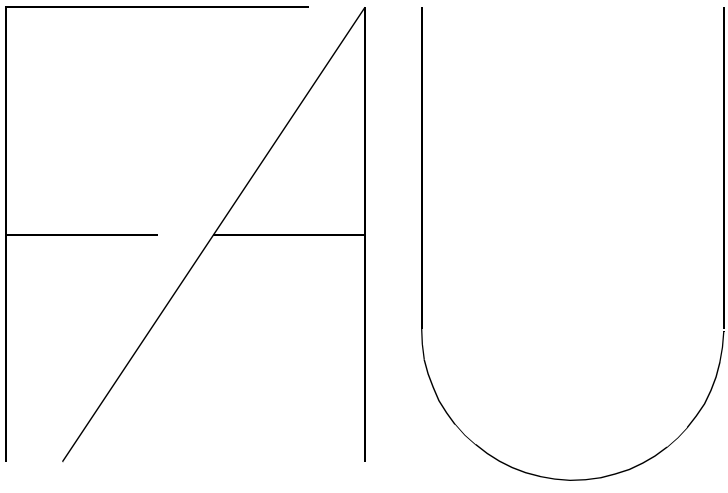
# Entwurf und Implementierung eines Windowmanagers für das Java-Betriebssystem JX

Jürgen Obernolte

Februar 2002 SA-I4-2002-04

Studienarbeit

Institut für Mathematische Maschinen und Datenverarbeitung der Friedrich-Alexander-Universität Erlangen-Nürnberg



# Entwurf und Implementierung

# eines

# Windowmanagers

# für das Java-Betriebssystem JX

## Studienarbeit im Fach Informatik

vorgelegt von

***Jürgen Obernolte***

geb. am 23.12.1974 in Marktredwitz

Angefertigt am

Institut für Mathematische Maschinen und Datenverarbeitung (IV)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 12.03.2002 _____

# Kurzfassung

Betrachtet man sich den heutigen Betriebssystem-Markt so findet man kaum noch ein Betriebssystem, daß nicht in irgendeiner Form mit einer grafischen Benutzeroberfläche ausgestattet ist.

Im Laufe dieser Studienarbeit wurde ein Windowmanager für das Betriebssystem JX in Java entwickelt. Besonderer Wert wurde dabei auf ein klares und übersichtliches Konzept gelegt. Desweiteren wurde versucht, die Schnittstelle zum Windowmanager so einfach wie möglich zu halten, sowie das System erweiterbar zu machen.

Da dem Windowmanager ein Client-Server-Modell zugrunde liegt, konnten hier die speziellen Features des Betriebssystems (Portale, Domains, Sicherheit, Multithreading) genutzt werden. Die Verwendung von Portalen ermöglichte es, auf ein umfangreiches Client-Server-Protokoll zu verzichten, da sich hiermit direkt Fernaufrufe realisieren lassen.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# 1 Einführung

## 1.1 Übersicht über Betriebssysteme mit grafischer Benutzeroberfläche

Bevor die Zielsetzungen und Rahmenbedingungen definiert werden, möchte ich noch einen kurzen Überblick über den heutigen Betriebssystem-Markt geben.

Dominierend ist hier wohl die Windows-Familie der Firma Microsoft. Ausgestattet mit einer durchdachten und einheitlichen Oberfläche erleichtert sie dem Benutzer das Arbeiten mit dem Computer. Das Arbeiten wird durch ein effizientes Hilfesystem (Kontexthilfe) unterstützt. Außerdem erscheint fast jedes Produkt zunächst für Windows, so daß es Programme in Hülle und Fülle gibt.

Weiter wäre das X-Windows System zu nennen, daß für Unix entwickelt wurde. Populär wurde es allerdings erst durch die Verbreitung des Unix-Systems Linux. Die neueste Version ist mittlerweile ziemlich ausgereift und die Optik wird mit jeder neuen Version ein bißchen moderner. Auch für X-Windows-Systeme werden immer mehr Programme entwickelt, so daß es mittlerweile ein ernstzunehmender Gegner für Windows ist.

Ein weitere sehr bekannte Oberfläche, ist die der Apple-Computer. Sie hebt sich von den anderen Oberflächen durch eine sehr lange Tradition ab. Schon die ersten Macintosh-Computer wurden mit einer grafischen Benutzeroberfläche ausgeliefert. Und mit der neuesten Version (AquaX) erreicht die Firma Apple einen neuen de-facto Standard in Sachen Präsentation und Optik.

Als letztes möchte ich noch kurz auf das Betriebssystem BeOS eingehen, da es als Vorlage für den Windowmanager diente. Wer schon einmal Gelegenheit hatte mit dem Betriebssystem BeOS Erfahrungen zu sammeln, der war sicherlich erstaunt über die sehr reaktionsfreudige Oberfläche. Bei den meisten anderen Oberflächen (besonders X-Windows) kann es schon mal vorkommen, daß man kurzfristig nicht weiterarbeiten kann, da gerade ein rechenintensiver Prozeß läuft, der das System lahmlegt. Im Gegensatz dazu ist bei BeOS, selbst bei hoher Arbeitslast, immer noch ein flüssiges Arbeiten möglich. BeOS wurde von Anfang an als Multimedia-System ausgelegt. Ein weiterer großer Vorteil von BeOS ist die strikte Objektorientierung. Die Schnittstelle zum Fenstersystem ist in C++ gehalten, im Gegensatz zu den anderen genannten Oberflächen, die normalerweise nur eine normal C-Schnittstelle besitzen.

## 1.2 Zielsetzung und Rahmenbedingungen

Ziel dieser Studienarbeit war die Entwicklung eines Windowmanagers unter dem Java-Betriebssystem JX. Basis des Betriebssystem ist dabei ein minimaler C-Kern, der maschinennahe Operationen realisiert (Threadhandling, Interruptmanagement, etc...) und Zugriff auf die Hardware gewährt und diese als native-Funktionen zur Verfügung stellt. Die eigentlichen Betriebssystemfunktionen sind dagegen in Java-Klassen implementiert (Scheduler, Dateisystem, ...).

Die Zielsetzung des Betriebssystems liegt darin, so viele Komponenten wie möglich aus dem Kern auszulagern. Erreicht wurde dies dadurch, daß viele Komponenten des Systems als isolierte, austauschbare, Pakete realisiert wurden. Dadurch läßt sich das System sehr leicht skalieren (vom Embedded System bis hin zum Datenbank-Server).

Die Programmiersprache Java bietet viele Vorteile gegenüber anderen Programmiersprachen (z.B. C, C++). Da wären zum einen die leichte Wartbarkeit des Codes zu nennen. Desweiteren bietet die Verwendung der mächtigen Java-Klassenbibliothek von Sun die Möglichkeit, Applikationen schnell und einfach zu entwickeln. Das größte Plus dürfte allerdings die Speicherverwaltung sein, da es keine Zeiger mehr gibt und somit Programmablauffehler, die durch falsche Verwendung von Zeigern entstehen, vermieden werden.

Ein großer Nachteil sollte allerdings auch nicht verschwiegen werden. Java-Programme laufen normalerweise immer langsamer ab als entsprechend übersetzte Programme anderer Programmiersprachen. Dies kommt dadurch zustande, da die Java-Klassen in einen virtuellen Bytecode übersetzt werden. Dieser virtuelle Bytecode wird dann von dem Laufzeitsystem erst in die jeweilige Maschinensprache der zugrunde liegenden CPU übersetzt.

## 1.3 Kapitelübersicht

Die Arbeit gliedert sich grob in zwei Teile. Im ersten Teil werden zuerst die Grundlagen und die Funktionsweise des Windowmanagers beschrieben. Es wird erläutert, welche zentrale Komponenten es gibt und worauf bei der Entwicklung acht gegeben wurde. Es wird außerdem noch mal kurz auf die Eigenheiten des Betriebssystems JX eingegangen.

Der zweite Teil beschreibt die Architektur und die Implementierung des Windowmanagers. Verbunden mit der Implementierung, wird außerdem noch kurz auf die aufgetretenen Schwierigkeiten eingegangen.

## 1.4 Zusammenfassung

Es wurden die Rahmenbedingungen und die Zielsetzung der Arbeit erläutert. Dabei wurde ein kurzer Überblick über bereit existierende Windowmanager gegeben. Desweiteren wurde auf das Betriebssystem JX sowie die Programmiersprache Java eingegangen. Zum Schluß wurde in der Kapitelübersicht ein Überblick über den Aufbau dieser Arbeit gegeben.

# 2 Grundlagen

## 2.1 Übersicht

Im folgenden Kapitel werden zunächst auf die grundlegenden Elemente einer Benutzeroberfläche eingegangen. Hierbei handelt es sich primär um Fenster und ihre Darstellung. Ein weiteres wichtiges Element des Windowmanagers ist die Eingabe des Benutzers, die normalerweise mithilfe von Maus und Tastatur erfolgt. Abschließend wird noch auf das Grafiksystem eingegangen.

## 2.2 Windowmanager vs. Grafische Benutzeroberfläche

Der Begriff des Windowmanagers ist nicht gleichzusetzen mit dem Begriff einer grafischen Benutzeroberfläche. Der Windowmanager ist nur ein Teil der Oberfläche. Unter einer grafischen Benutzeroberfläche versteht man normalerweise den Windowmanager samt zugehöriger Bibliotheken, sowie diverse Hilfsprogramme zum Verwalten der Oberfläche (Desktop, Explorer, etc...).

Der Windowmanager an sich ist nur da, um Fenster zu verwalten. Alle anderen Elemente, wie Buttons, Listviews, Eingabefelder, werden normalerweise durch eine dazugehörige Bibliothek implementiert.

Die diversen Hilfsprogramme wiederum (Desktop, Explorer, ...) sind eigenständige Programme, die auf den Windowmanager aufsetzen.

## 2.3 Das Fensterkonzept

Im JX-Windowmanager besteht ein Fenster aus den Fensterkontrollen und dem Arbeitsbereich, in dem die Benutzerausgaben erfolgen. Der Arbeitsbereich hat sein eigenes Koordinatensystem. Abb. 2.1 zeigt ein Standardfenster des Windowmanagers.

