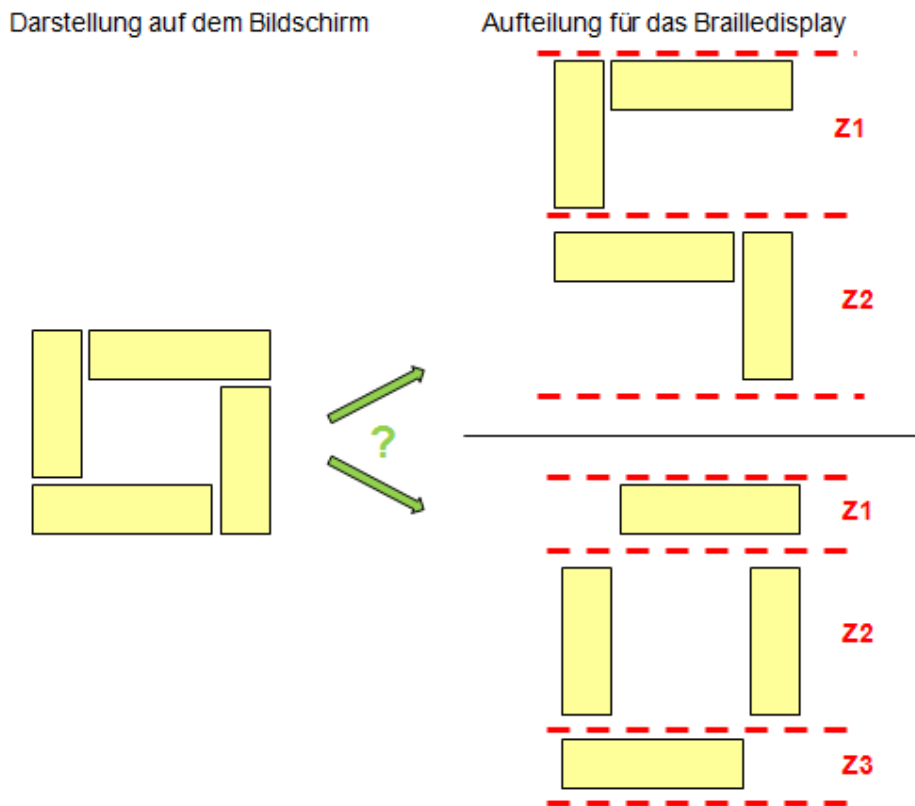


Abbildung 3.7: Anordnung von Widgets über mehrere Zeilen – Beispiel 2

3.2.2 Regeln

Zur Lösung der Konflikte, wie sie im vorherigen Abschnitt beschrieben sind, wurden Regeln erarbeitet. Diese beschreiben eine Strategie, um die grafischen Elemente in Zeilen einzuordnen.

- ✓ **Regel 1:** Jedes Widget wird genau einer Zeile zugeordnet.
- ✓ **Regel 2:** Jede obere Kante eines Widgets stellt den Abschluss der einen Zeile und den Beginn einer neuen Zeile dar.
- ✓ **Regel 3:** Jedes Widget bekommt eine Gewichtung pro Zeile, abhängig davon, wie viel Prozent des Widgets sich in der entsprechenden Zeile befindet. Diese Gewichtung ist ein Wert zwischen 0 und 100.
- ✓ **Regel 4:** Jedes Widget wird der Zeile zugeordnet, bei der es die höchste Gewichtung hat.

- ✓ **Regel 5:** Bei gleicher Gewichtung eines Widgets in mehreren Zeilen erhält die höher gelegene Zeile den Vorrang.
- ✓ **Regel 6:** Ist ein Widget das einzige Element in einer Zeile, kommt aber noch in anderen Zeilen vor, so wird die Gewichtung in der Zeile ohne weiterer Elemente auf 0 gesetzt.
- ✓ **Regel 7:** Entstehen leere Zeilen werden diese zur darüber liegenden Zeile hinzugefügt. Gibt es keine darüber liegende Zeile, so wird sie mit der darunter liegenden vereint.

Die erste Regel ist für ein einfaches und verständliches Arbeiten notwendig. Würden Widgets mehrmals angezeigt werden, weil sie mehreren Zeilen zugeordnet sind, dann fiel es dem Anwender schwer zu entscheiden, ob das angezeigte Widget nur einmal oder mehrmals auf dem Bildschirm zu sehen ist. Die zweite Regel erfasst alle potentiellen Zeilen. Sobald diese festliegen, kann die Verteilung der Widgets beurteilt werden. Dies geschieht am besten durch die Zuordnung von Gewichtungen (Regel 3). Um abzuschätzen, zu welcher Zeile das Widget am sinnvollsten zugeordnet werden kann, wird eine prozentuale Verteilung für die Gewichtung genutzt. Das grafische Element sollte in der Zeile positioniert werden, in der es zum überwiegenden Teil zu sehen ist (Regel 4). Haben mehrere Zeilen einen gleichen Anteil an dem Widget, so sollte die höher gelegene den Vorrang erhalten (Regel 5). Dies entspricht unserem visuellem Empfinden und ist deshalb auch für die Übertragung auf die Braillezeile geeignet.

Nach dem Anwenden der Regeln 1 bis 5 können Situationen auftreten, bei denen Widgets einer Zeile zugeordnet wurden, in der kein anderes Element vorkommt. Treten diese Widgets noch in anderen Zeilen auf, macht es keinen Sinn, sie gesondert anzuzeigen. In dem Fall kommt Regel 6 zur Anwendung und die Maximal-Gewichtung wird erneut für das Widget bestimmt. Weiterhin können leere Zeilen entstehen. Da diese keinen Zweck erfüllen werden sie ebenfalls zu anderen Zeilen hinzugefügt (Regel 7).

Werden diese Regeln konsequent auf eine beliebige Anordnung von Widgets angewendet, so ist das Ergebnis immer eine eindeutige Einordnung der grafischen Elemente in Zeilen.

3.2.3 Anwenden der Regeln auf die Konflikte

Wendet man diese Regeln auf die in Abbildung 3.6 und Abbildung 3.7 dargestellten Gruppierungen von Widgets an, so ergeben sich eindeutige Aufteilungen. Dies soll anhand des Beispiels in Abbildung 3.6 demonstriert werden.

Abbildung 3.8(a) zeigt die Ausgangssituation. Zunächst einmal werden die Grenzen für die Zeilen festgelegt. Hierfür werden alle Oberkanten der

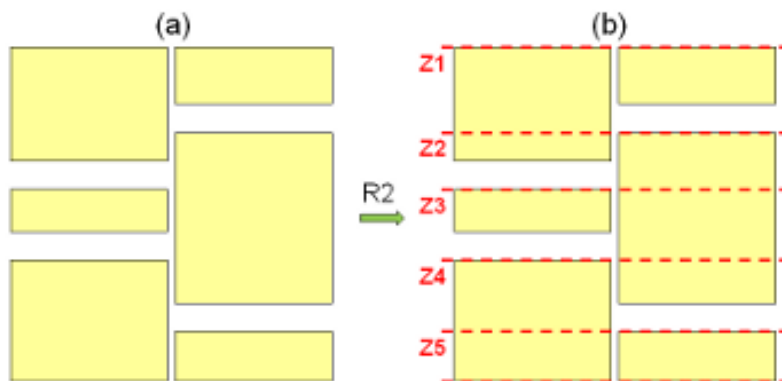


Abbildung 3.8: Auflösung von Konflikten – Teil 1

grafischen Elemente genommen und die Unterkante der Einheit. Doppelte Werte werden entfernt. Eine Zeile nimmt den Raum zwischen zwei aufeinanderfolgenden Werten ein. Das Ende einer Zeile bedeutet der Beginn einer neuen. Bei dieser Vorgehensweise wird der Freiraum zwischen zwei Widgets der oberen Zeile zugeordnet. Ist der Freiraum größer als durchschnittlich, so verleiht dies der oberen Zeile mehr Gewichtung. Das entspricht unserem visuellen Empfinden und wird somit auch für den blinden Anwender ausgenutzt. Abbildung 3.8(b) zeigt die Einteilung in Zeilen für das Beispiel.

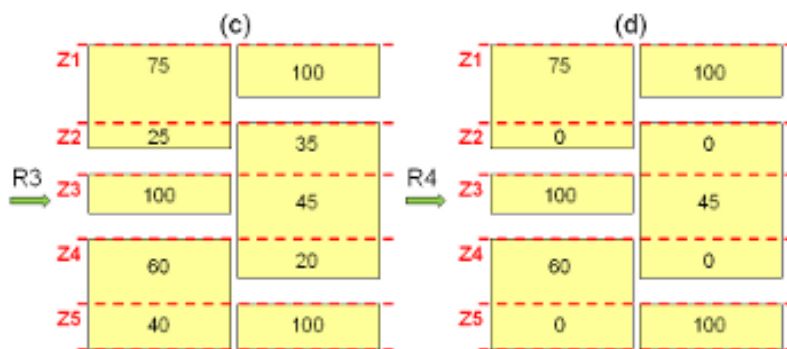


Abbildung 3.9: Auflösung von Konflikten – Teil 2

Nachdem die Zeilen festgelegt sind, offenbart sich, dass einige Widgets mehreren Zeilen zugeordnet wurden. Dies steht im Widerspruch zu Regel 1. Um diese Konflikte aufzulösen, werden im nächsten Schritt die Gewichtungen durch Anwendung von Regel 3 verteilt. Für das Beispiel können diese der Abbildung 3.9(c) entnommen werden. Durch die Vollstreckung von Regel 4 werden einige dieser Konflikte beseitigt (Abbildung 3.9(d)).

Nach diesem Schritt steht die Anordnung in Zeile 4 im Widerspruch zu Regel 6 und muss daher korrigiert werden. Die Gewichtung des einzelnen

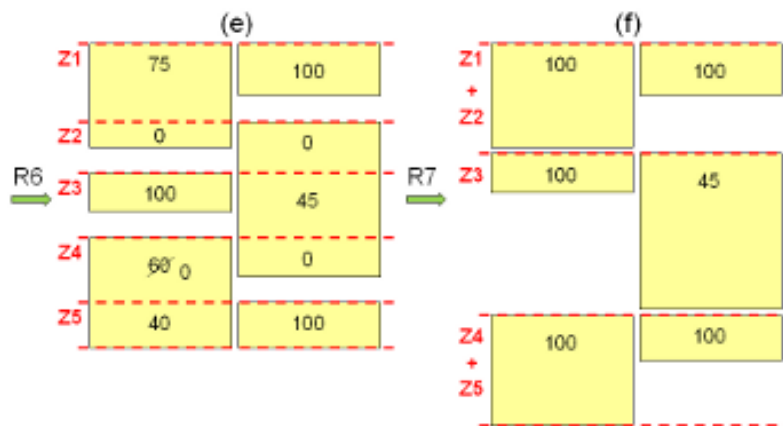


Abbildung 3.10: Auflösung von Konflikten – Teil 3

Widgets in Zeile 4 wird auf 0 gesetzt und die in Zeile 5 wiederhergestellt. Dadurch wird das Element Zeile 5 zugeordnet. Abbildung 3.10(e) zeigt, dass anschließend für dieses Beispiel alle Konflikte behoben sind. Leere Zeilen werden in einem letzten Schritt zu benachbarten Zeilen hinzugefügt. Die Anordnung der Elemente wird nicht mehr verändert, auch wenn sich die Gewichte pro Zeile anpassen. Eine übersichtliche Darstellung der endgültigen Inhalte der Zeilen zeigt Abbildung 3.10(f).

3.3 Baumtransformation

In diesem Abschnitt wird der Algorithmus erstellt, der flächendeckend die Bildschirminhalte zusammen sucht. Da die Ausgabe per Brailledisplay erfolgen soll, werden die dargestellten grafischen Elemente zunächst einmal in Zeilen eingeteilt. Je nach Größe des Displays können dann eine oder mehrere Zeilen per Braille angezeigt werden. Da eine gängige große Braillezeile maximal 80 Zeichen darstellen kann, ist die Ausgabe einer Zeile über die komplette Bildschirmbreite nicht handhabbar. Die Übersichtlichkeit und damit das Verständnis würden verloren gehen. Aus diesem Grund kommen die in der Analyse der grafischen Oberfläche (siehe Kapitel 3.1) erkannten logischen Einheiten hier zur Anwendung. Die Einteilung in Zeilen wird pro Einheit vorgenommen und nicht für den gesamten Bildschirm. Die Einheiten wiederum werden ebenfalls in eine Reihenfolge gebracht. Diese muss nicht zwangsläufig einer zeilenweisen Bildschirmanordnung entsprechen. Vielmehr ist bei dieser Sortierung die Arbeitsweise eines blinden Anwenders entscheidend. Dieser nutzt vor allem die Tastatur und die Braillezeile, um den Bildschirm zu erkunden. Damit ist eine Reihenfolge, in der die grafischen Elemente angesteuert werden, vorgegeben. Im Allgemeinen bedeutet dies, dass von zwei Einheiten, die auf gleicher Höhe nebeneinander angeordnet sind, erst

die linke und dann die rechte durchlaufen wird. Auf Brailledisplays, die die Darstellung mehrerer Zeilen erlauben kann es vorkommen, dass Platz für die Anzeige von mehr als einer logischen Einheit ist. Durch die Sortierung der Einheiten kann eindeutig festgestellt werden, welche Zeilen untereinander auf dem Brailledisplay präsentiert werden müssen.

Die Analyse der Accessibility Schnittstelle in Abschnitt 2.3 hat gezeigt, dass die Elemente auf dem grafischen Desktop intern in einer Baumstruktur präsentiert werden. Es muss also eine Transformation dieses Baumes in einen Graphen erfolgen, der die Elemente zeilenweise präsentiert.

Als Ausgangssituation für die Abarbeitung des Algorithmus wird eine Aktion des blinden Anwenders angenommen. Das dadurch ausgelöste Ereignis liefert dem Screenreader ein Objekt, welches das gerade manipulierte Widget im AT-SPI Baum repräsentiert. Dieses Objekt ist ein beliebiger Knoten in dem Accessibility Baum, der sich meist in der Nähe eines Blattes befindet oder selber ein Blatt ist.

Algorithm 1 Baumtransformation – Überblick

```
1: function ERSTELLE_MATRIX(Einheit)
2:   finde alle Widgets in der aktiven Einheit
3:   sortiere die Widgets in Zeilen ein
4:   löse auftretende Konflikte auf
5:   return Matrix mit in Zeilen sortierten Widgets
6: end function

7: procedure BAUMTRANSFORMATION(Widget)
8:   bestimme die aktive Einheit
9:   Matrix = ERSTELLE_MATRIX(aktive Einheit)
10:  while Anzahl der ermittelten Zeilen zu klein do
11:    if alle logischen Einheiten sind nicht bekannt then
12:      bestimme alle logischen Einheiten des Fensters
13:      sortiere alle logischen Einheiten
14:    end if
15:    hole die benachbarte Einheit zur aktiven Einheit
16:    Matrix2 = ERSTELLE_MATRIX(benachbarte Einheit)
17:    Matrix = Matrix + Matrix2
18:  end while
19: end procedure
```

Der Algorithmus 1 gibt einen Überblick über die auszuführenden Aktionen bei der Baumtransformation. Der Einstiegspunkt ist hierbei die Prozedur `Baumtransformation`, die beim Aufruf den aktiven Knoten des Baumes übergeben bekommt. Da dieser Knoten gleichzeitig die Repräsentation eines grafischen Elementes auf dem Desktop ist, wird im weiteren kein Unterschied

zwischen einem Widget und einem Knoten gemacht.

Zunächst einmal muss herausgefunden werden, wo sich das aktive Widget befindet. Dies geschieht durch die Bestimmung der logischen Einheit, in der das Widget platziert ist. Diese wird als aktive Einheit bezeichnet. Sobald sie ermittelt wurde, können alle in ihr enthaltenen Widgets aus dem AT-SPI Baum gesucht und in Zeilen einsortiert werden. Dabei können, wie im Abschnitt 3.2 gezeigt, verschiedene Konflikte auftreten. Diese müssen dann nach den bereits erstellten Regeln aufgelöst werden. Das Ergebnis der Prozedur wird eine Matrix sein, die alle ermittelten Widgets eindeutig in Bildschirmzeilen anordnet.

Bei zukünftigen Brailledisplays wird es möglich sein, mehrere Zeilen mit Text darzustellen. Bei solch einem Ausgabegerät kann es sein, dass die aktive Einheit weniger Zeilen enthält, als auf dem Brailledisplay angezeigt werden können. In dem Fall ist es notwendig, weitere logische Einheiten zu bestimmen und deren Inhalte zu analysieren. Eine angemessene Sortierung über alle logischen Einheiten bestimmt, welche miteinander benachbart sind und gleichzeitig auf solch einem Brailledisplay angezeigt werden können.

3.3.1 Zerlegung in Teilbäume

Die Einteilung der grafischen Oberfläche, wie in Abschnitt 3.1 festgelegt, liefert die Voraussetzung, um den Baum in kleinere Teilbäume zu zerlegen. Anstatt den gesamten Baum zu durchsuchen, müssen die Zeilen nur aus einem Zweig extrahiert werden.

Die Bestimmung der logischen Einheiten ist sehr stark vom Aufbau des AT-SPI Baumes abhängig. Eine detaillierte Analyse der Repräsentation von mehreren gängigen Anwendungen unter Linux liefert die Ausgangsbasis für diesen Teil des Algorithmus.

Folgende Anwendungen wurden zu diesem Zweck untersucht:

- GNOME Desktop
- Dateibrowser: Nautilus
- Texteditor: GEdit
- Textverarbeitung: OpenOffice.org Writer
- Tabellenkalkulation: OpenOffice.org Calculator
- Mail-Client: Evolution
- Taschenrechner
- verschiedene Einstelldialoge

In dieser heuristischen Analyse wurde versucht, gewisse Regelmäßigkeiten in der Anordnung von Knoten zu erkennen. Ziel war es, die Knoten im Baum identifizieren zu können, die jeweils eine Einheit repräsentieren.

Die wichtigste Erkenntnis dieser Analyse ist die Unterscheidung der Knoten in logische Einheiten und Widgets anhand ihrer Rollen. Diese Einteilung ist Tabelle 3.4 zu entnehmen.

Rollen logischer Einheiten	Rollen von Widgets
application, dialog, alert, frame, file chooser, window, popup menu, document frame, html container, menu bar, tool bar, statusbar, page tab list, table, list, tree, tree table, layered pane, panel, filler	accelerator label, label, push button, radio button, toggle button, text, editbar, entry, password text, terminal, paragraph, header, footer, caption, heading, check box, combo box, dial, progress bar, slider, spin button, menu, menu item, check menu item, radio menu item, tearoff menu item, list item, page tab, icon, table cell

Tabelle 3.4: Rollenverteilung für logische Einheiten und Widgets

Eine besondere Rolle haben Panel- und Filler-Knoten. Diese kommen häufig im AT-SPI Baum vor und nur in ganz bestimmten Fällen repräsentieren sie eine logische Einheit. Zusätzlich können Split-Pane-Knoten bei der Identifizierung von Einheiten behilflich sein. Es gibt noch eine Reihe weiterer Rollen, die allerdings für diese Arbeit keine Bedeutung haben.

Neben der Rollen-Bezeichnung ist die Sichtbarkeit eines Knotens eine weitere wichtige Eigenschaft, die für die Bestimmung der logischen Einheiten notwendig ist. Nur Widgets, die auch tatsächlich auf dem Bildschirm angezeigt werden, sind von Interesse.

Bei Betrachtung der Anordnung sichtbarer Knoten mit bestimmten Rollen können nun tatsächlich Gesetzmäßigkeiten festgestellt werden. Abbildung 3.11 zeigt beispielhaft einige gängige Anordnungen der Elemente im AT-SPI Baum. Detailliertere Aussagen zu dieser Analyse sind im Anhang A zu finden.



Abbildung 3.11: Anordnung von Knoten im AT-SPI Baum

Bestimmung der aktiven Einheit

Auf der Grundlage dieser Informationen kann nun der Programmteil zur Bestimmung der aktiven logischen Einheit erstellt werden. Ausgehend von dem aktiven Widget befindet sich der Fokus an einer beliebigen Position im Baum.

Zur übersichtlicheren Handhabung wird eine Aufteilung der Rollen wie im Teil-Algorithmus 2 vorgenommen.

Algorithm 2 Rollenaufteilung

- 1: ROLES_WINDOW = application, window, frame, dialog, alert, file chooser
 - 2: ROLES_DOC = document frame, html container
 - 3: ROLES_BAR = menu bar, tool bar, statusbar
 - 4: ROLES_NAV = table, list, tree, tree table
 - 5: ROLES_CLUSTER = ROLES_*, page tab list, layered pane
 - 6: HELP_ROLES = split pane
-

Ziel des Moduls Algorithm 3 ist die Ermittlung der logischen Einheit, welche das aktive Widget enthält. Dafür wird zunächst ein Vorfahre zum aktiven Widget gesucht, der eine der in den ROLES_CLUSTER enthaltenen Rollen besitzt. Solch ein Vorfahre wird immer gefunden, da alle Elemente des AT-SPI Baumes, die sich auf der Ebene unterhalb der Wurzel befinden, die Rolle `Application` haben und diese eine der Rollen in der Liste ist. Wahrscheinlich wird dieses Element mit der Funktion `FINDE_VORFAHRE_MIT_CLUSTER_ROLLE` aber nie erreicht, da auf der Ebene unterhalb der `Application`-Knoten die Fenster-Knoten liegen, die eine der Rollen `frame`, `dialog`, `alert` oder `file chooser` haben. Es konnte

Algorithm 3 Bestimmung der aktiven Einheit

```
1: function FINDE_VORFAHRE_MIT_CLUSTER_ROLLE(Knoten)
2:   hole Vater vom Knoten
3:   if Vater hat Rolle aus ROLES_CLUSTER or HELP_ROLES then
4:     return Vater
5:   else
6:     return FINDE_VORFAHRE_MIT_CLUSTER_ROLLE(Vater)
7:   end if
8: end function

9: function BESTIMME_AKTIVE_EINHEIT(Widget)
10:  Vorfahre = FINDE_VORFAHRE_MIT_CLUSTER_ROLLE(Widget)
11:  if Vorfahre hat Rolle aus ROLES_NAV then
12:    if Vorfahre liegt nicht am linken Rand des Fensters then
13:      return BESTIMME_AKTIVE_EINHEIT(Vorfahre)
14:    end if
15:  else if Vorfahre hat Rolle aus ROLES_WINDOW or
HELP_ROLES then
16:    bestimme alle logischen Einheiten des Fensters
17:    sortiere alle logischen Einheiten
18:    bestimme in den sortierten Einheiten die aktive Einheit
19:  else if Vorfahre hat die Rolle statusbar then
20:    hole Vater vom Vorfahre
21:    if Vater hat Rolle statusbar then
22:      aktive Einheit = Vater
23:    end if
24:  else
25:    aktive Einheit = Vorfahre
26:  end if
27:  return aktive Einheit und eventuell alle logischen sortierten Einheiten
28: end function
```

nicht eindeutig geklärt werden, ob es auf der Ebene mit den Fenster-Knoten noch Elemente mit anderen Rollen geben kann. Aus diesem Grund wurde die Rolle **Application** in die Liste mit aufgenommen.

Sobald der entsprechende Vorfahre-Knoten bestimmt wurde, kann ausgewertet werden, ob er tatsächlich eine logische Einheit repräsentiert. Elemente mit einer Rolle aus **ROLES_NAV** können nicht nur Einheiten, sondern auch Container von Widgets sein. Sie stellen genau dann eine logische Einheit dar, wenn sie am linken Rand des Fensters platziert sind. Hat der Vorfahre eine Rolle aus **ROLES_WINDOW** oder **ROLES_HELP**, so ist davon auszugehen, dass die richtige logische Einheit nicht gefunden wurde. In diesem Fall werden alle logischen Einheiten bestimmt, dann sortiert und die entsprechend aktive Einheit herausgesucht. Ein Sonderfall tritt auf, wenn der gefundene Knoten die Rolle **Statusbar** hat. Solch ein Knoten kann einen Vater-Knoten mit der gleichen Rolle besitzen. Dann ist der Vater-Knoten die Einheit und nicht der bis dahin gefundene Vorfahre. In allen anderen Fällen wurde mit dem Vorfahren die aktive logische Einheit zum Widget gefunden.

Bestimmung aller Einheiten

Algorithm 4 Bestimmung aller logischen Einheiten – Teil 1

```
1: function FINDE_ALLE_EINHEITEN(Fenster, Knoten-Liste)
2:   erstelle leere Liste: Einheiten
3:   finde sichtbaren Nachfahren mit Rolle aus ROLES_CLUSTER oder
   HELP_ROLES, überspringe bei der Suche alle Knoten aus Knoten-Liste
4:   if solch ein Nachfahre existiert then
5:     if Vorfahre hat Rolle aus ROLES_NAV then
6:       if Vorfahre liegt nicht am linken Rand des Fensters then
7:         füge Vorfahre zur Knoten-Liste hinzu
8:         return FINDE_ALLE_EINHEITEN(Fenster, Knoten-Liste)
9:       end if
10:    end if
11:    if Rolle des Nachfahren ist nicht aus HELP_ROLES then
12:      füge Nachfahren zu Einheiten hinzu
13:    end if
14:    e = ANALYSIERE_GESCHWISTER(Nachfahre)
15:    hänge e an Einheiten an
16:  else
17:    prüfe Fall 5
18:  end if
19:  return Einheiten
20: end function
```

Beim Bestimmen der aktiven Einheit wird der Baum von unten nach oben, also vom Blatt zur Wurzel, rekursiv durchsucht. Genau anders herum ist das Vorgehen, wenn alle logischen Einheiten bestimmt werden (Algorithmus 4 und 5). Dann wird vom Fenster-Knoten ausgegangen und die Nachfahren werden analysiert. Hierbei ist es wichtig, dass möglichst wenig Knoten betrachtet werden, um zu einem Ergebnis zu gelangen. Das Ziel ist, den AT-SPI Baum, durch die Einteilung in logische Einheiten, in Teilbäume zu zerlegen, die dann bei Bedarf einzeln durchsucht werden können. Um eine performante Einteilung in diese Teilbäume zu erreichen, wird die Analyse ebenenweise durchgeführt. Im Allgemeinen befinden sich alle logischen Einheiten auf den oberen Ebenen des Baumes.

Aus der detaillierten Analyse des AT-SPI Baumes (siehe Anhang A) konnten 5 Fälle abgeleitet werden, mit deren Hilfe der Aufbau der verschiedenen Teilbäume bestimmt werden kann.