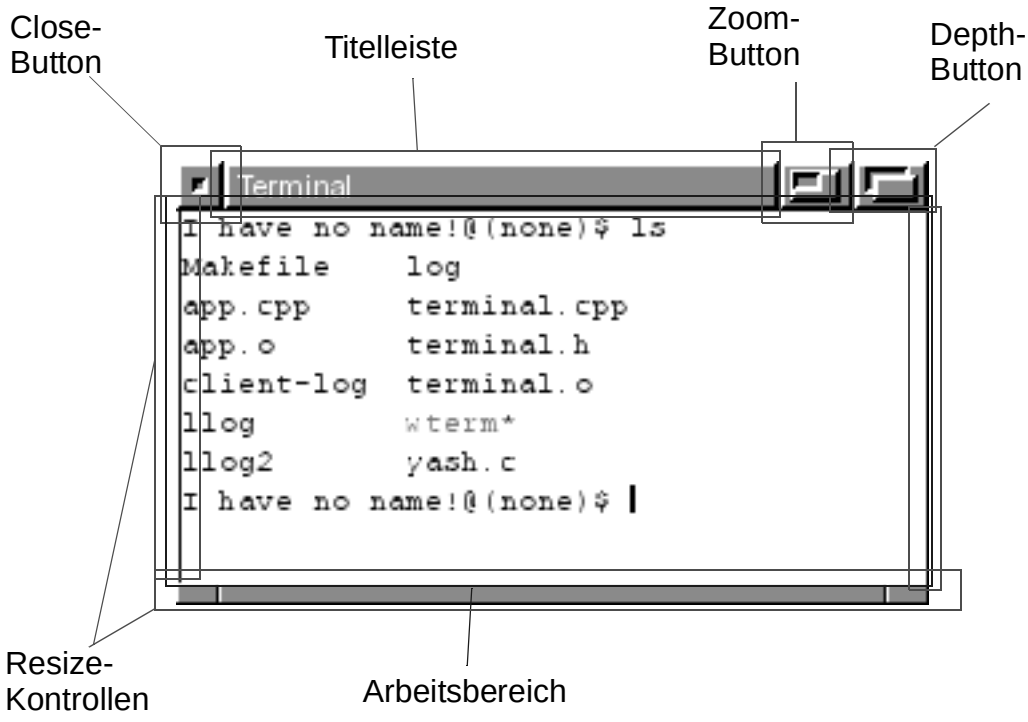



**Abb. 2.1 Ein typisches Fenster**

Der Close-Button dient dazu, das Fenster zu schließen. Die Titelleiste zeigt den Namen des Fensters und dient außerdem dazu, daß Fenster auf dem Bildschirm zu verschieben. Mit dem Zoom-Button wird die ursprüngliche Größe und Position wiederhergestellt. Der Depth-Button bringt das Fenster entweder in den Vorder- oder in den Hintergrund. Die Resizekontrollen dienen dazu, die Größe des Fensters zu verändern.

## 2.4 Das Konzept der Views

Ein Fenster des Windowmanagers ist in kleinere rechteckige Bereiche aufgeteilt, den sogenannten Views. Jede View entspricht einem Teil von dem, was das Fenster anzeigt. Dabei kann es sich z.B. um einen Scrollbar, ein Dokument oder ähnliches handeln. Momentan existieren pro Fenster nur zwei Views. Die eine View repräsentiert dabei die Fensterkontrollen, die andere View stellt den Arbeitsbereich des Fensters dar.

Die Views sind hierarchisch organisiert, d.h. jede View kann eine beliebige Anzahl an Childviews enthalten. Bei einem Fenster zum Beispiel, ist der Arbeitsbereich eine Childview der Fensterkontrollen.

Der Bildschirm selbst wird dabei auch durch eine View (der Topview) repräsentiert. Wird ein Fenster erzeugt und angezeigt, so sind die Views dieses Fensters Childviews der Topview.

Die View-Hierarchie läßt sich als verzweigender Baum darstellen, wobei die Topview die Wurzel darstellt. Jede View in der Hierarchie (außer der Topview) hat eine einzige Parentview. Jede View kann eine beliebige Anzahl an Childviews haben.

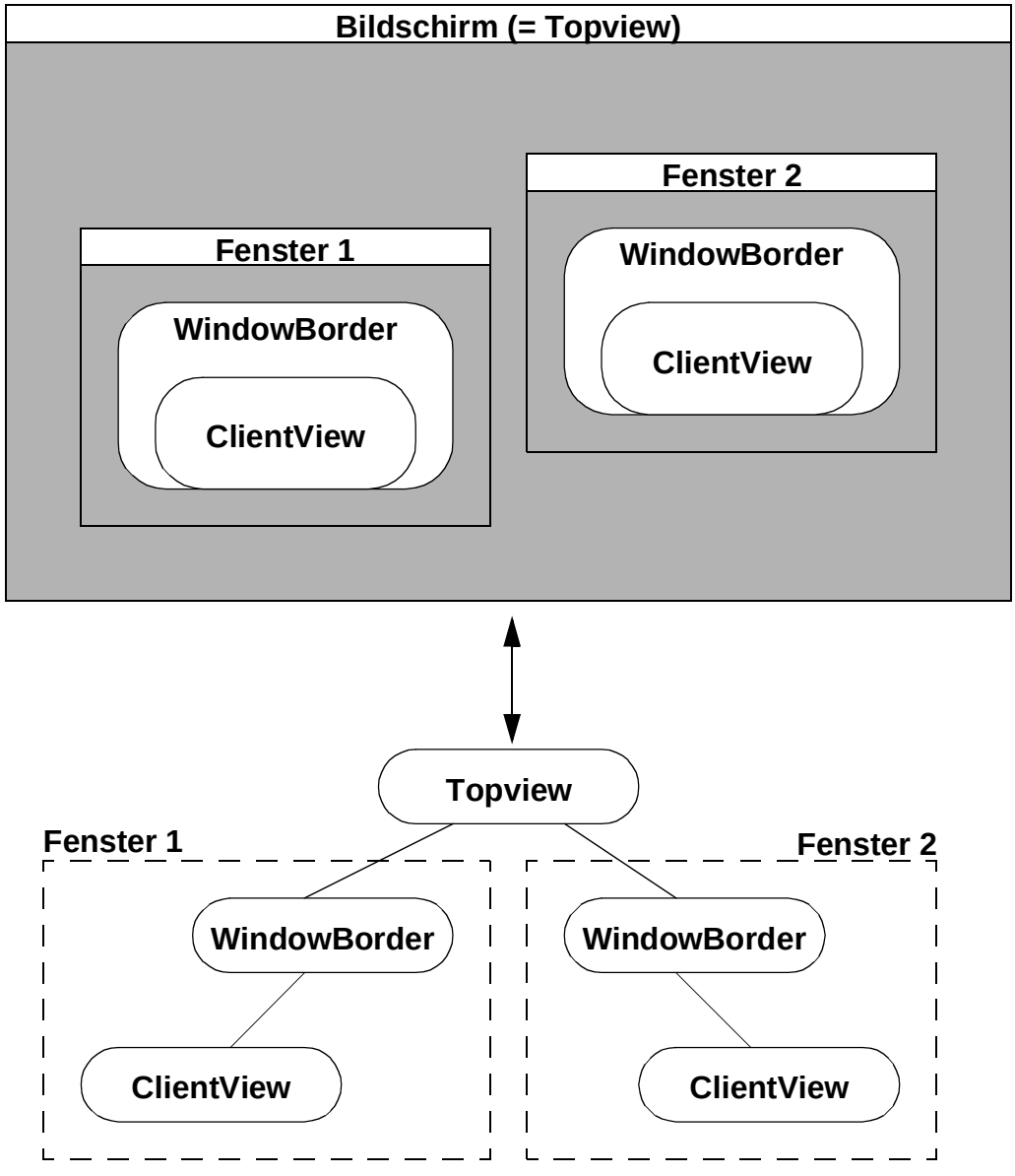



**Abb. 2.2 Die View-Hierarchie**

Abb. 2.2 zeigt den Baum für zwei Fenster.

Momentan existiert noch keine Unterstützung für das Erzeugen von Childviews seitens der Applikation. Eine Implementierung ist jedoch leicht möglich.

### 2.4.1 Die View-Geometrie

Jede View hat ihr eigenes Koordinatensystem. Der Ursprung des Koordinatensystem liegt dabei in der linken, oberen Ecke der View. Alle Ausgaben erfolgen relativ zum Ursprung. Abb. 2.3 zeigt den Aufbau des Koordinatensystems.

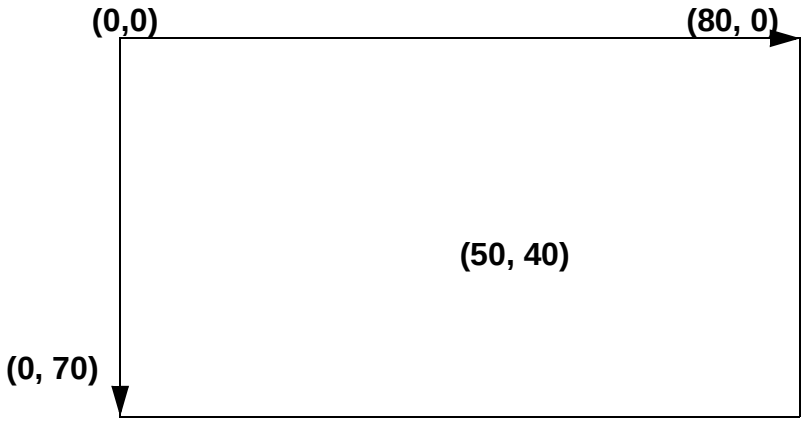



**Abb. 2.3 Koordinatensystem einer View**

Die Einheiten des Koordinatenssystems sind Pixel. D.h. der Punkt mit den Koordinaten (0, 0) entspricht dem Pixel an eben dieser Position.

### 2.4.2 Frame- und Bounds-Rechtecke

Da jede View (außer der Topview) innerhalb des Koordinatensystems der Parentview liegt, werden die Grenzen der View initial in Koordinaten der Parentview angegeben. Das Rechteck, das im Koordinatensystem der Parentview liegt, wird Frame-Rechteck genannt. Die internen Koordinaten des Rechtecks stellen das Bounds-Rechteck dar.

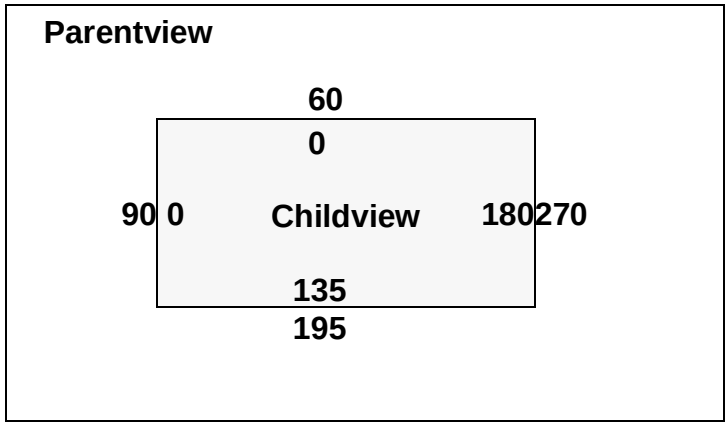



**Abb. 2.4 Frame und Bounds einer View**

Abb. 2.4 zeigt eine Childview, die 180 Einheiten breit und 135 Einheiten hoch ist. Betrachtet man die Childview aus der Perspektive der Parentview, so hat die Childview ein Frame-Rechteck mit der linken, oberen, rechten, unteren Koordinate bei 90, 60, 270, 195. Betrachtet man das selbe Rechteck aus der Perspektive der Childview, so hat das Rechteck die Koordinaten 0, 0, 180, 135.