- **Fall 1:** Der gefundene Knoten hat keine Geschwister oder nur solche, die die gleiche Rolle haben. In dem Fall müssen alle Geschwister des ersten Vorgängers vom Vater-Knoten mit mehr als einem sichtbaren Nachfolger untersucht werden.
- **Fall 2:** Stellt anhand der Rollen fest, ob der gefundene Knoten eine logische Einheit ist.
- **Fall 3:** Wenn Sibling-Knoten einer Einheit Widgets sind, ist das Parent-Element eine logische Einheit.
- **Fall 4:** Ein `Split Pane` wurde gefunden. Dieser teilt den Bereich meist in zwei Teile, von denen einer Seitenreiter enthält.
- **Fall 5:** Die logische Einheit befindet sich in einem Container Element, wie zum Beispiel einem `Panel`. Solche Container Elemente kommen sehr häufig vor. Sie sind nur in einigen wenigen Fällen interessant.

In jedem der Fälle werden gefundene logische Einheiten an die Liste der Einheiten angehängt. Für mehr Details zu den einzelnen Fällen kann der Quellcode zu dieser Arbeit herangezogen werden.

Algorithm 5 Bestimmung aller logischen Einheiten – Teil 2

```
21: function STANDARD_ANALYSE(Einheiten, Geschwister)
22:   prüfe Fall 2
23:   prüfe Fall 3
24:   for s in Geschwister do
25:     if Rolle von s ist split pane then
26:       prüfe Fall 4
27:     else if Rolle von s ist panel then
28:       prüfe Fall 5
29:     else if Rolle von s is filler then
30:       if s hat Geschwister mit Rolle page tab list then
31:         füge s zu Einheiten hinzu
32:       else
33:         prüfe Fall 5
34:       end if
35:     end if
36:   end for
37:   return Einheiten
38: end function

39: function ANALYSIERE_GESCHWISTER(Knoten)
40:   erstelle leere Liste: Einheiten
41:   hole alle Geschwister zum Knoten
42:   if Knoten hat Rolle aus ROLES_NAV oder page tab list then
43:     prüfe Fall 3
44:     for s in Geschwister do
45:       if Rolle von s ist filler then
46:         füge s zu Einheiten hinzu
47:       else if Rolle von s ist panel then
48:         prüfe Fall 5
49:       end if
50:     end for
51:   else if Knoten hat Rolle split pane then
52:     prüfe Fall 4
53:     Einheiten = STANDARD_ANALYSE
54:   else
55:     prüfe Fall 1
56:     Einheiten = STANDARD_ANALYSE
57:   end if
58:   return Einheiten
59: end function
```

3.3.2 Sortierung der Einheiten

Nachdem nun alle relevanten Teilbäume das heißt alle Einheiten in einem Fenster gefunden wurden, sollen diese im nächsten Schritt in eine Reihenfolge gebracht werden (Algorithmus 6).

Dieser Programmteil arbeitet rekursiv, bis alle Einheiten sortiert sind. Als Ausgangswerte wird eine Liste mit allen gefunden Einheiten, die Positionierung des Fensters auf dem Bildschirm und die Größe des Fensters benötigt. Zunächst werden alle Elemente aus der Liste gesucht, die am linken Fensterrand positioniert sind. Anschließend erfolgt eine Sortierung dieser Widgets nach der Positionierung ihrer oberen Kanten. Da davon auszugehen ist, dass sich die Elemente nicht überlagern, ist das Ergebnis eine sortierte Liste mit Elementen, die untereinander angeordnet sind. Das Element, welches am oberen Fensterrand beginnt ist das Erste in der Liste. Dies entspricht der Navigationsreihenfolge beim Arbeiten in einem Fenster. Elemente, die über die untere Kante des Fensters hinausgehen, werden aus der Liste wieder entfernt. Dieser Fall wird beim ersten Durchlauf der Methode nicht eintreten. Allerdings kann er durch den rekursiven Charakter der Funktion in späteren Durchläufen vorkommen. Die Abbildung 3.12 gibt wieder, bei welcher Art von Anordnung sich das Problem zeigt.

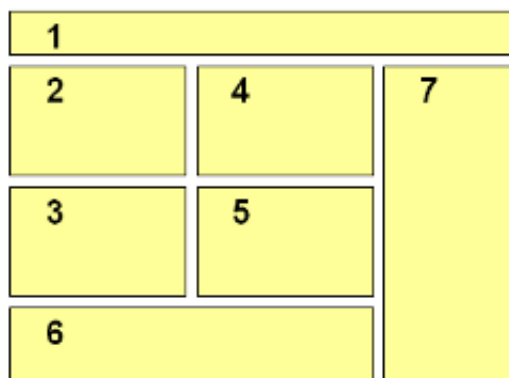


Abbildung 3.12: Beispiel einer Anordnung für Einheiten

Neben Einheiten, die nicht die ganze Fensterbreite ausfüllen, ist noch Platz für die Anzeige weiterer Einheiten. Diese müssen nun ebenfalls in die Liste einsortiert werden. Dafür wird zunächst einmal das erste Element in der Liste gesucht, welches nicht die ganze Fensterbreite einnimmt. Dann werden alle nachfolgenden Elemente mit gleicher Breite ermittelt. Die zusammenhängende Höhe dieser Einheiten entspricht der Höhe des als nächstes zu untersuchenden Raumes rechts daneben. Die Breite des Raumes entspricht der Fensterbreite minus der linken Kante einer der Einheiten. Damit ist die Größe für den nächsten Durchlauf der rekursiven Funktion gegeben. Die

Algorithm 6 Sortierung aller Einheiten - Teil 1

```
1: function TEILE_EINHEITEN(Einheiten)
2:   erstelle leere Listen: linkeE, andereE
3:   for e in Einheiten do
4:     if liegt e am linken Fensterrand then
5:       füge e zu linkeE hinzu
6:     else
7:       füge e zu andereE hinzu
8:     end if
9:   end for
10:  return linkeE, andereE
11: end function

12: function SORTIERE_EINHEITEN(Einheiten, F-Position, F-Größe)
13:  linkeE, andereE = TEILE_EINHEITEN(Einheiten)
14:  sortiere die Elemente in linkeE nach ihrer oberen Kante
15:  erstelle leere Liste: sort
16:  for e in linkeE do
17:    if die untere Kante von e ist nicht im Fenster then
18:      füge e zu andereE hinzu
19:    else if e ist nicht so breit, wie das Fenster then
20:      höhe = höhe von e
21:      breite = Breite des Fensters – Breite von e
22:      hänge e an sort an
23:    for alle n in linkeE, die nach e angeordnet sind do
24:      if n hat die gleiche Breite wie e then
25:        höhe = höhe + Freiraum über n + höhe von n
26:        hänge n an sort an
27:      else
28:        break
29:      end if
30:    end for
31:    erstelle leere Liste: nachbarE
32:    for a in andereE do
33:      if befindet sich a rechts neben e und den nachfolgenden
34:      Knoten gleicher Breite then
35:        füge a zu nachbarE hinzu
36:        entferne e aus andereE
37:      end if
38:    end for
```

Algorithm 7 Sortierung aller Einheiten – Teil 2

```
38:         s, a = SORTIERE_EINHEITEN(nachbarE, rechte obere Ecke
    von e, (höhe, breite))
39:         hänge s an sort an
40:         füge a zu andereE hinzu
41:     else
42:         hänge e an sort an
43:     end if
44: end for
45: return sort, andereE
46: end function
```

Positionierung ist die rechte obere Ecke der ersten Einheit, die nicht über das gesamte Fenster ging. Als letztes Argument für den rekursiven Aufruf der Funktion fehlt noch die Liste mit den Elementen, die als nächstes betrachtet werden sollen. Zu dieser Liste gehören alle Elemente, die sich in diesem neu ermittelten Raum befinden, auch die, die diesen Raum nur anschneiden. Diese werden nun nach der hier beschriebenen Methode sortiert. Die dabei entstehende Liste mit sortierten Elementen wird nach dem letzten Element mit der entsprechenden Breite eingefügt. Dann wird überprüft, ob von den nachfolgenden Elementen ebenfalls einige nicht über die gesamte Fensterbreite gehen. Wenn die Liste der links angeordneten Einheiten abgearbeitet ist, können noch einzelne Einheiten nicht in die neue Liste einsortiert sein. Diese befinden sich komplett innerhalb einer anderen Einheit. In einem letzten Schritt müssen diese abhängig von der Positionierung innerhalb der anderen Einheit vor oder hinter dieser einsortiert werden.

3.3.3 Bestimmung der aktiven Einheit in einer sortierten Liste von Einheiten

Nachdem nun alle logischen Einheiten ermittelt und sortiert wurden, soll als nächstes die Einheit bestimmt werden, die das aktive Widget enthält (Algorithmus 8).

Die Ermittlung der aktiven Einheit geschieht, indem alle Einheiten nacheinander darauf getestet werden, ob sie das aktive Widget enthalten. Sobald die entsprechende Einheit gefunden wurde, kann mit der Suche aufgehört werden.

3.3.4 Ermittlung aller Widgets in der Einheit

Widgets in einer Einheit sind alle sichtbaren Knoten des Teilbaumes, die eine bestimmte Rolle haben (siehe Tabelle 3.4). Bereits bei der Ermittlung aller

Algorithm 8 Bestimmung der aktiven Einheit in einer Liste von sortierten Einheiten

```
1: function FINDE_AKTIVE_EINHEIT(Einheiten, Widget)
2:   for e in Einheiten do
3:     if die linke ober Ecke von Widget liegt innerhalb von e then
4:       return e
5:     end if
6:   end for
7:   return Fenster-Knoten
8: end function
```

Einheiten werden die Widgets von bestimmten Einheiten erfragt. Wenn dies der Fall ist, ist es sinnvoll, sich diese für den hier beschriebenen Programmteil zu merken. Für alle anderen Einheiten wird der Teilbaum, der die Einheit repräsentiert nach allen sichtbaren Widgets durchsucht.

Hat die Einheit die Rolle `page tab list`, so ist ein Zwischenschritt nötig, bevor der Teilbaum durchsucht werden kann. Alle Child-Knoten einer `page tab list` sind Widgets. Von diesen hat allerdings nur eines den Zustand `selected`. Dies ist der Knoten, dessen Teilbaum nach weiteren Widgets durchsucht werden kann.

Algorithm 9 Ermittlung aller Widgets einer Einheit

```
1: function FINDE_WIDGETS(Einheit)
2:   erstelle leere Liste: widgets
3:   if Widgets der Einheit sind bereits bekannt then
4:     return widgets
5:   else if Rolle der Einheit ist page tab list then
6:     hole alle Children zu dem Knoten der Einheit
7:     hänge Children an widgets an
8:     for child in Children do
9:       if child hat Zustand selected then
10:        Einheit = child
11:       break
12:     end if
13:   end for
14:   end if
15:   hole alle Nachfahren von Einheit, die auf dem Bildschirm sichtbar sind und eine bestimmte Rolle haben
16:   füge diese Nachfahren zu widgets hinzu
17:   return widgets
18: end function
```

Das Ergebnis dieses Moduls ist eine unsortierte Liste mit allen sichtbaren

grafischen Elementen des Teilbaumes.

3.3.5 Aufbau eines Gewichteten Graphen

Nachdem nun die Einheiten bekannt sind und die Widgets der Einheiten ermittelt wurden, können die Erkenntnisse aus Abschnitt 3.2 angewendet werden.

Bestimmung der Zeilen

Zunächst wird die Einheit in Zeilen aufgeteilt. Laut Regel 2 entspricht jede obere Kante eines Widgets einer Zeilengrenze. Zur Begrenzung der letzten Zeile wird die untere Kante der Einheit hinzugenommen (siehe Algorithmus 10). Sobald die Zeilen bekannt sind, müssen die Widgets diesen zugeordnet werden. Dies geschieht in Form eines Graphen. Zeilen-Knoten werden mit Widget-Knoten über eine Kante verbunden, wenn das Widget im Bereich der Zeile angeordnet ist. Dabei kann es, wie bereits im Abschnitt 3.2.1 gezeigt, geschehen, dass einige Widgets mehreren Zeilen zugeordnet werden. Deshalb wird nun Regel 3 eingesetzt und die Widgets erhalten für jede Zeile eine Gewichtung. Ein gewichteter Graph entsteht (Abbildung 3.13).

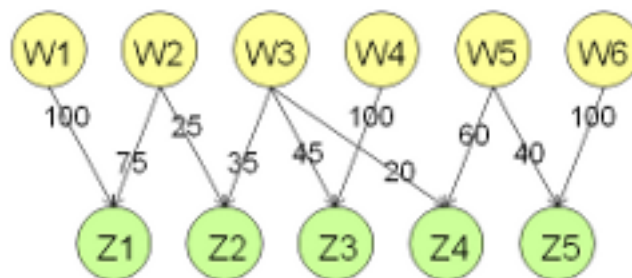


Abbildung 3.13: Gewichteter Graph

Bestimmung der Gewichtung (Clipping)

Als nächstes ist zu klären, wie die Gewichtung der einzelnen Kanten ermittelt werden kann. Dieser Sachverhalt kann auf ein Problem zurückgeführt werden, welches in der Informatik als Clipping bezeichnet wird.

„Unter Clipping versteht man das Abschneiden der Teile eines Bildes, die außerhalb eines vorher vereinbarten Fensters liegen.“[Pla-88]

Der hier zu betrachtende Bildschirmausschnitt ist die Zeile. Da diese über die gesamte logische Einheit geht und alle Widgets vollständig innerhalb der Einheit liegen, gibt es für die Zeile nur eine obere und eine untere Kante. Es werden keine Begrenzungen auf der linken und rechten Seite

der Zeile für diese Betrachtungen benötigt. Des weiteren ist bemerkenswert, dass alle Widgets eine rechteckige Form haben und im rechten Winkel zu der Zeile angeordnet sind. Dies vereinfacht die Problematik erheblich. Es ist ausreichend, wenn jedes Widget auf seine linke Kante reduziert wird. Um zu ermitteln, wie viel Prozent des Widgets sich innerhalb der Zeile befinden, müssen nun lediglich die möglichen Schnittpunkte der linken Kante des Widgets mit der Zeile berechnet werden. Dann kann der Teil innerhalb der Zeile zur gesamten Kante ins Verhältnis gesetzt werden:

$$\text{Gewichtung} = \frac{\text{Laenge der Kante innerhalb der Zeile} * 100}{\text{Gesamtlänge der Kante}}$$

Beim Clipping können verschiedene Fälle auftreten, die berücksichtigt werden müssen. Diese Fälle sind in Abbildung 3.14 skizziert.

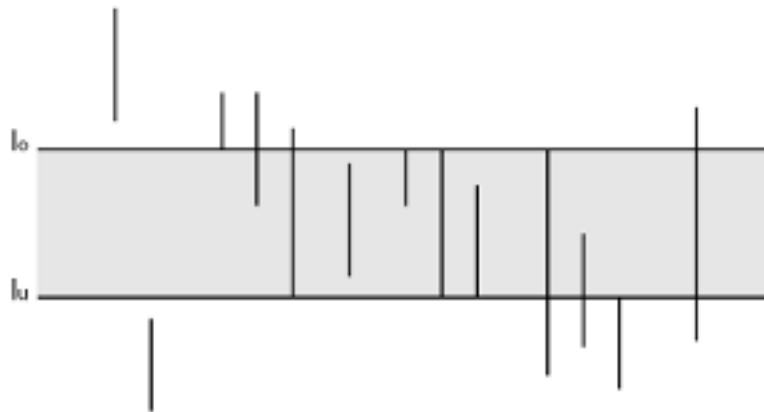


Abbildung 3.14: Clipping – Fallunterscheidungen

Datenstruktur für den gewichteten Graphen

Für die Umsetzung des gewichteten Graphen ist es notwendig, Überlegungen zur Datenstruktur anzustellen, die diesen Graphen im Programm nachbildet. Die einfachste Darstellungsweise für Graphen ist ihre Abbildung als sogenannte Adjazenzmatrix. Zwei Knoten w und z eines Graphen heißen adjazent oder benachbart, wenn es eine Kante von w nach z gibt. Ein Knoten w eines Graphen ist inzident mit einer Kante a des Graphen, wenn w ein Endknoten der Kante a ist.

Ist V die Anzahl der Knoten in einem Graphen, so ist die Größe einer Adjazenzmatrix standardmäßig $V \times V$. Der Inhalt der Matrix besteht aus booleschen Werten, wobei $a[w, z]$ auf *true* gesetzt wird, wenn w und z adjazent zueinander sind. In einem gewichteten Graphen werden die booleschen Werte durch die Gewichtung der Kanten ersetzt. Eine Kante mit der Gewichtung 0 bedeutet, die beiden entsprechenden Knoten sind nicht miteinander

Algorithm 10 Erstellung eines gewichteten Graphen – Teil 1

```
1: function GET_ZEILENBEGRENZUNG(widgets)
2:   erstelle leere Liste: Grenzen
3:   for w in widgets do
4:     füge die y-Koordinate von w zu Grenzen hinzu
5:   end for
6:   sortiere die Grenzen und entferne doppelte Werte
7:   return Grenzen
8: end function

9: function CLIPPING( $l_o, l_u, w_o, w_u$ )
10:                                     ▷  $l_o$  – obere Grenze der Zeile
11:                                     ▷  $l_u$  – untere Grenze der Zeile
12:                                     ▷  $w_o$  – obere Grenze des Widgets
13:                                     ▷  $w_u$  – untere Grenze des Widgets
14:   if  $w_o > l_u$  then
15:     return 0
16:   else if  $w_u < l_o$  then
17:     return 0
18:   else if  $w_u \leq l_u$  then
19:     if  $w_o \geq l_o$  then
20:       return 100
21:     else
22:        $p = w_u - l_o$ 
23:     end if
24:   else if  $w_o \geq l_o$  then
25:      $p = l_u - w_o$ 
26:   else if  $w_o < l_o$  and  $w_u > l_u$  then
27:      $p = l_u - l_o$ 
28:   end if
29:    $g = w_u - w_o$ 
30:   return  $p * 100 / g$ 
31: end function
```

verbunden. Bei einem lichten Graphen gibt es sehr viele 0-Werte in der Adjazenzmatrix. Es wird viel Speicherplatz für Kanten benötigt, die gar nicht existieren. Eine Alternative wäre der Einsatz von Adjazenzlisten. Zu jedem Knoten gibt es eine Liste mit Knoten, die mit diesem verbunden sind. Da auf den Knoten meist keine Reihenfolge definiert ist, kann ein und der selbe Graph durch unterschiedliche Adjazenzlisten-Strukturen dargestellt werden. In den meisten Algorithmen ist zudem die Handhabung einer Adjazenzmatrix einfacher als die von Adjazenzlisten. Eine inzidente Datenstruktur ist keine Alternative, da diese das Auslesen kompletter Zeilen erschwert.

Für die Baumtransformation soll eine Adjazenzmatrix verwendet werden. Diese kann aufgrund der Eigenschaften der Knoten verkleinert werden. Eine Zeile der Matrix entspricht einer Zeile in einer logischen Einheit und einer Spalte der Matrix entspricht einem Widget. Da es zwischen zwei Zeilen-Knoten bzw. zwischen zwei Widget-Knoten keine Kanten geben kann, entspricht solch eine Anpassung der Matrix einer Entfernung aller Zeilen und Spalten, die nur 0-Werte enthalten würden.

Algorithm 11 Erstellung eines gewichteten Graphen – Teil 2

```

32: function ERSTELLE_ADJAZENZMATRIX(widgets)
33:   Grenzen = GET_ZEILENBEGRENZUNG(widgets)
34:   füge die unter Kante der Einheit zu Grenzen hinzu
35:   erstelle leere Matrix           ▷ Zeilenanzahl: Anzahl Grenzen-1
36:                                   ▷ Spaltenanzahl: Anzahl Widgets
37:   initiiere Matrix mit 0 Werten
38:   for i in Zeilennummern der Matrix do
39:     for j in Spaltennummern der Matrix do
40:       o = obere Kante des j.ten Widgets
41:       u = untere Kante des j.ten Widgets
42:       Matrix[i][j] = clipping(Grenzen[i], Grenzen[i+1], o, u)
43:     end for
44:   end for
45:   return Matrix
46: end function

```

3.3.6 Auflösen von Konflikten

Nachdem die Adjazenzmatrix mit den Gewichtungen aufgestellt wurde, muss im letzten Schritt geprüft werden, ob jedes grafische Element genau einer Zeile zugeordnet ist. Treten hierbei Konflikte auf, so kommen die Regeln 4 bis 7 aus Abschnitt 3.2 zum Einsatz.

Zunächst einmal werden die Maximal-Gewichtungen für jedes Widget bestimmt (Regel 4 und 5). Alle anderen Gewichtungen, die ungleich 0 sind werden negiert. Damit ist gewährleistet, dass die Information der Gewich-

Algorithm 12 Anwenden von Regel 4 bis 7 – Teil 1

```
1: function BESTIMME_MAX_GEWICHTE(matrix)
2:   erschelle leere Liste: max
3:   for jede Spalte in matrix do
4:     ermittle den höchsten Wert in der Spalte
5:     gibt es mehrere Werte, so wähle den mit dem kleinsten
       Zeilen-Index ▷ Regel 5
6:     multipliziere alle anderen Werte in der Spalte mit  $-1$ 
7:     hänge den Zeilen-Index für den höchsten Wert an max an
8:   end for
9:   return max
10: end function
```

Algorithm 13 Anwenden von Regel 4 bis 7 – Teil 2

```
11: function VERSCHIEBE_EINZELNE_WIDGETS(matrix, zeile, widget)
       ▷ max – Liste, die zu jedem Widget die Zeilennummer mit der
       Maximal-Gewichtung speichert
12:   matrix[zeile][widget] = 0
13:   suche kleinste Gewicht für widget aus der matrix
14:   if kein Gewicht ungleich 0 für widget gefunden then
15:     setze Gewicht für widget in zeile auf 100
16:   else
17:     multipliziere das gefundene Gewicht mit  $(-1)$ 
18:   end if
19:   return Zeile mit Maximal-Gewichtung für widget
20: end function

21: function VEREINE_ZEILEN_NACH_R7(matrix, zeilen, i, j)
       ▷ i – Index der Zeile, die erhalten bleibt
       ▷ j – Index der Zeile, die zur i.ten Zeile hinzugefügt wird
22:   for jede Spalte w in der matrix do
23:     if matrix[i][w] > 0 or matrix[j][w] > 0 then
24:       matrix[i][w] = Betrag(matrix[i][w]) + Betrag(matrix[j][w])
25:     else
26:       matrix[i][w] = matrix[i][w] + matrix[j][w]
27:     end if
28:   end for
29:   entferne Zeile j aus matrix
30:   entferne Zeile j aus zeilen
31: end function
```

Algorithm 14 Anwenden von Regel 4 bis 7 – Teil 3

```
32: function LOESE_KONFLIKTE(matrix, widgets, zeilenGrenzen)
33:   max = BESTIMME_MAX_GEWICHTE(matrix)           ▷ Regel 4
34:   konflikte = alle Indizes, deren Werte nur einmal in max enthalten
   sind
35:   ▷ Die Indizes entsprechen den Spalten der Matrix, den Widgets
36:   while konflikte existieren do
37:     wIndex = erster Wert aus konflikte
38:     zIndex = Wert aus max, der wIndex zugeordnet ist
39:     if matrix[zIndex][wIndex] < 100 then           ▷ Regel 6
40:       zeile = VERSCHIEBE_EINZELNE_WIDGETS(matrix, zIndex,
   wIndex)
41:       max[wIndex] = zeile
42:       konflikte = alle Indizes, deren Werte nur einmal in max ent-
   halten sind
43:     else                                           ▷ kein Konflikt
44:       entferne ersten Eintrag aus konflikte
45:     end if
46:   end while
47:   for jede Zeile do                                 ▷ Regel 7
48:     if Zeile ist nicht in max then
49:       if Zeile ist die erste Zeile then
50:         VEREINE_ZEILEN_NACH_R7(matrix, zeilenGrenzen, 0, 1)
51:       else
52:         VEREINE_ZEILEN_NACH_R7(matrix, zeilenGrenzen,
   Zeile-1, Zeile)
53:       end if
54:     end if
55:   end for
56: end function
```

tung des Widgets für die entsprechende Zeile nicht verloren geht und in folgenden Schritten des Algorithmus genutzt werden kann. Zusätzlich zur Matrix wird eine Liste angelegt, die die Indizes der Zeilen enthält, die die Maximal-Gewichtungen enthalten. Diese Liste ist so sortiert, dass jeder Eintrag einem Element in der Liste mit den Widgets zugeordnet werden kann.

Laut Regel 6 können weitere Konflikte auftreten, wenn ein Element das einzige in einer Zeile ist, gleichzeitig aber noch in anderen Zeilen auftritt. Deshalb werden aus der Liste zu den Maximal-Gewichtungen alle Werte herausgesucht, die nur einmal in der Liste vorkommen. Dies sind potentielle Konflikt-Kandidaten. Solange wie mögliche Konflikte gefunden werden, wird nun überprüft, welchen Wert das entsprechende Widget in der entsprechenden Zeile hat. Ist dieser Wert 100, so kann die Zeile aus der Liste mit den möglichen Konflikt-Kandidaten gestrichen werden. Wenn der Wert jedoch kleiner als 100 ist, so werden, wie in Regel 6 beschrieben, die Gewichtungen umverteilt. Danach wird die Liste mit den möglichen Konflikt-Kandidaten aktualisiert.

Wenn alle Konflikte beseitigt sind, können eventuell entstandene leere Zeilen entfernt werden (Regel 7).

Damit ist die Baumtransformation abgeschlossen. Die Adjazenzmatrix bildet einen gewichteten Graphen ab, in dem jeder Widget-Knoten mit genau einem Zeilen-Knoten verbunden ist. Das Auslesen der Inhalte der einzelnen Zeilen ist aufgrund der gewählten Datenstruktur einfach zu handhaben. Wenn die Liste mit Widgets vor dem Aufbau der Adjazenzmatrix nach der horizontalen Positionierung der Widgets (x-Werte der linken oberen Ecke eines Widgets) sortiert werden, dann sind die Zeilen in der Adjazenzmatrix sofort einsatzfähig. Andernfalls müssen die Widgets vor der Ausgabe noch in die richtige Reihenfolge gebracht werden.

3.4 Darstellung auf dem Brailledisplay

Für die Ausgabe der einzelnen Zeilen müssen verschiedene Facetten der Brailledarstellung Berücksichtigung finden.