Wenn eine View an eine neue Position verschoben wird, so ändern sich die Koordinaten des Frame-Rechtecks, nicht aber die Koordinaten des Bounds-Rechtecks. Wird der Inhalt einer View gescrollt, dann ändern sich die Koordinaten des Bounds-Rechtecks, nicht aber die des Frame-Rechtecks.

### 2.4.3 Scrolling

Der Inhalt einer View wird gescrollt, indem die Koordinatenwerte innerhalb des View-Rechtecks verschoben werden. Mit anderen Worten, die Koordinaten des Bounds-Rechtecks der View werden verändert.

Ein kleines Beispiel: Angenommen, das Bounds-Rechteck hat die Koordinaten [(0,100) / (100, 200)]. Scrollt man jetzt um 50 Einheiten, so schauen die Koordinaten folgendermaßen aus: [(0, 150) / (100, 250)]. Abb. 2.5 zeigt das ganze anschaulich.

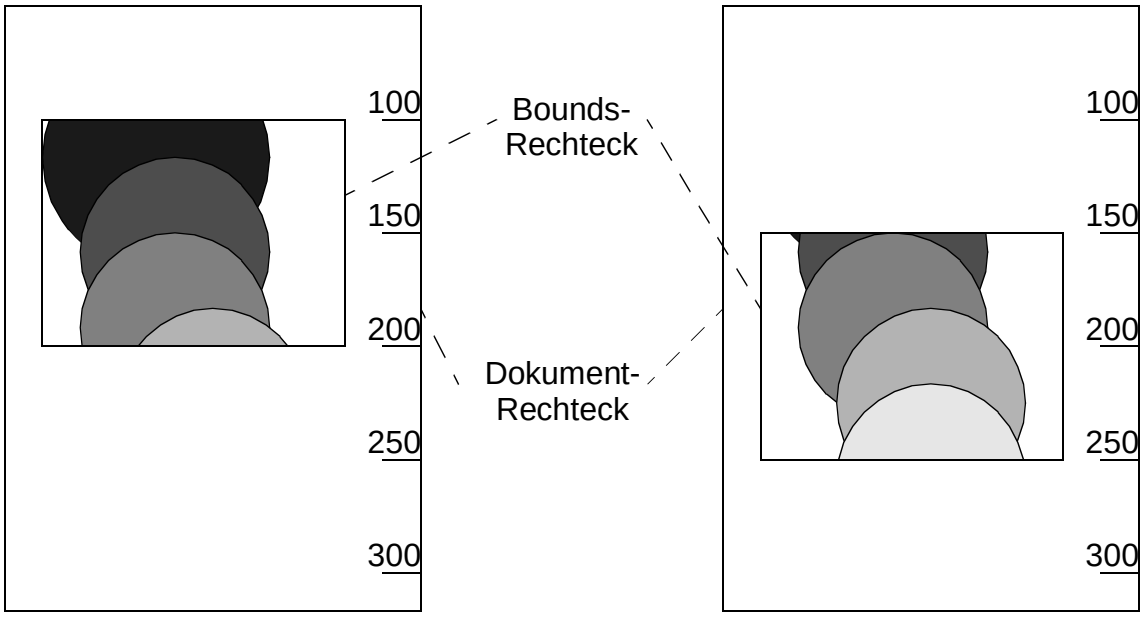



**Abb. 2.5 Scrollen einer View**

Das Scrollen einer View bewegt weder den Frame noch verändert es die Größe des Frames. Es bewegt einzig den Inhalt der View. Im obigen Beispiel umfaßt das Dokument-Rechteck alles, was von der View gezeichnet werden kann. Wenn z.B. die View dafür zuständig ist, ein Buch anzuzeigen, dann wäre das Dokument-Rechteck groß genug, um alle Zeilen und Seiten des zu umfassen. Da jedoch eine View nur innerhalb ihres Bounds-Rechtecks zeichnen kann, ist alles was außerhalb liegt unsichtbar. Damit man auch die unsichtbaren Teile des Buches zu sehen bekommt, muß man die Koordinaten des Bounds-Rechtecks verändern.

### 2.4.4 Sichtbarer Bereich und Zeichenbereich

Normalerweise ist die ganze Fläche einer View der Bereich, in dem gezeichnet werden darf. Es kann niemals außerhalb der Grenzen einer View gezeichnet werden. Da die Views jedoch hierarchisch angeordnet sind, kann es natürlich vorkommen, daß sich zwei Views überlappen. Der sichtbare Bereich einer View besteht aus den Bereich, der durch das Bounds-Rechteck repräsentiert wird. Davon werden die Rechtecke aller überlappenden Childviews abgezogen. Desweiteren kann eine View auch niemals außerhalb der Grenzen der Parentview oder den Grenzen irgendeines Vorgängers zeichnen.

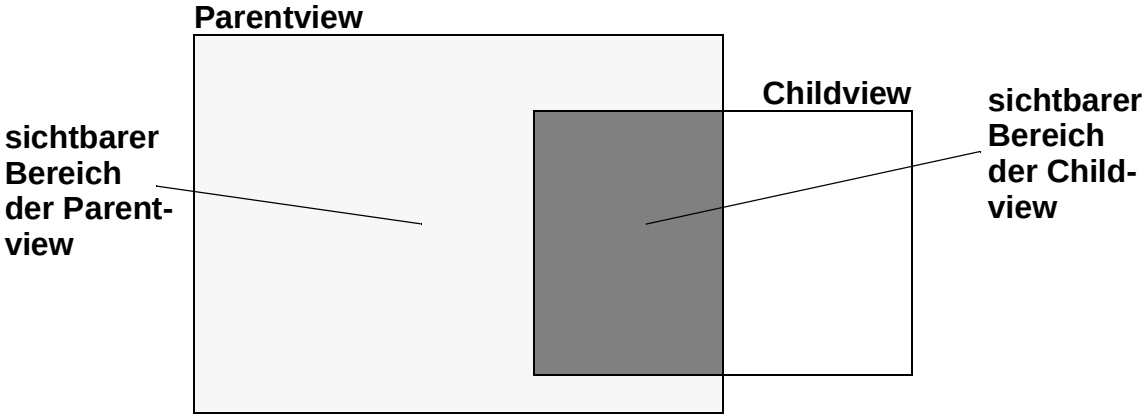



**Abb. 2.6 Sichtbarer Bereich einer View**

Abb. 2.6 zeigt ein Beispiel für den sichtbaren Bereich einer View. Der sichtbare Bereich der Parentview ist dabei der hell schraffierte Teil. Der sichtbare Bereich der Childview ist der dunkel schraffierte Teil. Außerdem sieht man, daß ein Teil der Childview außerhalb des Frames der Parentview liegt.

Weiterhin kann der sichtbare Bereich eines Fensters beschnitten werden, wenn das Fenster, zu dem die View gehört, von einem anderen Fenster verdeckt wird. Abb. 2.7 zeigt das selbe Beispiel wie in Abb. 2.6, diesmal ist jedoch die Parentview teilweise von einem anderen Fenster verdeckt.

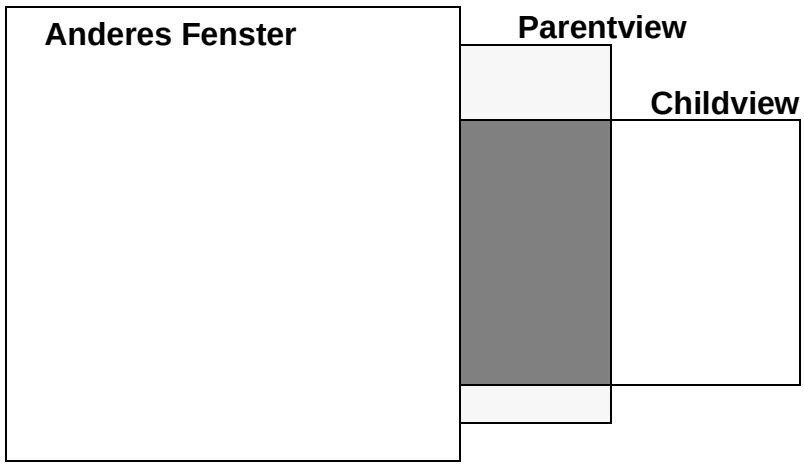



**Abb. 2.7 Sichtbarer Bereich einer View verdeckt von einem Fenster**

Alle Zeichenausgaben einer View erfolgen nur im sichtbaren Teil der View. Normalerweise ist der Zeichenbereich identisch mit dem sichtbaren Bereich. Es gibt jedoch Fälle, in denen Zeichenbereich vom sichtbaren Bereich abweicht.

- Muß ein Teil einer View neu gezeichnet werden, z.B. weil ein verdeckendes Fenster verschoben wurde, dann besteht der Zeichenbereich nur aus dem Bereich, der neu gezeichnet werden muß.
- Wird der Inhalt einer View gescrollt, so versucht der Windowmanager so viel wie möglich vom Inhalt der View zu verschieben. Der Zeichenbereich besteht dann aus den Bereichen, die der Windowmanager nicht verschieben konnte.

## 2.5 Behandlung von Maus- und Tastaturereignissen

Bewegt der Benutzer die Maus, oder drückt eine Taste, so wird das dem Windowmanager mitgeteilt. Der Windowmanager findet dann heraus, für welches Fenster die Eingabe gedacht ist und schickt eine Nachricht an das entsprechende Fenster. Die Applikation kann dann entsprechend darauf reagieren.

Ist das Ziel der Eingabe allerdings die View der Fensterkontrollen, so werden die Nachrichten innerhalb des Windowmanagers verarbeitet und keine Nachricht an die Applikation geschickt. Je nachdem, welche Kontrolle betätigt wurde, reagiert der Windowmanager unterschiedlich. Drückt z.B. der Benutzer mit der Maus auf die Titelleiste und verschiebt dann die Maus, so wird das Fenster bewegt.

## 2.6 Das Grafiksystem

### 2.6.1 Das Koordinatensystem des Bildschirms bzw. einer Bitmap

Im Windowmanager wird der Bildschirm bzw. eine Bitmap durch ein zwei-dimensionales Koordinatensystem beschrieben. Abb. 2.8 zeigt den Aufbau des Koordinatensystems.