Es gibt eine ganze Reihe von Brailledisplays mit unterschiedlichen Größen. Dementsprechend kann nicht jedes Display die gleiche Menge von Informationen darstellen. Für diese Arbeit sollen die gängigen Braillezeilen mit 20, 40 oder 80 Braillemodulen betrachtet werden. Jedes Modul erlaubt die Projektion von einem Braillezeichen. Zusätzlich wird gezeigt, wie das Hyperbrailledisplay mit 7200 berührungsempfindlichen Stiften eine Ausgabe im Flächenmodus anzeigen kann. Dies entspricht einer Auflösung von 60x120 Pins oder einer Abbildung von 6 Zeilen mit je 60 Zeichen.

Da selbst 80 Zeilen nicht für die Darstellung vieler Zeilen ausreichen, wird der Flächenmodus in 2 Darstellungsformen eingeteilt: einer kompakten und einer normalen Form. Die kompakte Form soll einen Überblick der

Inhalte der Zeile präsentieren. Hierfür sollen möglichst viele Elemente dargestellt werden. Dies wird erreicht, indem für jedes Element nur ein begrenzter Platz zur Verfügung steht. Ist dieser nicht ausreichend, so ist nur der Anfang des Elementes zu sehen. Möchte der Nutzer das gesamte Element sehen, so kann er die Anzeige über die Routing-Tasten einer Braillezeile oder die Berührungsempfindlichen Stifte beim Hyperbrailledisplay aktivieren. In der normalen Form werden alle Elemente komplett dargestellt. Dies kann zur Folge haben, dass nur der Anfang einer Zeile angezeigt wird und der Anwender scrollen muss, um die restlichen Inhalte zu sehen.

Um einzelne Elemente einer Zeile auseinander halten zu können, wird zwischen ihnen ein Freiraum von mehreren Zeichen gelassen. Weiterhin ist es sinnvoll, die Widgets für den Nutzer zu kennzeichnen, die spezielle Aktionen erlauben. So können Kontrollkästchen wie im Strukturmodus durch `[]` oder `[x]` ergänzt werden. Eingabefelder sollten mit Hilfe eines Cursors markiert sein und Auswahlboxen um ihren Inhalt eckige Klammern platziert bekommen. Enthält eine Zeile gleichartige Elemente, wie Menüpunkte, Buttons oder Tabellenzellen, so kann dies am Anfang der Zeile kurz deklariert werden. Dies hilft dem Verständnis des Nutzers.

Im folgenden werden kurz einige Beispiele für die Darstellung auf Brailledisplays gezeigt.

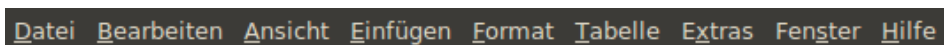


Abbildung 3.15: Menüzeile

Abbildung 3.15 zeigt eine Menüzeile, die folgendermaßen auf einem Brailledisplay abgebildet werden kann:

Display mit 20 Modulen
 Normal: `Datei` `Bearbeiten` `...`
 Kompakt: `Datei` `Bearb$` `Ansi` `...`

Display mit 40 Modulen:
 Normal: `Datei` `Bearbeiten` `Ansicht` `Einfügen` `...`
 Kompakt: `Datei` `Bearb$` `Ansic$` `Einfü$` `Format` `...`

Display mit 80 Modulen:
 Normal und Kompakt:
`mnu: Datei` `Bearbeiten` `Ansicht` `Einfügen` `Format`
`Tabelle` `Extras` `Fens` `...`

Bei diesen Darstellungen sind zwischen den Elementen jeweils 3 freie Module. Ein `$` am Ende eines Elementes bedeutet, dass dieses nicht vollständig dargestellt ist und `...` am Ende der Zeile zeigen an, dass der Nutzer

scrollen kann, um zu dem restlichen Inhalt zu gelangen. Da das Beispiel nur aus einer Zeile besteht, macht es keinen Sinn eine Darstellung für das Hyperbrailledisplay aufzuzeigen. Diese würde sich für eine einzelne Zeile nicht von den obigen Anzeigen unterscheiden.

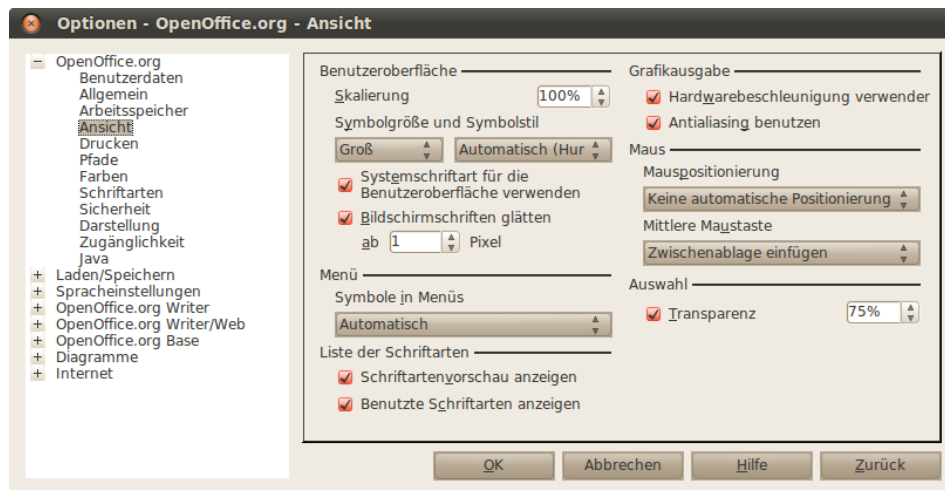


Abbildung 3.16: Dialog der Anwendung Open Office.org

Ein sinnvolles Beispiel für die Darstellung auf dem Hyperbrailledisplay soll für Abbildung 3.16 konstruiert werden.

Display mit 6 Zeilen und je 60 Modulen:

Normal und Kompakt:

Benutzeroberfläche		Grafikausgabe	
Skalierung	[100%]	[x]	Hardwarebeschleunig...
Symbolgröße und Symbolstil	[Groß] [Automatisch(Hu]	[x]	Antialiasing benutzen
[x]	Systemschriftart für \$		Maus
			Mauspositionierung
			[Keine automatische Posit\$]

Beim mehrzeiligen Brailledisplay sollte möglichst eine Anordnung verwendet werden, die der auf dem Bildschirm entspricht. Dies kann durch die Auswertung der Widget-Breite im Verhältnis zur Breite der Einheit geschehen. Bei sehr breiten Einheiten kann auch bei solchen Displays mit Scrollen des Bildschirmausschnittes nach links oder rechts gearbeitet werden. Die Zeilen bleiben, so wie bei der Baumtransformation ermittelt, erhalten.

Kapitel 4

Umsetzung

4.1 Implementierung

Die in Abschnitt 3.3 dargestellte Baumtransformation ist nicht bis ins kleinste Detail beschrieben. Die Implementierung schließt diese Lücken. Hierbei werden unter anderem Lösungen für das Zusammenfügen mehrerer Adjazenzmatrizen und das Auslesen von Widgets einer Einheit aus dem Baum erstellt. Diese Details können dem Quellcode entnommen werden.

4.1.1 Integration in den Screenreader SUE

Eine Umsetzung der Baumtransformation erfolgt für den Linux-Screenreader SUE (Screenreader & Usability Extensions). SUE nutzt die in Abschnitten 2.2.4 und 2.3 beschriebenen Linux-Schnittstellen und ist damit sehr gut für die Umsetzung geeignet. Der Einsatz der Programmiersprache Python erlaubt die Verwendung des Python-Wrappers `pyatspi` für die Accessibility Schnittstelle.

Der Screenreader ist ein Open Source Produkt und kann unter der folgenden Adresse frei heruntergeladen werden:

```
http://sourceforge.net/projects/sue/
```

Der modulare Aufbau des Screenreaders SUE gewährleistet eine einfache Integration des Algorithmus.

Abbildung 4.1 gibt einen Überblick des Zusammenspiels zwischen der Accessibility Schnittstelle, dem Screenreader und der Ausgabe zu den Braillegeräten. Den Kern von SUE bildet die **Access Engine**. Diese übernimmt die Kommunikation mit AT-SPI und damit alle Abfragen an den Accessibility Baum. Die Aufbereitung der Inhalte für die Ausgabegeräte wird von Scripten übernommen. Zwischen dem Kern und den Scripten gibt es eine Kommunikationsschicht, die **AccessEngineAPI**. Dies ist die Schicht, in

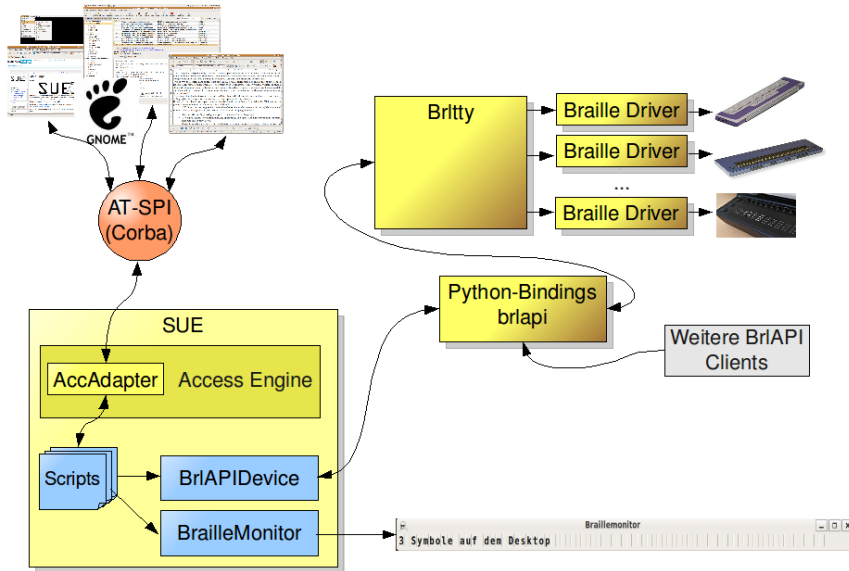


Abbildung 4.1: Kommunikationswege der Accessibility Architektur

der die Baumtransformation integriert wird. Bei dieser Vorgehensweise können verschiedene Scripte die Transformation nutzen, um ihre Inhalte aufzubereiten. Zu diesen Scripten zählen neben dem **BrailleScript** auch das **MagnificationScript** für die Abbildung des Großschriftmonitors und das **ReviewScript** für den Erkundungsmodus.

Für ein effizientes Arbeiten sollte die Matrix nicht bei jeder Aktion des Anwenders neu erstellt werden. So wird beim Schreiben in ein Textfeld nur der Inhalt des Accessibles verändert, aber nicht seine Anordnung oder seine Größe. Entsprechend ist der Neuaufbau der Matrix bei jeder Zeicheneingabe überflüssig. Für die Umsetzung des Flächenmodus wird die Matrix immer dann neu erstellt, wenn das Fenster oder der Fokus gewechselt wird. Desweiteren können die Daten der Matrix von mehreren Scripten genutzt werden. Dazu werden sie im **AETier** in dem Attribut **area_matrix** gespeichert. Das **AETier** ist ein Objekt, das für jede Anwendung erstellt wird. Es speichert und verwaltet anwendungsspezifische Daten, die der Screenreader benötigt und steuert die Ausführung der Scripte.

Für die Einführung des Flächenmodus ist eine Umschaltmöglichkeit zwischen dem Struktur- und dem Flächenmodus notwendig. Die benötigten Strukturen wurden bereits bei der Umsetzung des Strukturmodus implementiert. Als Auslöser für den Wechsel zwischen den Modi wird das Tastenkürzel **Alt+Shift+X** gewählt. Dieses wird im Basis-Script, dem **BasicEventScript**, implementiert und mit den entsprechenden Aktionen verknüpft.

Für die Ausgabe auf der Braillezeile werden die Daten der Matrix vom `BrailleScript` aufbereitet. Das bedeutet, das `BrailleScript` liest die entsprechenden Zeilen aus und fragt die anzuzeigenden Inhalte zu den zugehörigen Widgets ab. Anschließend erfolgt eine Analyse der Breite der einzelnen Widgets, um den Platz zu bestimmen, den diese auf dem Brailledisplay zugewiesen bekommen. Dabei wird auch berücksichtigt, ob die Anzeige in kompakter oder normaler Form erfolgen soll (siehe Abschnitt 3.4). Als Ergebnis schickt das `BrailleScript` den anzuzeigenden String an das Braillegerät.

4.1.2 Umsetzung der Baumtransformation

Die Baumtransformation kann im folgenden Modul gefunden werden:

```
src\AccessEngine\AccessEngineAPI\Output.py
```

Der Algorithmus ist modular aufgebaut und setzt sich aus den nachstehenden Funktionen zusammen:

- **setAreaMatrix** – arbeitet die Transformation wie im Algorithmus 1 ab und speichert die Matrix im AETier.
- **_getWindow** – ein kleines Modul, zum effizienten Abfragen des Fenster-Knotens.
- **_findParentCluster** – bestimmt die aktive Einheit wie in Algorithmus 3 gezeigt.
- **_findAllCluster** – sucht alle Einheiten in einem Fenster wie in den Algorithmen 4 und 5 beschrieben.
- **_sortAllCluster** – sortiert alle gefundenen Einheiten (siehe Algorithmus 6).
- **_getActiveCluster** – bestimmt die aktive Einheit in einer sortierten Liste von Einheiten (siehe Algorithmus 8).
- **_getWidgetsInCluster** – sucht alle Widgets in einem Teilbaum (siehe Algorithmus 9).
- **_sortWidgets** – sortiert die gefundenen Widgets in Zeilen ein und nutzt dabei die aufgestellten Regeln aus Abschnitt 3.2 zur Konfliktbe-seitigung (siehe Algorithmen 10, 11, 12, 13, 14).