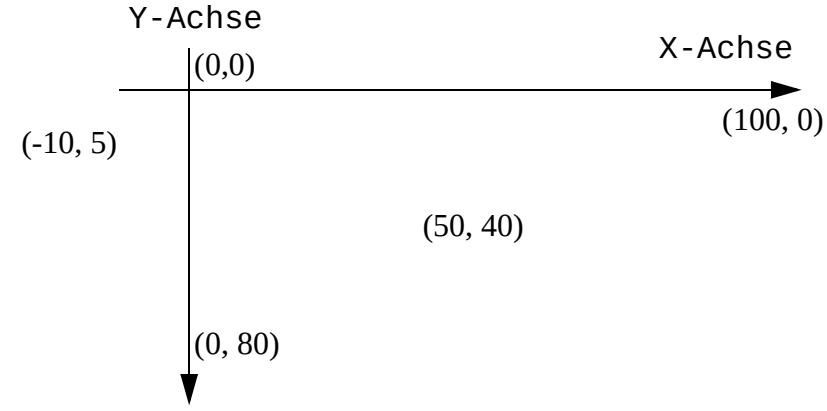



**Abb. 2.8 Das Koordinatensystem**

Die Einheiten des Koordinatenssystems sind Pixel, d.h. der Punkt mit den Koordinaten (50,40) entspricht dem Pixel an dieser Position.

### 2.6.2 Farbdarstellung

Farben im Windowmanager werden durch das RGBA-Modell spezifiziert. RGBA heißt, das jede Farbe aus vier Komponenten besteht, nämlich Rot, Grün, Blau und einem Alpha-Wert. Der Alpha-Wert gibt an, wie ’durchsichtig’ die Farbe ist. Jede dieser Komponenten wird als 8-Bit Wert abgespeichert und hat einen Wertebereich von 0...255.

### 2.6.3 Farbräume

Um einen Pixel in einer bestimmten Farbe auf dem Bildschirm darstellen zu können, muß man letztendlich die Farbtiefe des Bildschirms bzw. der Grafikkarte wissen. Gängige Farbtiefen sind hierbei

- 8 Bit Indexed
  Der Pixelwert stellt einen Index in eine Farbtabelle dar. Die Farbtabelle wiederum, spezifiziert die eigentlichen Farben.
- 15/16 Bit RGB
  Eine Farbe wird als RGB-Wert abgespeichert, wobei die Rot-, Grün- und Blauanteile jeweils 5 Bit breit sind. Bei 16 Bit RGB ist der Grün-Anteil 6 Bit breit.
- 32 Bit RGB
  Eine Farbe wird wiederum als RGB-Wert abgespeichert, wobei Rot, Grün und Blau jeweils 8 Bit belegen. Die verbliebenen 8 Bit sind ungenutzt.

Da der Windowmanager das RGBA-Modell als Farbmodell hat, müssen die Farben beim Zeichnen immer entsprechend umgewandelt werden. Dazu gibt es zwei Möglichkeiten:

1) Man spezifiziert explizit die Bitbreite des Rot-, Grün- und Blauanteils mitsamt der zugehörigen Bitposition, oder
2) man bestimmt vordefinierte Farbmodelle (hier Farbräume genannt), die die Bitbreite und Bitposition der einzelnen Komponenten wiedergeben.

Die erste Möglichkeit ist zwar flexibler, da beliebige Farbmodelle unterstützt werden können, hat jedoch den entscheidenden Nachteil, das die Umwandlung einer Farbe mehr Rechenzeit in Anspruch nimmt, als bei der zweiten Möglichkeit. Beim Zeichnen von Linien oder Rechtecken hat das zwar keine großen Auswirkungen, da man nur eine einzige Farbe hat. Möchte man jedoch eine Bitmap zeichnen, die eine andere Farbtiefe als der Bildschirm besitzt, dann summiert sich diese zusätzliche Rechenzeit sehr schnell auf.

Der Windowmanager benutzt die zweite Möglichkeit zur Darstellung der Farbmodelle.

### 2.6.4 Zeichenmodi

Der Zeichenmodus bestimmt, auf welche Art und Weise die Pixel des Hintergrunds mit den zu setzenden Pixel kombiniert werden. Die einfachste Art des Zeichnens, besteht darin, den Hintergrundpixel einfach mit dem zu zeichnenden Pixel zu ersetzen. Um den Entwickler beim Zeichnen zu unterstützen, existieren noch mehr Möglichkeiten, wie Vordergrund und Hintergrund miteinander kombiniert werden. Folgende Liste gibt einen kurzen Überblick über die Kombinationen:

- Ersetzen des Hintergrundpixels mit dem zu zeichnenden Pixel
- Invertieren des Hintergrundpixels.
- Bedingtes Setzen des zu zeichnenden Pixels. Diese Möglichkeit wird nur beim Zeichnen von Bitmaps verwendet. Bedingtes Setzen heißt, daß nur die Pixel der Bitmap gezeichnet werden, die nicht einem vordefinierten Farbwert entsprechen.

### 2.6.5 Zeichenoperationen

Das A und O einer Benutzeroberfläche ist natürlich die Ausgabe auf dem Bildschirm. Damit man überhaupt etwas sieht, muß das System eine gewisse Anzahl an Funktionen bereitstellen, mit denen sich etwas darstellen läßt. Zu solchen Funktionen gehören z.B. Zeichnen von Linien oder Rechtecken.

### 2.6.6 Schriften

Um Text auf dem Bildschirm anzeigen zu können, muß man zum einen die Schriften in geeigneter Weise verwalten, und zum anderen muß man Funktionen zur Verfügung stellen, mit denen sich dann der Text auch ausgeben läßt. Heutzutage existiert eine Vielzahl an Schriftarten-Formaten und ebenso viele Tools bzw. Bibliotheken, um diese zu verwalten oder anzuzeigen. Jedoch sind sie alle ausnahmslos nicht in Java, sondern meist in C, oder C++ geschrieben. Da die Portierung jedoch zu viel Zeit verschlungen hätte, und eine Neu-Entwicklung nicht zur Debatte stand, existiert momentan nur eine Unterstützung für einen einzigen Bitmapfont.

### 2.6.7 Bitmaps

Um die ganzen Zeichenfunktionen möglichst einfach und portabel zu handhaben, und nicht nur auf die Grafikkarte beschränkt zu bleiben, wurde das Konzept der Bitmap eingeführt. Bei einer Bitmap im Windowmanager handelt es sich nicht um eine herkömmliche Bitmap, die nur Bilddaten aufnimmt. Vielmehr repräsentiert eine Bitmap ein bestimmtes Gerät (normalerweise Bildschirm/Grafikkarte), daß die verschiedenen Grafikfunktionen implementiert. Dabei kann es sich allerdings auch um ein normales Hauptspeicher-Objekt handeln, so daß dann die Bitmap eine ganz normale Bitmap darstellt.

Das Konzept wurde eingeführt, um alles so weit wie möglich zu abstrahieren. Für den Windowmanager ist es dann egal, ob es sich bei der Bitmap um eine Grafikkarte handelt, oder vielleicht um eine Netzwerkverbindung, die die Grafikoperationen, einfach übers Netzwerk schickt.

Momentan stellt die Schnittstelle zur Bitmap folgende Funktionen zur Verfügung:

- Zeichnen von Linien
- Zeichnen von Rechtecken
- Zeichnen von Bitmaps
- Ausgabe von Text
- Kopieren eines rechteckigen Bereichs

## 2.7 Zusammenfassung

Dieses Kapitel hat zunächst kurz den Unterschied zwischen einem WIndowmanager und einer grafischen Benutzeroberfläche erläutert. Im Anschluß wurde auf das Fensterkonzept eingegangen, sowie auf das Konzept der Sichten (Views). Es wurde erklärt, wie Maus- und Tastaturereignisse behandelt werden. Zum Schluß wurde auf das zugrunde liegende Grafiksystem eingegangen.

# 3 JX Spezifische Elemente

## 3.1 Überblick

Im folgenden Kapitel möchte ich noch kurz einige Eigenheiten des Betriebssystems JX erläutern, die für das Verständnis der nachfolgenden Kapitel unerläßlich sind. Dabei geht es vor allem um Betriebssystem-Spezifische Elemente, wie Prozesse, IPC, Multithreading, etc..., die unter Java nur wenige oder gar keine Unterstützung haben.

## 3.2 Prozesse vs. Domains

Unter einem Prozess versteht man normalerweise einen eigenständigen, von anderen Prozessen unabhängigen, Ablauf eines Programms. Jeder Prozess hat dabei seine eigenen Ressourcen, wie Dateideskriptoren, Speicherverwaltung, usw..

JX stellt ein ähnliches Konzept zur Verfügung, das Konzept der Domain. Unter einer Domain versteht man eine grundlegende Einheit der Protektion und Ressourcen-Verwaltung. Jede Domain (außer DomainZero) enthält zu 100% Java Code.

Jede Domain besitzt ihren eigenen Heap und Garbage Collector. Die Garbage Collectoren laufen unabhängig voneinander und können unterschiedliche Algorithmen benutzen.

Jede Domain besitzt auch ihren eigenen Scheduler. Je nach Wahl des Schedulers, läßt sich damit z.B. zwischen kooperativen oder preemptiven Multitasking wählen.

## 3.3 RMI, IPC, RPC und Portale

Kein Client-Server-Modell kommt ohne Interprozesskommunikation (IPC) aus. Programmiert man nicht in Java, dann muß man normalerweise ein geeignetes Kommunikations-Protokoll entwerfen und verifizieren oder testen. Darauf aufbauend kann man dann die Fernaufrufe entwickeln, die normalerweise Nachrichtenbasiert sind.

Unter Java fällt dieser Entwicklungsschritt weg, da es von Haus aus entsprechende Mechanismen zur Verfügung stellt. Die Rede ist hier von Remote Method Invocation (RMI), das nichts anderes als einen Fernaufruf darstellt. Der Entwickler muß sich dabei nicht mehr um die Interprozesskommunikation kümmern.

Unter JX tritt anstelle der Remote Method Invocation, das Konzept der Portale. Portale sind die grundlegenden Inter-Domain Kommunikations Mechanismen. Dienste, die die eine Domain zur Verfügung stellt, können von einer anderen Domain genutzt werden. Ein Dienst besteht dabei, aus einem normalen Objekt, daß ein sogenanntes *Portal Interface* implementieren muß, und mit einen dazugehörigen *Service-Thread* verbunden ist. Der Dienst wird dann über ein sogenanntes *Portal* angesprochen.

Wenn ein Thread eine Methode an einem Portal aufruft, dann wird der Thread blokkiert, und die Ausführung wird vom Service-Thread fortgeführt. Alle Parameter werden zur Ziel-Domain kopiert. Hat der Service-Thread seine Aufgabe ausgeführt, so wird der wartende Thread deblockiert, und kann weiterlaufen.

Ein Problem beim Aufrufen von solchen Methoden, ist das Kopieren der Parameter. Normalerweise führt das dazu, das Daten dupliziert werden, was besonders dann problematisch ist, wenn es sich um eine große Anzahl an Daten handelt. JX hat jedoch interne Mechanismen, die dieses Problem umgehen.

## 3.4 Multithreading

Verwendet man mehrere Threads innerhalb einer Domain, so muß man sich unweigerlich um das Problem der Synchronisation und Schutz einzelner Objekte vor multiplen Zugriffen Gedanken machen. Java bietet dazu die Möglichkeit, Klassen mit dem Attribut `synchronized` zu versehen, um einzelne Objekte vor multiplen Zugriffen zu schützen. Zum Zeitpunkt der Entwicklung des Windowmanagers, stand diese Java-Konstrukt allerdings noch nicht zur Verfügung. Das Problem wurde dadurch umgangen, daß kein preemptiver Scheduler verwendet wurde, sondern ein kooperativer Scheduler. Kooperativ heißt, daß ein Thread solange Rechenzeit in Anspruch nimmt, bis er die Kontrolle wieder abgibt. Die Abgabe der Kontrolle erfolgt hierbei explizit, via `Thread.yield()`, oder implizit, in dem der Thread bei einem Methodenaufruf blokkiert wird.

## 3.5 Memory Objekte

Um größere Speicherbereiche anzusprechen, bedient sich Java der Hilfe von Byte-Arrays, die allerdings viele Nachteile haben. Unter JX existieren für das Handling von großen Speicherblöcken sogenannte Memory Objekte. Memory Objekte werden, wie andere Objekte auch, über normale Methodenaufrufe angesprochen. Viel wichtiger als das, ist jedoch, daß sie bei Inter-Domain-Aufrufen als Portale übergeben werden. D.h. der Speicher wird bei der Parameterübergabe nicht kopiert, sondern nur als Referenz übergeben.

## 3.6 Zusammenfassung

Dieses Kapitel gab einen kurzen Einblick in JX-Spezifischen Eigenheiten. Im wesentlichen erweitert JX die Sprache Java noch um Betriebssystem-Spezifische Klassen und Methoden.

# 4 Architektur

## 4.1 Überblick

Im folgenden Kapitel wird die Architektur des Windowmanagers näher erläutert. Es wird dabei auf das Client-Server-Modell eingegangen, daß dem Windowmanager zugrunde liegt. Desweiteren wird kurz der Programmfluß zwischen Client und Server skizziert, sowie auf das Doppelleben eines Fenster auf der Client- und Server-Seite eingegangen.

## 4.2 Client-Server-Architektur

Da es normalerweise nicht nur eine Applikation gibt, die Fenster erzeugt, macht es wenig Sinn, den Windowmanager nur als eine Klassenbibliothekt anzulegen. Vielmehr müssen die Fenster zentral verwaltet werden. Daher bietet sich die Client-Server-Architektur an, wobei der Windowmanager der Server ist und die Applikationen die Clients.

Beim entwerfen einer Client-Server-Architektur muß man verschiedene Aspekte in Betracht ziehen. Dazu gehören z.B. die geschickte Wahl der Interprozesskommunikation, damit es nicht zu ungewollten Verzögerungen kommt. JX stellt doch jedoch schon von Haus auf entsprechende Mechanismen zur Verfügung, so daß man sich darüber nur noch wenig Gedanken machen muß.

Ein weiterer wichtiger Aspekt ist das Thema Multithreading. Da es normalerweise viele Clienten gibt, die mit dem Windowmanager kommunizieren, würde es durchaus Sinn machen, für jedes Fenster einen eigenen Thread zu starten. Auch hier wird man wiederum von JX unterstützt. Da Fenster selbst als Portale implementiert sind, wird für jedes erzeugte Fenster auch automatisch ein Service-Thread gestartet, der ankommende Methodenaufrufe entgegennimmt (siehe Kapitel 3.3 auf Seite 17).

## 4.3 Programmfluß zwischen Client und Server

Abb. 4.1 zeigt den Programmfluß zwischen dem Windowmanager (= Server) und einer Applikation (= Client).

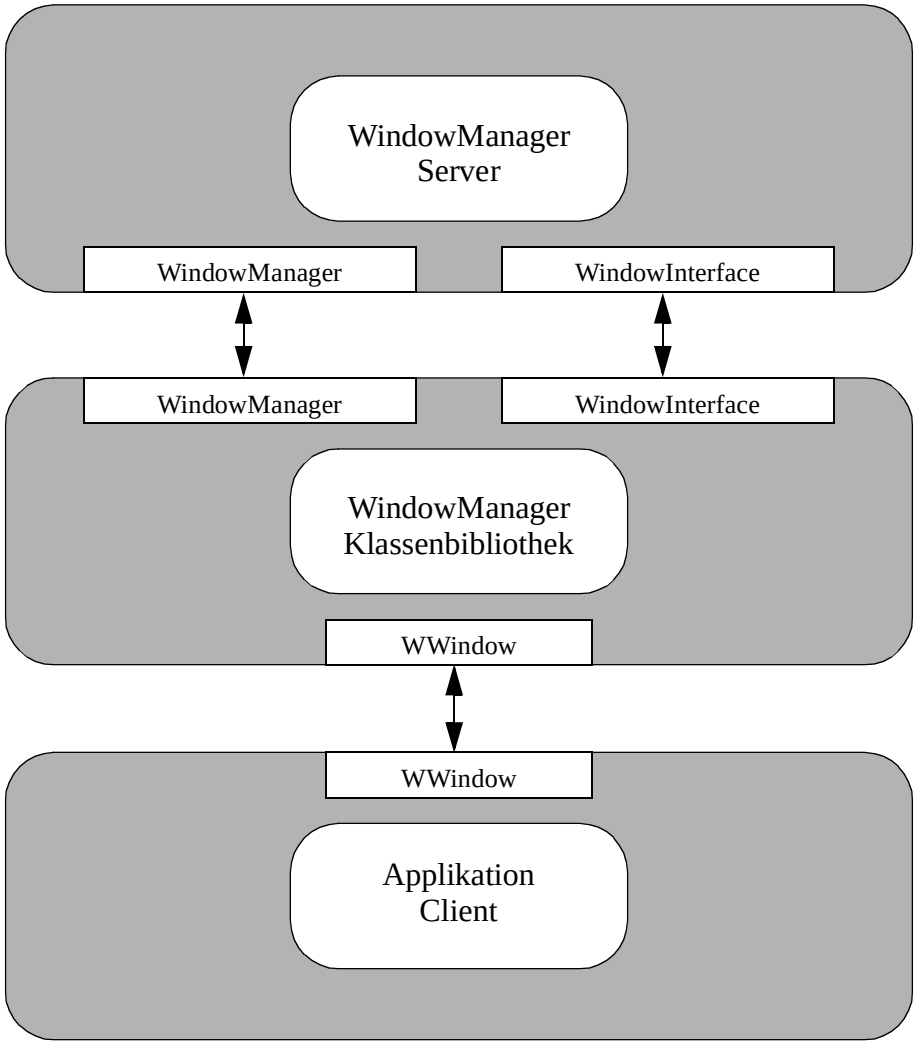



**Abb. 4.1 Client-Server-Architektur**

Der Windowmanager ist als Server implementiert und läuft als eigenständiger Prozess in einer eigenen Domain. Möchte eine Applikation ein Fenster erzeugen, so bedient sie sich der Klasse `WWindow`. Die Klasse `WWindow` benutzt das Portal `WindowManager` um ein Fenster zu erzeugen. Das Fenster wird dann vom Windowmanager erzeugt und als Objektreferenz zurückgegeben. Da dabei eine Domaingrenze überschritten wird (Client und Server laufen in verschiedenen Domänen), wird das zurückgegebene Objekt in ein Portal umgewandelt.

## 4.4 Die Server-Seite eines Fensters

Ein Fenster auf der Server-Seite wird durch ein Objekt der Klasse `WWindowImpl` repräsentiert. Diese Klasse implementiert dabei das Interface `WWindowInterface`, das von der Client-Seite benutzt wird, um mit dem Windowmanager / dem Fenster zu kommunizieren. Abb. 4.2 zeigt die einzelnen Komponenten eines Fensters.

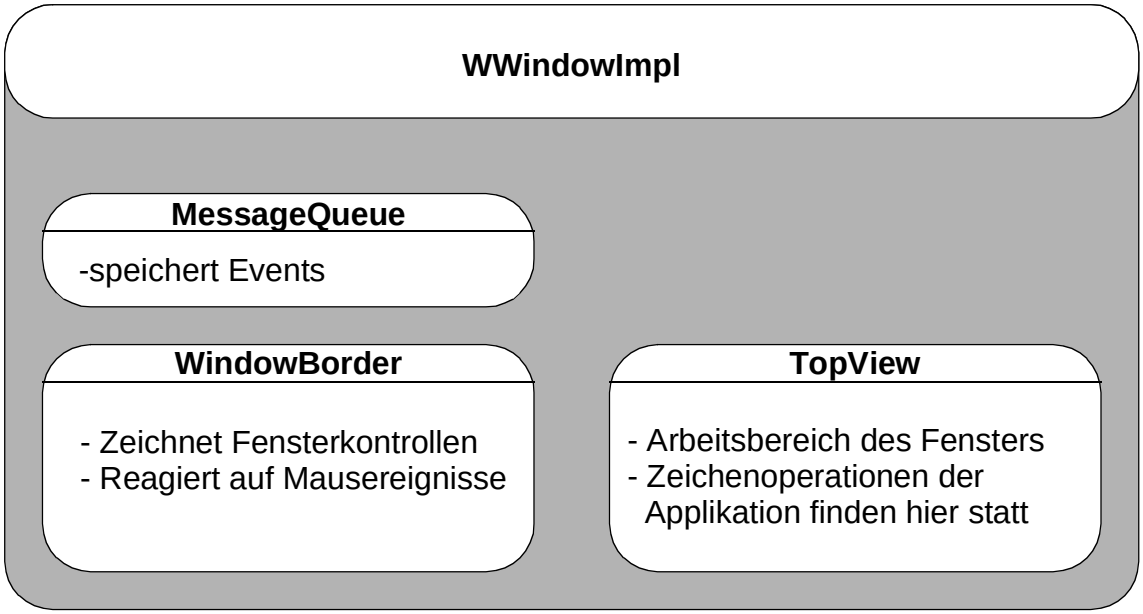



**Abb. 4.2 Komponenten der Klasse WWindowImpl**

Ein Fenster besteht im wesentlichen aus drei Komponenten:

a) **MessageQueue**
In der Nachrichtenschlange werden auftretende Events, die das Fenster betreffen, in der Reihenfolge ihres Eintreffens eingereiht. Momentan werden folgende Events verarbeitet:

- Paint-Event: Wird dann gesendet, wenn Teile des Fensters neu gezeichnet werden müssen
- Key-Down-Event: Wird gesendet, wenn eine Taste gedrückt wurde
- Key-Up-Event: Wird gesendet, wenn eine Taste losgelassen wurde
- Mouse-Down-Event: Ein Mausbutton wurde gedrückt
- Mouse-Up-Event: Ein Mausbutton wurde losgelassen
- Mouse-Move-Event: Die Maus wurde bewegt (innerhalb des Fensters)

- Window-Activated-Event: Wird dann geschickt, wenn das Fenster zum aktiven Fenster gemacht wurde (= Fenster hat den Eingabefokus bekommen)
- Window-Frame-Changed-Event: Wird dann geschickt, wenn die Größe oder die Position des Fenster verändert wurde

b) **WindowBorder**
ist ein Objekt, dessen Klasse von der Klasse `WView` abgeleitet ist. Die Klasse `WindowBorder` ist im wesentlichen für die Erzeugung, das Zeichnen und Verwalten von Events für die Fensterkontrollen zuständig. Alle Mausereignisse, die im Bereich der Fensterkontrollen stattfinden, werden an dieses Objekt weitergeleitet.

c) **TopView**
ist ein Objekt der Klasse `WView`. Dieses Objekt stellt den eigentlichen Arbeitsbereich des Fensters dar. Alle Zeichenoperationen der Applikation werden im Arbeitsbereich ausgeführt.
Desweiteren werden alle Ereignisse, die den Arbeitsbereich betreffen, als Nachrichten verpackt und in die Nachrichtenschlange eingereiht.