Um die Interprozesskommunikation zu verringern und nicht jeden Knoten einzeln zu analysieren, stellt `pyatspi` drei Methoden zur Verfügung, die Massenabfragen erlauben. Beim Einsatz dieser Methoden muss allerdings

berücksichtigt werden, dass auch für jeden Knoten, der durch `pyatspi` analysiert wird, ein ATK-Objekt erstellt wird. Bei dem Durchsuchen größerer Teilbäume wird sich dies durch verzögerte Reaktionszeiten bemerkbar machen. Im einzelnen sind die erwähnten Methoden:

- `pyatspi.utils.findAncestor` – gibt den nächsten Vorfahren mit einer bestimmten Eigenschaft zurück.
- `pyatspi.utils.findDescendant` – gibt den nächsten Nachfahren mit einer bestimmten Eigenschaft zurück.
- `pyatspi.utils.findAllDescendants` – sucht alle Nachfahren mit bestimmten Eigenschaften aus dem Baum.

Diese Funktionen gehen von einem bestimmten Knoten im Baum aus, der beim Aufruf mit übergeben wird. Die zu erfüllenden Eigenschaften der Ziel-Knoten wird durch ein ebenfalls zu übergebenes Prädikat bestimmt. Dieses Prädikat kann ganz einfach aufgebaut sein:

```
lambda x: x.getRole() == 'push button'
```

oder wie in Algorithmus 15 kompliziertere Anforderungen stellen.

Der Baum wird solange durchsucht, bis das Prädikat `True` zurückgibt oder kein entsprechender Knoten gefunden werden kann.

`pyatspi.utils.findAncestor` wird in dem Modul `_findParentCluster` eingesetzt, `pyatspi.utils.findDescendant` kommt im Modul `_findAllCluster` zu Anwendung und `pyatspi.utils.findAllDescendants` holt alle Widgets des Teilbaumes im Modul `_getWidgetsInCluster`.

Eine weitere Reduzierung der Interprozesskommunikation wird über das Zwischenspeichern von bereits erfragten Eigenschaften zu den einzelnen Widgets erreicht. Zu den Eigenschaften, die in mehreren Modulen der Transformation benötigt werden, gehören die Rolle, die Positionierung und die Größe eines Widgets.

4.2 Testen und Evaluieren des Algorithmus

4.2.1 Systematisches Testen

Das Ziel der Tests war das Aufspüren möglichst vieler Fehler. Dazu wurde zunächst das Whitebox-Testverfahren eingesetzt. Um etwas Abstand zum Quelltext zu entwickeln und das Phänomen der Betriebsblindheit abzuschwächen wurden diese erst zwei Wochen nach dem Abschluss der Programmierarbeiten durchgeführt. Bei den Whitebox-Tests wurde mit verschiedenen Testfällen jeder Programmpfad der Baumtransformation durchlaufen. Ein Ziel war es, möglichst viele Grenzfälle abzudecken, da diese eine große

Algorithm 15 Definition eines Prädikates

```
1: function PREDICATE(acc, roles, jumpNodes)
2:   for node in jumpNodes do
3:     if acc.isEqual(node) then
4:       return False
5:     end if
6:     try: component = acc.queryComponent()
7:     except NotImplementedError: return False
8:     box = component.getExtents(pyatspi.DESKTOP_COORDS)
9:     if box.x >= 0 and box.y >= 0 and box.width > 0 and
    box.height > 0 then
10:      role = acc.getRoleName()
11:      if role in roles then
12:        return True
13:      end if
14:    end if
15:    return False
16:  end for
17: end function

18: pyatspi.utils.findDescendant(acc,
19:   lambda x: predicate(x, ROLES_CLUSTER, jumpNodes), True)
```










Fehlerquote haben. Gefundene Fehler wurden sofort behoben, um anschließend den Test noch einmal fehlerfrei auszuführen.

In der nächsten Testrunde kam das Blackbox-Verfahren zum Einsatz. Hierbei wurde anhand verschiedener Testfälle überprüft, ob die Trennung in Teilbäume und der Aufbau der Adjazenzmatrix den Erwartungen entsprechen. Auch hier wurden gefundene Fehler sofort behoben. Die Testfälle wurden solange wiederholt, bis alle fehlerfreie Ergebnisse lieferten.

4.2.2 Performance-Tests

In dieser Arbeit wurde bisher davon ausgegangen, dass Zugriffe auf den AT-SPI Baum teuer sind. Dies soll nun anhand von Testfällen mit Daten belegt werden.

Testfälle

-  **Testfall I:** offenes Dokument in OpenOffice.org Textverarbeitung; der Fokus liegt auf der Werkzeugleiste ‚Standard‘
-  **Testfall II:** offenes Dokument in OpenOffice.org Textverarbeitung; der Fokus liegt auf einer Tabelle im Dokument
-  **Testfall III:** Dialog ‚Optionen‘ in OpenOffice.org Textverarbeitung; der Fokus liegt auf einem Textfeld im Unterpunkt ‚Benutzerdaten‘
-  **Testfall IV:** Dialog ‚Optionen‘ in OpenOffice.org Textverarbeitung; der Fokus befindet sich im Baum auf der linken Seite
-  **Testfall V:** Editor GEdit; der Fokus liegt auf der Werkzeugleiste
-  **Testfall VI:** Dialog ‚Einstellungen‘ im Editor GEdit; Der Fokus liegt auf dem Seitenreiter ‚Ansicht‘
-  **Testfall VII:** Der Fokus liegt auf einem Symbol auf dem Desktop
-  **Testfall VIII:** E-Mail-Client Evolution; der Fokus liegt auf dem Button ‚Kalender‘
-  **Testfall IX:** Taschenrechner ‚GCalc‘; der Fokus liegt auf dem Button ‚Lös‘

Messungen

Zum Vergleich wird die Matrix in der ersten Performance-Messung für das gesamte Fenster erstellt. Dabei werden drei Zeiten genommen:

- **Funktion 1:** Die Matrix wird für das gesamte Fenster erstellt. Die Funktion besteht aus den Teilfunktionen 1a und 1b.
- **Funktion 1a:** Der AT-SPI Baum wird für das gesamte Fenster ausgelesen. (siehe Algorithmus 9)
- **Funktion 1b:** Die Daten, die in 1a ausgelesen wurden werden verarbeitet. Das Ergebnis ist die Adjazenzmatrix, die die Einteilung der Widgets in Zeilen angibt. (siehe Algorithmen 10, 11, 12, 13, 14)

Fall	1	1a	1b
I	1,107933	0,991148	0,076842
II	1,277260	1,186505	0,058931
III	0,330537	0,279515	0,027554
IV	0,354665	0,292079	0,040294
V	0,724761	0,700958	0,006950
VI	0,207562	0,181258	0,007519
VII	0,117118	0,076631	0,024316
VIII	0,567206	0,540661	0,007741
IX	0,228655	0,204585	0,007741

Tabelle 4.1: Performance Messungen in Sekunden – ohne Zerlegung des Baumes

Die zweite Reihe der Messungen bezieht sich auf die Baumtransformation, wie im Abschnitt 3.3 beschrieben:

- **Funktion 2:** Bestimmung der aktiven Einheit (siehe Algorithmus 3)
- **Funktion 3:** Erstellen der Matrix zur aktiven Einheit (siehe Algorithmen 9, 10, 11, 12, 13, 14)
- **Funktion 3a:** Der AT-SPI Baum wird für die aktive Einheit ausgelesen. (siehe Algorithmus 9)
- **Funktion 3b:** Die Daten, die in 3a ausgelesen wurden werden verarbeitet. Das Ergebnis ist die Adjazenzmatrix, die die Einteilung der Widgets in Zeilen angibt. (siehe Algorithmen 10, 11, 12, 13, 14)
- **Funktion 4:** Das Bestimmen und sortieren aller Einheiten. (siehe Algorithmen 4, 5, 6, 8)
- **Funktion 5:** Die gesamte Baumtransformation wie in Algorithmus 1 beschrieben.

Fall	2	3	3a	3b	4	5
I	0,001536	0,125904	0,098968	0,007285	0,075357	0,509380
II	0,011155	0,348921	0,296556	0,021444		0,360664
III	0,097454	0,015308	0,000037	0,004881		0,340774
IV	0,012280	0,228910	0,206705	0,007640		0,241692
V	0,004602	0,088208	0,065570	0,004640	0,046246	0,745620
VI	0,002125	0,193828	0,156282	0,012752		0,203351
VII	0,008689	0,089932	0,061795	0,011518		0,100117
VIII	0,119243	0,012228	0,000026	0,010303		0,147284
IX	0,114908	0,004639	0,000018	0,003823		0,170959

Tabelle 4.2: Performance Messungen der Baumtransformation in Sekunden

Interpretation der Ergebnisse

In den Testfällen sollten mindestens 6 Bildschirmzeilen bestimmt werden. Dies entspricht der Größe des BrailleDis 9000 aus dem Hyperbraille-Projekt (siehe Abschnitt 2.5.2).

Die 4. Funktion wird bei den Testfällen II, IV, VI und VII nicht benötigt, da genügend Zeilen vorhanden sind.

In den Fällen III, VIII und IX werden bereits alle Einheiten in der 2. Funktion bestimmt und als Teil der Auswertung müssen die Widgets der aktiven Einheit betrachtet werden. Diese werden zwischengespeichert und damit ist ein Auslesen des AT-SPI Baumes in Funktion 3a für die Testfälle III, VIII und IX unnötig.

Die erste Feststellung, die anhand der Performance-Messungen getroffen werden kann, ist, dass das Auslesen des AT-SPI Baumes in den meisten Testfällen ca. 90% der benötigten Zeit in Anspruch nimmt.

Der Vergleich zwischen Funktion 1 und 5 zeigt, dass die Baumtransformation, die in dieser Arbeit vorgestellt wurde mindestens genauso gut ist, wie die Einteilung des gesamten Fensters in Zeilen. In großen Fenstern, für die Funktion 1 mehr als eine Sekunde braucht, konnte die benötigte Zeit in Funktion 5 erheblich verringert werden; in Testfall II zum Beispiel um einen Faktor von 3,5. Eine Ausnahme bildet der Testfall V. Hier sorgen ungünstige Umstände dafür, dass auch in Funktion 5 das gesamte Fenster ausgelesen wird. Da das Dokument während des Tests ohne Inhalt war, wurde die Menüleiste, die Werkzeugleiste, das Dokument und die Statusleiste benötigt, um 6 Bildschirmzeilen zu ermitteln. Für Dialoge und kleinere Anwendungen kann Funktion 5 nur im geringeren Maße eine bessere Performance bieten. Allerdings ist die benötigte Zeit bereits in Funktion 1 akzeptabel und ermöglicht ein flüssiges Arbeiten.

Der Algorithmus kann an der einen oder anderen Stelle sicherlich effektiver umgesetzt werden. So ist es denkbar, die Sortierverfahren durch effizientere zu ersetzen. Allerdings zeigen die Zeitmessungen, dass für das

Auswerten der ermittelten Knoten aus dem Baum wenig Zeit benötigt wird. Deshalb ist es nicht sehr sinnvoll, viel Zeit und Aufwand in die Umsetzung von effizienteren Verfahren zu investieren.

4.2.3 Regel 6

Beim Erstellen der Regeln in Abschnitt 3.2 standen zwei verschiedene Lösungswege für die Auflösung des Konfliktes in Regel 6 zur Diskussion. Diese Varianten sind:

✓ **Regel 6 (a):** Ist ein Widget das einzige Element in einer Zeile, kommt aber noch in anderen Zeilen vor, so werden Zeilen zusammengelegt. Befindet sich die Oberkante des Widgets in der betroffenen Zeile, so wird diese mit der darunter liegenden Zeile vereint. In allen anderen Fällen wird sie zur darüber liegenden Zeile hinzugefügt.

oder

✓ **Regel 6 (b):** Ist ein Widget das einzige Element in einer Zeile, kommt aber noch in anderen Zeilen vor, so wird die Gewichtung in der Zeile ohne weiterer Elemente auf 0 gesetzt.

Nach ersten Überlegungen schien Variante (a) die besseren Ergebnisse zu liefern. Deshalb wurde zunächst diese Variante von Regel 6 in der Baumtransformation verwendet. Anhand eines Korrektheitsbeweises soll nun überprüft werden, ob die Ergebnisse in jedem Fall richtig sind.

Korrektheitsbeweis für Variante (a)

Vorraussetzung:

G – Graph

V – Menge aller Knoten

L – Menge aller Knoten, die Zeilen repräsentieren

W – Menge aller Knoten, die Widgets repräsentieren

E – Menge aller Kanten

$G = (V, E)$

$V = L \cup W$, L und W sind disjunkte Teilmengen von V

$E \subseteq [L, W]$

$L = \{l_1, \dots, l_n\}; |L| = n$

$W = \{w_1, \dots, w_m\}; |W| = m$

Gewichtungsfunktion:

$$g([l, w]) = \{x \mid x \in \mathbb{R}; -100 < x \leq 100;$$

$$\forall w \in W : \sum_{k=1}^n |g([l_k, w])| = 100\}$$

$$[l, w] \in E, \text{ wenn } g([l, w]) \neq 0$$

Maximal-Gewichtung:

$$\max(w) = \{l_k \mid \forall g([l_h, w]) : g([l_k, w]) > g([l_h, w]) \vee$$

$$(g([l_k, w]) = g([l_h, w]) \wedge k < h)\}$$

Vereinigung von zwei Zeilen:

$$p(l_{k-1}, l_k) = \{l \mid \forall w \in W :$$

$$((\forall g([l_{k-1}, w]) > 0 \vee \forall g([l_k, w]) > 0) :$$

$$g([l, w]) = |g([l_{k-1}, w])| + |g([l_k, w])|) \wedge$$

$$((\forall g([l_{k-1}, w]) \leq 0 \wedge \forall g([l_k, w]) \leq 0) :$$

$$g([l, w]) = g([l_{k-1}, w]) + g([l_k, w]))\}$$

Behauptung:

Wenn für zwei Widgets $w_i, w_j \in W$ mit $i \neq j$ gilt:

$$\{\text{wenn } [l_k, w_i] \in E, \text{ dann } [l_k, w_j] \notin E \mid \forall l_k \in L\}$$

dann werden w_i und w_j niemals ihre Maximal-Gewichtungen in der gleichen Zeile haben ($\max(w_i) \neq \max(w_j)$).

Beweis:

Fall 1: Regel 6 wird durch w_i in Zeile l_k ausgelöst

$$\forall w \text{ in } W : g([l_k, w]) \leq 0, \text{ wenn } w \neq w_i$$

$$\max(w_i) = l_k$$

$$g([l_k, w_j]) = 0$$

Fall 1(a): Die Oberkante des Widgets liegt in l_k

$$\begin{aligned} g([l_{k-1}, w_i]) &= 0 \\ g([l_{k+1}, w_j]) &= 0, \text{ da } g([l_{k+1}, w_i]) < 0 \end{aligned}$$

$$\begin{aligned} g([l_k, w_i]) + |g([l_{k+1}, w_i])| &> 0 \\ g([l_k, w_j]) + g([l_{k+1}, w_j]) &= 0 \end{aligned}$$

Zeile l_k wird entfernt, alle nachfolgenden Indizes verringern sich um 1, so dass l_{k+1} zu l'_k wird.

$$\begin{aligned} \max(w_i) &= l'_k \\ g([l'_k, w_j]) &= 0 \\ \max(w_i) &\neq \max(w_j) \end{aligned}$$

Fall 1(b): Die Oberkante des Widgets liegt nicht in l_k

$$\begin{aligned} g([l_{k-1}, w_i]) &< 0 \\ g([l_{k-1}, w_j]) &= 0 \end{aligned}$$

Es werden Zeilen l_k und l_{k-1} vereint. Dies geschieht analog zu Fall 1(a).

Fall 2: Regel 6 wird durch w_j in Zeile l_k ausgelöst
Dieser Fall ist analog zu Fall 1.

Fall 3: Regel 6 wird durch w_a in Zeile l_k ausgelöst Zeile l_k wird mit Zeile l_{k-1} vereint ($p(l_k, l_{k+1})$ kann analog dazu gezeigt werden)

$$\begin{aligned} \max(w_a) &= l_k \\ \forall w \text{ in } W : g([l_k, w]) &\leq 0, \text{ wenn } w \neq w_a \end{aligned}$$

Problemfall:

$$\begin{aligned} g([l_k, w_i]) &< 0, \text{ dann } g([l_k, w_j]) = 0 \\ g([l_{k-1}, w_i]) &> 0 \\ g([l_{k+1}, w_j]) &< 0, \text{ dann } g([l_{k+1}, w_i]) = 0 \\ p(l_{k-1}, l_k) &= l'_{k-1} \\ g([l'_{k-1}, w_i]) &> 0 \\ g([l'_{k-1}, w_j]) &< 0 \\ \max(w_i) &\neq \max(w_j) \\ \max(w_i) &= \max(w_a) \end{aligned}$$

Damit ist der Konflikt für w_a gelöst. Aber wenn es ein w_b gibt, für welches immer noch ein Konflikt in l'_k besteht, kann sich folgende Situation ergeben:

$$\begin{aligned}
&g([l'_k, w_b]) > 0, \max(w_b) = l'_k \\
&\forall w \text{ in } W : g([l'_k, w]) \leq 0, \text{ wenn } w \neq w_b \\
&g([l'_k, w_j]) < 0 \\
&g([l'_{k-1}, w_b]) < 0 \\
&\Rightarrow p(l'_{k-1}, l'_k) = l''_{k-1} \\
&g([l''_{k-1}, w_i]) > 0 \\
&|g([l''_{k-1}, w_j])| > \max(w_j), \text{ dann } \max(w_j) = l''_{k-1} \\
&\max(w_i) = \max(w_j)
\end{aligned}$$

Diese Situation führt zu einem Widerspruch. Damit wurde bewiesen, dass die Anwendung von Regel 6 Variante (a) nicht in jedem Fall korrekte Ergebnisse liefert. Durch die Ergänzung einer Zusatzbedingung, kann dies berichtigt werden:

$$p(l_k, l_{k+1}) = l_k, l_{k+1}, \text{ wenn:}$$

$$\begin{aligned}
&\exists w_i, w_j \in W : \\
&g([l_k, w_i]) \neq 0 \wedge g([l_{k+1}, w_i]) = 0 \\
&g([l_k, w_j]) = 0 \wedge g([l_{k+1}, w_j]) \neq 0
\end{aligned}$$

Diese Bedingung verbietet das Zusammenlegen von Zeilen unter bestimmten Umständen. Damit kann aber für den oben dargestellten Konflikt von w_a keine bessere Anordnung gefunden werden. Variante (b) hingegen liefert für solche Fälle ein besseres Ergebnis. Aus diesem Grund wurde entschieden, Regel 6 Variante (b) umzusetzen und die Baumtransformation wurde entsprechend angepasst.

Kapitel 5

Zusammenfassung und Ausblick

In dieser Arbeit wurde gezeigt, dass die Fähigkeiten blinder Anwender noch lange nicht ausgeschöpft sind und hier noch viel ungenutztes Potential zu finden ist. Dies bietet eine Reihe von Herausforderungen, um neue, zukunftsorientierte Arbeitstechniken für diese Zielgruppe zu entwickeln. Ein Schritt in diese Richtung ist die Entwicklung von Flächendisplays, die einen größeren Bildschirmausschnitt darstellen können. Die neu aufgestellte Baumtransformation zeigt, wie es unter Linux möglich ist, einen zusammenhängenden Bildschirmausschnitt auszulesen. Der Algorithmus kann, wie in der Umsetzung geschehen, für derzeitige Screenreader eingesetzt werden oder aber für die Realisierung neuer Arbeitsmethoden, wie sie in Abschnitt 2.5.2 dargestellt wurden.

Es wurden in dieser Arbeit nicht alle Probleme für die Umsetzung des Algorithmus gelöst. Der Schwerpunkt lag auf einer allgemeingültigen Lösung, die alle Teile der Oberfläche auslesen kann. Für die Anwendung OpenOffice.org Tabellenkalkulation ist die Umsetzung allerdings nicht performant genug. Bei dieser Anwendung hat die Tabelle 1.073.741.824 Child-Knoten. Davon sind abhängig von der Größe und Auflösung des Bildschirms wahrscheinlich maximal 1000 Knoten sichtbar. Die Herausforderung besteht darin, mit möglichst wenig Zugriffen auf den Baum festzustellen welche Knoten dies sind. Ein weiteres mögliches Problem könnten HTML Container sein. HTML Container bei der Anwendung Firefox unterscheiden sich von Document Frames bei der Anwendung OpenOffice.org Textverarbeitung in einem wesentlichen Punkt: Im HTML Container wird das gesamte Dokument abgebildet, während bei der Textverarbeitung nur der sichtbare Teil des Dokumentes im AT-SPI Baum zu finden ist. Aus diesem Grund müssen HTML Container noch einmal gesondert auf ihre Performance getestet werden. Bei Bedarf könnten zusätzliche Einheiten für die Darstellung von Webseiten festgelegt werden.

Derzeit wird von der GNOME-Community an einer neuen, überarbeiteten GNOME-Variante gearbeitet: GNOME 3. Die Interprozesskommunikation wird zukünftig mit Dbus anstatt mit Corba durchgeführt und die aktuell verwendeten Window-Manager Metacity und Compiz werden durch GNOME Shell ersetzt. Dies hat Auswirkungen auf Screenreader, die unter GNOME 3 zum Einsatz kommen. Für den Umstieg auf Dbus wird gegenwärtig AT-SPI 2 entwickelt. Sobald die neue Schnittstelle fertig gestellt ist, muss noch einmal geprüft werden, welche Veränderungen am Interface vorgenommen wurden. Allgemein ist davon auszugehen, dass sich am AT-SPI Baum und an der Programmierschnittstelle nicht allzu viele Details ändern werden, so dass die in dieser Arbeit entwickelte Baumtransformation auch für AT-SPI 2 eingesetzt werden kann. GNOME Shell ergänzt den aktuellen Aufbau der Oberfläche um einen neuen Workspace und eine Activity Leiste. Für die Integration in den Screenreader müssen die Arbeitsweise mit diesen neuen Elementen und deren Aufbau im AT-SPI Baum analysiert werden.

Ein interessanter Forschungsaspekt könnte eine automatisierte Analyse zur Einteilung des AT-SPI Baumes in Teilbäume sein. Wenn solch ein Ansatz verwirklicht werden könnte, wären zukünftige Anwendungen und Veränderungen der Oberflächen schnell in den Screenreader integrierbar.

Anhang A

Anordnung der Knoten im AT-SPI Baum – Heuristische Analyse

Zur Ermittlung der Knoten, die logische Einheiten repräsentieren, wurde eine Analyse des AT-SPI Baumes für mehrere Anwendungen durchgeführt (siehe Abschnitt 3.3.1). In diesem Anhang werden die Ergebnisse der Analyse dargestellt.

Zunächste wurden alle möglichen Rollenbezeichnungen aus der Beschreibung der AT-SPI Schnittstelle [SPI-1] entnommen. Dann wurde anhand einer Reihe von Beispiel-Fenstern geschaut, wie diese jeweils im Baum angeordnet sind. Dabei konnte festgestellt werden, dass die Anordnung der Knoten Gesetzmäßigkeiten unterliegt, die für das Erkennen von Einheiten ausgenutzt werden können.

Die nachfolgenden Betrachtungen beziehen sich auf einen Teilbaum, der ein Fenster repräsentiert. Solche Teilbäume starten mit Knoten, die direkt unterhalb eines Application-Knotens liegen und eine der folgenden Rollen haben: `dialog`, `alert`, `frame`, `file chooser`, `popup menu`, `window`.

menu bar, tool bar, status bar

- Es gibt nur ein Knoten mit der Rolle `menu bar`. Dieser ist eine Einheit.
- Es gibt beliebig viele Knoten mit der Rolle `tool bar`. Diese sind jeder für sich eine Einheit.
- Eine Statusbar kann Child-Knoten mit der Rolle `statusbar` haben. Es gibt in einem Fenster nur einen Statusbar-Knoten, der eine Einheit ist. Das ist der, der am weitesten oben in der Hierarchie steht.
- Wenn eine Einheit mit der Rolle `menu bar`, `tool bar` oder `statusbar` gefunden wurde, dann

- überprüfe, ob Sibling-Knoten Einheiten sind
- wenn es keine sichtbaren Sibling-Knoten gibt, oder nur Sibling-Knoten, die die gleiche Rolle haben wie der Ausgangs-Knoten, dann suche vom Parent-Knoten den nächsten Ancestor-Knoten, der mehr als ein Child-Knoten hat; überprüfe, ob die Sibling-Knoten von diesem Ancestor-Knoten Einheiten sind

page tab list

- Jeder Knoten mit der Rolle `page tab list` ist eine Einheit.
- untersuche die Sibling-Knoten:
 - wenn diese Widgets sind, dann ist der Parent-Knoten eine Einheit
 - wenn es Sibling-Knoten mit der Rolle `filler` gibt, ist dieser eine Einheit
- Alle Child-Knoten haben die Rolle `page tab`. Von diesen hat nur ein Knoten den Zustand `selected`. Nur dieser Knoten wird nach Widgets durchsucht. Die anderen können ebenfalls Widgets enthalten, sind aber inaktiv und werden nicht angezeigt.
- Alle Child-Knoten werden als Widgets behandelt, wenn sie sichtbar sind.

tree, tree table, table, list

- Knoten mit der Rolle `tree`, `tree table`, `table` oder `list` sind Einheiten, wenn sie am linken Fensterrand platziert sind.

split pane

- überprüfe, ob Sibling-Knoten Einheiten sind
- überprüfe die Nachkommen Ebenenweise, bis eine Einheit gefunden wurde

panel

- Panel-Knoten müssen nur betrachtet werden, wenn sie sichtbar sind.
- Panel-Knoten werden nur betrachtet, wenn sie Sibling-Knoten von interessanten Knoten sind.
- Wenn ein Panel-Knoten betrachtet wird, durchsuche die Nachfahren ebenenweise, bis
 - eine Einheit oder ein Split-Pane-Knoten gefunden wurde.