## 4.5 Die Client-Seite eines Fensters

Möchte eine Applikation ein Fenster erzeugen, so leitet sie einfach die Klasse `WWindow` ab. Die Klasse `WWindow` befindet sich in der Klassenbibliothek für den Windowmanager. Sie ist dazu gedacht, die ganzen Implementierungsdetails vor dem Applikationsentwickler zu verbergen, und dem Entwickler eine möglichst einfache Schnittstelle an die Hand zu geben.

Der nächste Schritt besteht dann darin, einfach ein Objekt der abgeleiteten Klasse zu erzeugen. Im Konstruktor der Klasse `WWindow` wird nach der Domäne “WindowManager” gesucht und eine Referenz darauf gespeichert. Über einen Portalaufruf der Domäne “WindowManager” wird dann das Fenster (als Objekt der Klasse `WWindowImpl`) erzeugt und das Objekt rausgegeben. War die Erzeugung des Fensters erfolgreich, so wird im Anschluß durch den Konstruktor noch ein Looperthread gestartet.

Der Looperthread der Klasse `WWindow` ruft in einer Endlosschleife die Funktion `peekMessage()` auf. Befindet sich eine Nachricht in der Nachrichtenschlange, so wird diese entnommen und ausgewertet. Je nachdem, um welches Event es sich handelt, wird eine entsprechende Methode aufgerufen.

Der erste Schritt einer Applikation besteht darin, daß sie sich erstmal eine von WWIndow abgeleitete Klasse erstellt. Möchte die Applikation nun auf ein Ereignis reagieren, so überschreibt sie einfach die entsprechende Funktion. Möchte die Applikation z.B. auf einen Tastendruck reagieren, so überschreibt sie die Methode `onKeyDown` der Klasse `WWindow`.

Möchte die Applikation nun etwas im Fenster zeichnen, so ruft sie die entsprechenden Methoden der Klasse `WWindow` auf. Die Klasse `WWindow` wiederum benutzt die Portalaufrufe des Objektes `WWindowImpl`, um die entsprechenden Funktionen auszuführen.

Abb. 4.3 zeigt den Programm- bzw. Datenfluß zwischen einem Objekt der Klasse `WWindow` und einem Objekt der Klasse `WWindowImpl`.

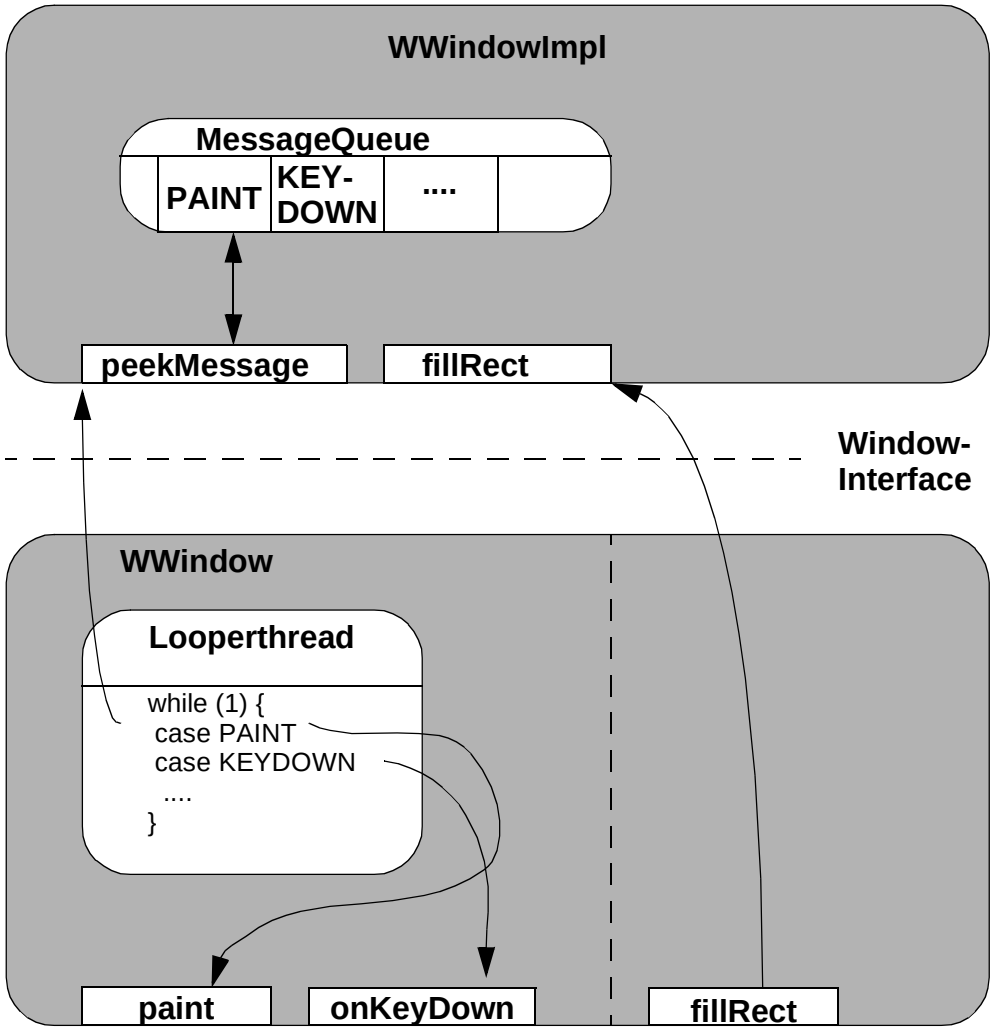



**Abb. 4.3 Programmfluß zw. Client-Window und Server-Window**

## 4.6 Eingabeevents

Besondere Bedeutung kommt den Eingabeevents zu. Wichtig ist dabei, daß die Events schnell genug verarbeitet werden, damit es nicht zu ungewollten Verzögerungen kommt. Bei Tastatureingaben ist das noch nicht so wichtig, da hier die Events nicht besonders schnell reinkommen. Anders ist das jedoch bei Mausbewegungen. Hier muß dafür gesorgt werden, daß die Events sofort verarbeitet werden, da es sonst zu Sprüngen des Mauszeigers kommen kann.

Abb. 4.4 zeigt wie der Programmfluß bei einem Eingabeereignissen ist.

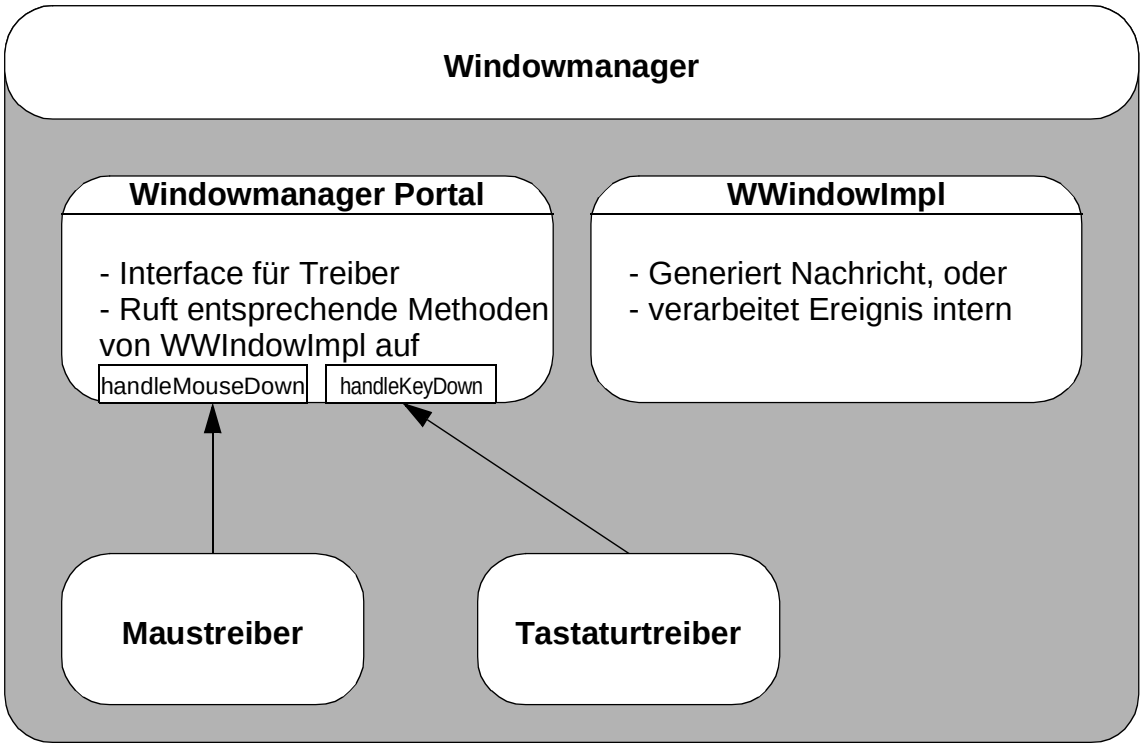



**Abb. 4.4 Programmfluß bei Eingabeereignissen**

Maus- und Tastaturtreiber laufen normalerweise in der Domain des WIndowmanagers. Tritt nun ein Ereignis ein, so ruft der Treiber eine entsprechende Handlerfunktion des Windowmanagers auf. Der Windowmanager wiederum ruft dann statische Methoden der Klasse `WWindowImpl` auf. Je nachdem, welche View des Fensters das Ereignis betrifft, wird entweder eine Nachricht an die Applikation geschickt, oder das Ereignis intern verarbeitet.

## 4.7 Zusammenfassung

In diesem Kapitel wurde auf die Architektur des WIndowmanager eingegangen. Es wurde das zugrundeliegende Client-Server-Modell erläutert, sowie der Programmfluß zwischen den einzelnen Komponenten. Es wurde gezeigt, daß ein Fenster quasi 'zweimal' existiert; nämlich einmal auf der Client-Seite und einmal auf der Server-Seite. Hierbei wurde detailliert der Aufbau beider Seiten beschrieben. Das Handling von Eingabeevents beendete dieses Kapitel.

# 5 Implementierung

## 5.1 Überblick

Das folgende Kapitel gibt einen detaillierten Einblick in die Implementierung. Dabei werden die wichtigsten Klassen und ihre Methoden angesprochen. Zuerst werden die grundlegenden Datentypen erläutert, die von allen Komponenten benutzt werden. Dann wird ein genauer Blick auf die Gerätetreiber geworfen und die Schnittstellen beschrieben. Im Anschluß werden die Klassen WBitmap und WRegion erläutert, die beide von der Klasse WView benötigt werden. Darauf aufbauend wird die Klasse, die für die Fenster zuständig ist erklärt. Den Schluß des Kapitels bildet die Implementierung des Windowmanagers, sowie die Klasse WWindow, die als Applikationsschnittstelle gedacht ist.

Bei den vorgestellten Klassen werden immer nur die wichtigsten Methoden vorgestellt, nebensächliche Funktionen wurden weggelassen, um die das ganze übersichtlich zu halten.

## 5.2 Grundlegen Datentypen

Zunächst einmal werden grundlegende Datentypen erläutert, die sowohl vom Windowmanager, als auch von Gerätetreibern und der Applikationsschnittstelle benutzt werden.

### 5.2.1 Koordinaten-Datentypen

Um Koordinaten oder Rechtecke zu beschreiben existieren folgende zwei Klassen:

- `class PixelPoint`
  Diese Klasse repräsentiert Punkte eines zweidimensionalen Koordinatensystems. Die Koordinaten werden dabei als Integerzahlen gespeichert.
- `class PixelRect`
  Ein PixelRect-Objekt repräsentiert ein Rechteck. Sie werden dazu benutzt um die Frames von Fenstern, Views oder Bitmaps zu definieren. Das Rechteck wird

durch zwei Koordinatentupel definiert. Abb. 5.1 zeigt ein Objekt der Klasse `PixelRect`.

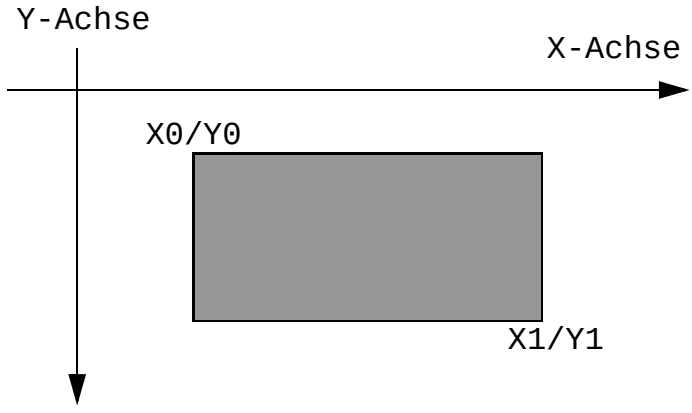



**Abb. 5.1 Ein Objekt der Klasse `PixelRect`**

### 5.2.2 Darstellung von Farben

Wie im Kapitel 2.6.2 auf Seite 13 schon erläutert wurde, wird eine Farbe durch vier Komponenten repräsentiert. Die dazugehörige Klasse ist die Klasse `PixelColor`. Ein Objekt dieser Klasse besitzt vier Farb-Komponenten: Rot, Grün, Blau und Alpha. Jede dieser Komponenten wird als `byte` abgespeichert und der Wertebereich geht von 0...255. Der Alphawert zeigt an wie transparent die Farbe ist (momentan noch ungenutzt). Abb. 5.2 zeigt den Aufbau der Klasse `PixelColor`.

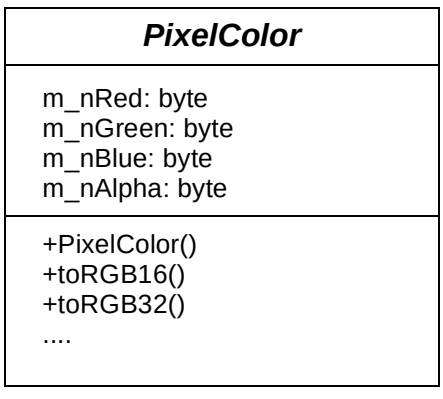



**Abb. 5.2 Die Klasse `PixelColor`**

Die Methoden *toRGB16()* und *toRGB32()* dienen dazu, den Farbwert in die entsprechende Farbdarstellung umzuwandeln.

### 5.2.3 Farbräume (Color Spaces)

Die Art auf die Farben für eine Bitmap spezifiziert werden, hängt ganz vom Farbraum (Color Space) ab, in dem sie interpretiert werden. Der Farbraum gibt die Farbtiefe einer Bitmap an (d.h. wieviele Bits pro Pixel abgespeichert werden). Der Farbraum bestimmt weiterhin, wie die Farben interpretiert werden, ob er Grauschattierungen oder True Colors repräsentiert, ob er in Farbkomponenten aufgespalten ist, usw...

Der Farbraum wird durch ein Objekt der Klasse `ColorSpace` repräsentiert. Momentan werden folgende Farbräume unterstützt:

- CS_RGB16
  Drei Komponenten pro Pixel, mit folgendem Anordnung (r = rot, g = grün, b = blau):
  `rrrrrgggggbbbbb`
- CS_RGB32
  Vier Komponenten pro Pixel, mit folgender Anordnung (r = rot, g = grün, b = blau, 0 = ungenutzt):
  `00000000rrrrrrrrggggggggbbbbbbbb`

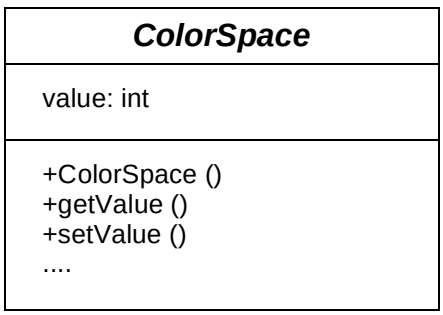



**Abb. 5.3 Die Klasse `ColorSpace`**

### 5.2.4 Zeichenmodi (Drawing Modes)

Der Zeichenmodus bestimmt, auf welche Art und Weise die Pixel des Hintergrunds mit dem zu setzenden Pixel kombiniert werden. Der Zeichenmodus wird durch ein Objekt der Klasse `DrawingMode` repräsentiert. Es werden folgende Zeichenmodi unterstützt:

- DM_COPY
  Ersetzt den Hintergrundpixel mit dem zu zeichnenden Pixel.
- DM_INVERT
  Invertiert den Hintergrundpixel.
- DM_OVER (wird nur beim Zeichnen von Bitmaps verwendet)
  Zeichnet nur diejenigen Pixel einer Bitmap, die nicht einem vorher definierten Farbwert entsprechen. Dieser Zeichenmodus wird verwendet um eine Bitmap mit einer Maske (definiert durch einen bestimmten Farbwert) zu zeichnen.

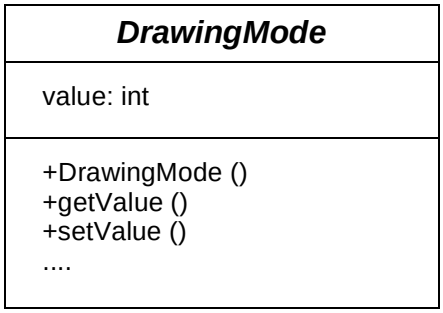



**Abb. 5.4 Die Klasse `DrawingMode`**

## 5.3 Geräte-Treiber

### 5.3.1 Grafikkarte

Eine Grafikkarte muß folgende zwei Interfaces zur Verfügung stellen. Zum einen das Interface `DeviceFinder` zur Detektion der Grafikkarte, zum anderen das Interface `FramebufferDevice`, daß die eigentliche Schnittstelle zur Grafikkarte ist.

#### 5.3.1.1 Detektion der Grafikkate

Um eine vorhandene Grafikkarte anzusprechen bedient sich der Windowmanager der Hilfe des Interfaces `DeviceFinder`, daß nur eine Funktion zur Verfügung stellt:

```
public interface DeviceFinder {
    public Device[] find();
}
```

**Abb. 5.5 Das `DeviceFinder-Interface`**

Ist eine Grafikkarte vorhanden, so gibt der Aufruf von `find()` eine nichtleere Liste zurück. Diese Liste enthält ein Objekt der entsprechenden Grafikkarte. Das Objekt muß das Interface `FramebufferDevice` (im nächsten Kapitel beschrieben) implementieren.

#### 5.3.1.2 Schnittstelle zur Grafikkarte

Um eine Grafikkarte ansprechen zu können, muß natürlich eine geeignete Schnittstelle existieren. Die Schnittstelle enthält folgende Funktionen:

- Setzen des Bildschirmauflösung
- Zeichnen einer Linie
- Zeichnen eines Rechtecks
- Kopieren eines rechteckigen Bereichs

Diese Funktionen werden allerdings nicht direkt vom Windowmanager angesprochen, sondern auf dem Umweg über eine Bitmap. Das Bitmap-Objekt ruft dann die entsprechenden Funktionen der Grafikkarte auf. Natürlich kann es auch vorkommen, daß eine Grafikkarte gewisse Funktionen nicht beherrscht (z.B. Zeichnen eines Rechtecks). Dann wird diese Funktion vom Bitmap-Objekt selbst ausgeführt.

Abb. 5.6 zeigt das Interface für die Grafikkarte im Überblick.