- ein Widget gefunden wurde. Untersuche alle Sibling-Knoten des Widgets:
 - * Wird eine Einheit gefunden, so ist der Parent-Knoten des Widgets eine Einheit.
 - * Wird keine Einheit gefunden, so ist der Panel-Knoten eine Einheit.

filler

- Ein Knoten mit der Rolle **filler** ist eine Einheit, wenn er sichtbar ist und Sibling-Knoten hat, die Einheiten sind.
- Ein Filler-Knoten, der sichtbar ist und als Sibling-Knoten von interessanten Knoten betrachtet wird, von denen keiner eine Einheit ist, wird wie ein Panel-Knoten behandelt.

weitere Einheiten

- Knoten mit der Rolle **layered pane** sind Einheiten.
- Wenn Widgets Sibling-Knoten von Einheiten sind, dann ist der Parent-Knoten eine Einheit.

Glossar

Accessibility Schnittstelle ist eine Schnittstelle, mit der Hilfsmittelprogramme wie Screenreader Informationen zu Ereignissen und dem Aufbau der Oberfläche eines Programmes erhalten.

Accessiblity Software siehe AT-Software.

API – Application Programming Interface; Programmierschnittstelle; definiert eine Programmanbindung an ein Softwaresystem auf Quelltextebene.

AT-Software – Assistive-Technology-Software; Hilfsmittelprogramme, die Menschen mit Einschränkungen das Arbeiten am Computer erleichtern bzw. ermöglichen.

Erkundungsmodus ist eine Arbeitsmethode mit einem Screenreader. Im Erkundungsmodus sondiert der Nutzer die Bildschirminhalte und versucht sich einen Überblick zu verschaffen. In diesem Modus müssen alle grafischen Element auf dem Bildschirm anzeigbar sein, auch Elemente, die den Fokus nicht erhalten können (z. Bsp.: Label) und Elemente die inaktiv, also „ausgegraut“ dargestellt sind.

Flächenmodus ist eine Arbeitsmethode mit einem Screenreader. Im Flächenmodus wird der Bildschirm zeilenweise ausgegeben. Hierbei spielen logische Zusammenhänge zwischen den grafischen Elementen keine Rolle. Dem sehgeschädigten Nutzer soll ein möglichst großer Ausschnitt des Bildschirms angezeigt werden.

GUI – Graphical User Interface, bezeichnet die grafische Benutzeroberfläche.

Navigationsmodus ist eine Arbeitsmethode mit einem Screenreader. Im Navigationsmodus bewegt sich der Anwender nur zwischen den fokussierbaren Elementen auf dem Bildschirm. Alle weiteren angezeigten grafischen Elemente sind in diesem Modus nicht sichtbar.

Screenreader ist ein Programm, welches in die Kategorie der Hilfstechnologien einzuordnen ist. Ein Screenreader liest die Bildschirminhalte aus und bereitet sie neu auf, um sie blinden und sehbehinderten Nutzern über alternative Ausgabemedien zu präsentieren. Die Ausgabe kann zum Beispiel in Form von Sprache oder Braille erfolgen.

Strukturmodus ist eine Arbeitsmethode mit einem Screenreader. Im Strukturmodus wird ein grafisches Element, welches auf dem Bildschirm abgebildet ist, präsentiert. Der Screenreader gibt möglichst viele Informationen zu diesem Element preis, so zum Beispiel Name, Art, Tastaturkürzel, Container-Elemente, Index. Der Schwerpunkt in diesem Arbeitsmodus liegt auf dem Erkennen struktureller Zusammenhänge zwischen den Elementen auf dem Bildschirm.

Widget ist ein grafisches Element in einer GUI.

Sachregister

- 1st Generation Access, 12
- 2nd Generation Access, 12
- 3rd Generation Access, 12

- Accessibility API, 12
- Accessible, 18, 19
- Adjazenzliste, 60
- Adjazenzmatrix, 58
- aktive Einheit, 46
- Alert, 35
- Anwendungsfenster, 30
- AT-SPI, 15
- AT-SPI Baum, 16
- AT-SPI Registry, 16, 17
- ATK, 15

- Baumtransformation, 44
- BrailleDis 9000, 25

- Clipping, 57
- Corba-Objekt, 16, 17

- Datenstruktur, 58
- Dialog, 35

- Erkundungsmodus, 6

- Filler-Knoten, 47
- Flächenmodus, 6, 21

- GAIL, 15
- gewichteter Graph, 57
- Gewichtung, 41
- GNOME Accessibility Projekt, 15
- GNOME Desktop, 29

- heuristischen Analyse, 47
- Hyperbraille, 24

- IAccessible2, 14
- Inhaltsbereich, 33
- inzident, 60

- Java Accessibility API, 12, 14

- logische Einheit, 29, 46

- Menüleiste, 31
- MSAA, 14

- Navigationsmodus, 6

- Off-Screen-Modell, 14, 22, 25
- OSM-Baum, 25

- Panel-Knoten, 47

- Rollen, 47

- Screenreader, 5
- sichtbar, 47
- Spezialfenster, 33
- Split-Pane-Knoten, 47
- Statusleiste, 33
- Strukturmodus, 5

- Tabelle, 37
- Titelleiste, 30
- Toolbox, 34

- Utility Window, 34

- Werkzeugleiste, 32

Tabellenverzeichnis

3.1	Kleine Beispiel-Tabelle	37
3.2	Beispiel-Tabelle mit mehreren Zeilen in einer Zelle	37
3.3	Beispiel-Tabelle vertikal verbundener Zellen	38
3.4	Rollenverteilung für logische Einheiten und Widgets	47
4.1	Performance Messungen ohne Zerlegung des Baumes	72
4.2	Performance Messungen der Baumtransformation	73

Abbildungsverzeichnis

2.1	Linux: GNOME Accessibility Architektur	15
2.2	Schematische Darstellung des AT-SPI Baumes	18
3.1	Schematische Darstellung eines Anwendungsfensters	31
3.2	Schematische Darstellung eines Spezialfensters	34
3.3	Beispiel eines Inhaltsbereiches mit Textfeldern	36
3.4	Anordnung von Widgets mit eindeutiger Zeilenzuordnung	38
3.5	Anordnung von Widgets mit unterschiedlichen Höhen	39
3.6	Anordnung von Widgets über mehrere Zeilen – Beispiel 1	40
3.7	Anordnung von Widgets über mehrere Zeilen – Beispiel 2	41
3.8	Auflösung von Konflikten – Teil 1	43
3.9	Auflösung von Konflikten – Teil 2	43
3.10	Auflösung von Konflikten – Teil 3	44
3.11	Anordnung von Knoten im AT-SPI Baum	48
3.12	Beispiel einer Anordnung für Einheiten	53
3.13	Gewichteter Graph	57
3.14	Clipping – Fallunterscheidungen	58
3.15	Menüzeile	64
3.16	Dialog der Anwendung Open Office.org	65
4.1	Kommunikationswege der Accessibility Architektur	67

Literaturverzeichnis

- [Ott-02] Algorithmen und Datenstrukturen, Thomas Ottmann, Peter Widmayer, 4. Auflage
Spektrum Akademischer Verlag GmbH, Heidelberg, Berlin, 2002
- [Sed-02] Algorithmen, Robert Sedgewick, 2. Auflage
Pearson Education Deutschland, 2002
- [Pla-88] Computergrafik: Einführung – Algorithmen – Programmentwicklung, Jürgen Plate, 2. Auflage
Franzis-Verlag GmbH, München, 1988
- [Tur-09] Algorithmische Graphentheorie, Volker Turau, 3. überarbeitete Auflage
Oldenbourg Wissenschaftsverlage GmbH, München, 2009
- [Hel-06] Touch and Blindness: Psychology and Neuroscience, Morton A. Heller, Soledad Ballesteros
Lawrence Erlbaum Associates, Inc., 2006
- [Mil-06] Processing Spatial Information From Touch and Movement: Implications From and For Neuroscience, Susanna Millar
Touch and Blindness: Psychology and Neuroscience, 2006 (p. 25-48)
- [Hel-06a] Picture Perception and Spatial Cognition in Visually Impaired People, Morton A. Heller
Touch and Blindness: Psychology and Neuroscience, 2006 (p. 49-72)
- [Ken-06] Form, Projection and Pictures for the Blind, John M. Kennedy and Igor Juricevic
Touch and Blindness: Psychology and Neuroscience, 2006 (p. 73-94)
- [Bun-10] Studienarbeit: Erstellen und Umsetzen eines Braille-Ausgabekonzeptes, Ramona Bunk, Februar 2010

- [Koc-94] Designing an OffScreen Model for a GUI, Dirk Kochanek, September 1994,
ICCHP '94: Proceedings of the 4th international conference on Computers for handicapped persons
Publisher: Springer-Verlag New York, Inc.
- [Völ-08] Tactile Graphics Revised: The Novel BrailleDis 9000 Pin-Matrix Device with Multitouch Input
Thorsten Völkel, Gerhard Weber, Ulrich Baumann
International Conference on Computers Helping People with Special Needs (ICCHP) 2008 (p. 865-872)
- [Kra-08] An Off-Screen Model for Tactile Graphical User Interfaces
Michael Kraus, Thorsten Völkel, Gerhard Weber
International Conference on Computers Helping People with Special Needs (ICCHP) 2008 (p. 835-842)
- [Köh-09] What You Feel is What You Get: Mapping GUIs on Planar Tactile Displays
Wiebke Köhlmann, Oliver Nadig, Maria Schiewe, Gerhard Weber
HCI International 2009
- [COB-01] Baum: Produkte und Dienstleistungen für Blinde und Sehbehinderte, COBRA-Handbuch
http://www.baum.de/cms/de-de/cobra_hilfe/ (vom 11.12.2010)
- [JAW-01] JAWS Bedienschulung
<http://www.docstoc.com/docs/51503864/JAWS-Bedienschulung> (vom 11.12.2010)
- [NCS-01] Forscher entwickeln Vollbild-Display für Blinde – Innovation – derStandard.at <http://derstandard.at/1269448617317/Forscher-entwickeln-Vollbild-Display-fuer-Blinde> (vom 18.05.2010)
- [GNO-1] GNOME Human Interface Guidelines 2.2.1
<http://library.gnome.org/devel/hig-book/stable/>
(vom 18.11.2010)
- [GNO-2] GNOME Shell – A design for a personal integrated digital work environment, William Jon McCann, Jeremy Perry
14.November 2009
http://people.gnome.org/~mccann/shell/design/GNOME_Shell-20091114.pdf
- [Kor-1] The AEGIS concept, Peter Korn, Oktober 2010,
Präsentation auf der AEGIS Konferenz, 07.10.-08.10.2010 in Sevilla, Spanien

- [Kor-2] Peter Korn's Weblog (Thursday December 14, 2006)
http://blogs.sun.com/korn/entry/completing_the_accessibility_picture_iaccessible2
(vom 23.11.2010)
- [Tel-03] The Reading Machine That Hasn't Been Built Yet – AccessWorld, March 2003
<http://www.afb.org/afbpress/pub.asp?DocID=aw040204>
- [Tel-05] From Optacon to Oblivion: The Telesensory Story – AccessWorld, July 2005
<http://www.afb.org/afbpress/pub.asp?DocID=aw060403>
- [App-1] Apple Accessibility
<http://www.apple.com/accessibility/> (vom 23.11.2010)
- [App-2] Accessibility Overview: Introduction to Accessibility Overview
<http://developer.apple.com/library/mac/#documentation/Accessibility/Conceptual/AccessibilityMacOSX/OSXAXIntro/OSXAXintro.html> (vom 23.11.2010)
- [MS-1] IAccessible API Documentation
<http://msdn.microsoft.com/en-us/library/ms696165>
(vom 23.11.2010)
- [MS-2] IAccessible2 – The Linux Foundation
<http://www.linuxfoundation.org/collaborate/workgroups/accessibility/iaccessible2> (vom 23.11.2010)
- [MS-3] IAccessible2 API Documentation
<http://accessibility.linuxfoundation.org/a11yspecs/ia2/docs/html/>
(vom 23.11.2010)
- [Jav-1] javax.accessibility (Java 2 Platform SE 5.0)
<http://download.oracle.com/javase/1.5.0/docs/api/javax/accessibility/package-summary.html> (vom 23.11.2010)
- [GAP-1] The GNOME Accessibility Project
<http://web.archive.org/web/20011010200241/-developer.gnome.org/projects/gap/> (vom 03.12.2009)
<http://projects.gnome.org/accessibility/> (vom 03.12.2009)
- [LxF-1] atk/at-spi – The Linux Foundation
<http://www.linuxfoundation.org/en/Accessibility/ATK/AT-SPI>
(vom 03.12.2009)
- [Cor-1] CORBA FAQ
<http://www.omg.org/gettingstarted/corbafaq.htm>
(vom 30.01.2010)

- [Bon-1] GNOME als Entwicklungsplattform
<http://www.oreilly.de/german/freebooks/rlinux3ger/anhang26.html> (vom 30.01.2010)

- [Bon-2] Accessibility/BonoboDeprecation – GNOME Live!
<http://live.gnome.org/Accessibility/BonoboDeprecation>
(vom 05.02.2010)

- [SPI-1] AT-SPI IDL: AT-SPI Interfaces and Subinterfaces
<http://people.gnome.org/~billh/at-spi-idl/html/> (vom 24.11.2010)

- [SPI-2] Pyatspi: API Documentation
<http://people.gnome.org/~parente/pyatspi/doc/> (vom 24.11.2010)