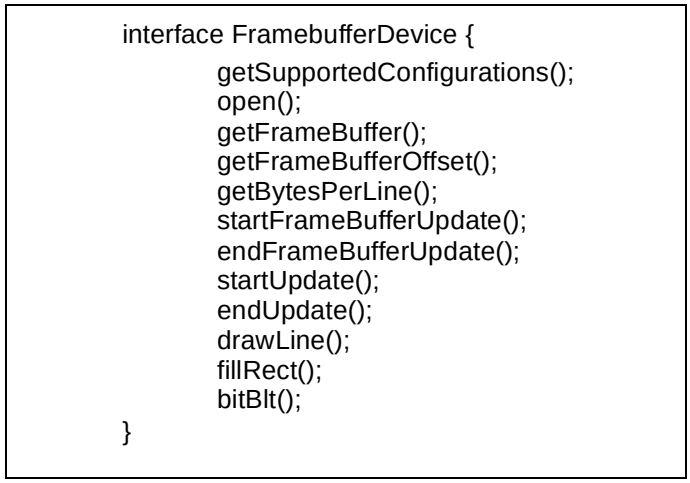



**Abb. 5.6 Das `FramebufferDevice-Interface`**

Die Grafikkarte muß das Interface `FramebufferDevice` implementieren, um korrekt angesprochen zu werden. Im folgenden werden die einzelnen Methoden näher erläutert:

- `DeviceConfigurationTemplate[] getSupportedConfigurations ()`

  Gibt die verfügbaren Videomodi zurück. Ein Videomodus wird dabei durch ein Objekt vom Typ `FramebufferConfigurationTemplate` repräsentiert.

- `public void open(DeviceConfiguration conf)`

  Wird aufgerufen, um die Grafikkarte mit einer bestimmten Auflösung zu öffnen. `DeviceConfiguration conf` ist dabei ein Objekt vom Typ `Framebuffer-Configuration`, daß die Bildschirmauflösung spezifiziert.

- `public DeviceMemory getFrameBuffer ()`

  Wird aufgerufen um ein DeviceMemory Objekt auf den Videospeicher zu erhalten. Dieses Objekt wird benötigt, damit Funktionen, die die Grafikkarte nicht beherrscht, emuliert werden können.

- `public int getFrameBufferOffset ()`

  Gibt den Offset innerhalb des Videospeichers an, an dem der eigentliche Bildschirm beginnt.

- `public int getBytesPerLine ()`

  Die Anzahl der Bytes pro Videozeile. Ist mindestens Breite des Bildschirms * Anzahl Bytes pro Pixel.

- `public void startFrameBufferUpdate () / public void endFrameBufferUpdate ()`

  Diese Funktionen werden aufgerufen, bevor und nachdem auf den Framebuffer direkt zugegriffen wird. Diese Funktion gibt der Grafikkarte die Möglichkeit ausstehende Beschleunigerfunktionen abzuschließen.

- `public void startUpdate () / public void endUpdate ()`

  Diese Funktionen werden aufgerufen, bevor und nachdem Beschleunigerfunktionen der Grafikkarte benutzt werden.

- `public int drawLine (PixelRect cDraw, PixelRect cClipped, PixelColor cColor, DrawingMode nDrawingMode)`

  Zeichnet eine geclippte Linie in der angegeben Farbe und Zeichenmodus. `cDraw` enthält dabei die Koordinaten der ungeclippten Linie. Unterstützt die Grafikkarte diese Funktion nicht, so ist der Rückgabewert -1 ansonsten 0.

- `public int fillRect (PixelRect cRect[], int nCount, PixelColor cColor, DrawingMode nDrawingMode)`

  `Zeichnet Rechtecke in der angegebenen Farbe und Zeichenmodus.` Unterstützt die Grafikkarte diese Funktion nicht, so ist der Rückgabewert -1 ansonsten 0.

- `public int bitBlt (PixelRect acOldPos[], PixelRect acNewPos[], int nCount)`

  `Kopiert rechteckige Bereiche des Bildschirms.` Unterstützt die Grafikkarte diese Funktion nicht, so ist der Rückgabewert -1 ansonsten 0.

### 5.3.2 Eingabegeräte

Es werden momentan zwei Arten von Eingabegeräten unterstützt: Maus und Tastatur.

#### 5.3.2.1 Datentypen und -strukturen für den Tastaturtreiber

Ein Objekt der Klasse `Keycode` repräsentiert den Wert einer gedrückten Taste. Dieser Wert entspricht dabei normalerweise dem ASCII-Wert der Taste und wird als Integerzahl abgespeichert. Werden dagegen Tasten gedrückt, die nicht in der ASCII-Tabelle vorkommen (z.B. Funktionstasten), so sollten die vordefinierten Konstanten genommen werden (z.B. VK_F1...VK_F12).
Der Tastaturtreiber muß dafür sorgen, das der Scancode in den entsprechenden Keycode umgewandelt wird. Momentan existiert noch keine Unterstützung für verschiedene Tastaturlayouts.

Ein Objekt der Klasse `Qualifier` gibt an, welche Modifizierungstasten (Shift, Alt, Ctrl) während des Tastendrucks gültig sind. Es sollten die vordefinierten Konstanten genommen werden (z.B. SHIFT, ALT, ...).

#### 5.3.2.2 Schnittstelle für Eingabegeräte

Um den Windowmanager Eingabeevents mitzuteilen sollten Eingabegeräte ein Objekt der Klasse `EventListener` benutzen. Diese Klasse verbirgt die Kommunikation zum Windowmanager und gibt den Eingabegeräten eine einfache Schnittstelle an die Hand. Abb. 5.7 zeigt die Klasse `EventListener` im Überblick.

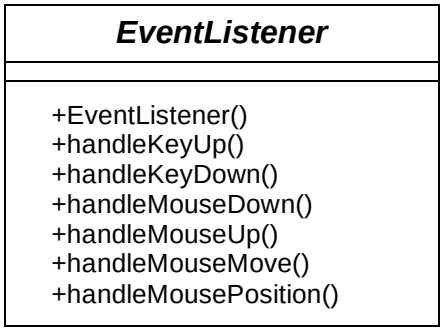



**Abb. 5.7 Die Klasse `EventListener`**

#### 5.3.2.3 Funktionen für Maustreiber

Für den Maustreiber stehen folgende Methoden des Interfaces `EventListener` zur Verfügung, um den Windowmanager Eingabeevents mitzuteilen.

- `void handleMouseDown (int nButton)`
  Behandlung eines Mouse Down Events. `nButton` gibt an welcher Button gedrückt wurde (1 für den ersten, 2 für den zweiten, 4 für den vierten, usw...).
- `void handleMouseUp (int nButton)`
  Behandlung eines Mouse Up Events. `nButton` gibt an welcher Button losgelassen wurde.
- `void handleMouseMove (int nDeltaX, int nDeltaY)`
  Behandlung einer Mausbewegung. Die beiden Parameter geben an, um wieviel Einheiten die Maus in X- bzw. Y-Richtung bewegt wurde.
- `void handleMousePosition (int nPosX, int nPosY)`
  Behandlung einer Mausbewegung. Die beiden Parameter geben an, auf welche Position (in Bildschirmkoordinaten) die Maus gesetzt werden soll.

#### 5.3.2.4 Funktionen für Tastaturtreiber

Für den Tastaturtreiber stehen folgende Methoden des Interfaces `EventListener` zur Verfügung.

- `void handleKeyDown (Keycode eKeyCode, Keycode eRawCode, Qualifiers eQual)`
  Teilt dem Windowmanager mit, daß eine Taste gedrückt wurde. Der erste Parameter ist dabei der umgewandelte Scancode. Der zweite Parameter ist der unbehandelte Scancode. Der letzte Parameter gibt an, welche Modifizierungstasten gültig sind (z.B. Shift oder Alt).
- `void handleKeyDown (Keycode eKeyCode, Keycode eRawCode, Qualifiers eQual)`
  Teilt dem Windowmanager mit, daß eine Taste losgelassen wurde. Der erste Parameter ist dabei der umgewandelte Scancode. Der zweite Parameter ist der unbehandelte Scancode. Der letzte Parameter gibt an, welche Modifizierungstasten gültig sind (z.B. Shift oder Alt).

## 5.4 Zugriff auf Bitmaps - Die Klasse WBitmap

Die Klasse WBitmap stellt das Herzstück aller Zeichenoperationen dar. Nur mit den Methoden dieser Klasse lassen sich die verschiedenen Operationen, wie Linien zeichnen oder Text ausgeben, ausführen. Eine Bitmap beschreibt ein rechteckiges Image als zweidimensionales Array von Pixeldaten. Die Bitmap speichert die Farbwerte jedes Pixels innerhalb eines rechteckigen Bereichs.

Die Bitmap kann entweder ein ganz normaler Speicherbereich sein oder sie kann sich auf den Speicher einer Grafikkarte beziehen. Dadurch ist es möglich, auch auf den Speicher der Grafikkarte zuzugreifen. Der Speicher wird dabei als `MemoryObjekt` verwaltet, daß eine größere Flexibilität als eine Byte-Array erlaubt.

Abb. 5.8 zeigt die Klasse WBitmap im Überblick.

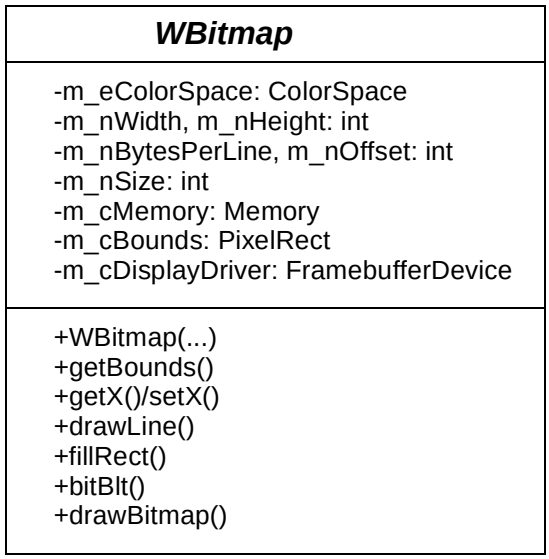



**Abb. 5.8 Die Klasse `WBitmap`**

Die Klasse stellt folgende Methoden zur Verfügung:

- `public WBitmap (int nWidth, int nHeight, ColorSpace eColorSpace, int nBytesPerLine)`
  Dieser Konstruktor erstellt eine Bitmap mit der angegeben Breite und Höhe (`nWidth` und `nHeight`). Der Farbraum wird durch den Parameter `eColorSpace` bestimmt. Der letzte Parameter gibt an wieviele Bytes eine Zeile der Bitmap breit ist. Dieser Wert muß mindestens so groß sein, wie die Breite der Bitmap multipliziert mit der Anzahl Bytes pro Pixel.
- `public PixelRect getBounds ()`
  Gibt das Rechteck zurück, daß die Größe der Bitmap definiert. Das Rechteck ist folgendermaßen spezifiziert: (0, 0 / nWidth - 1, nHeight -1).

- `public byte get8 (int nOffset)`
  `public void set8 (int nOffset, byte nValue)`
  `public short get16 (int nOffset)`
  `public void set16 (int nOffset, short nValue)`
  `public int get32 (int nOffset)`
  `public void set32 (int nOffset, int nValue)`
  Diese Funktionen setzen/lesen byte, short oder int Werte am angegebenen Byte-Offset.
- `public void drawLine (PixelRect cDraw, PixelRect cClip, PixelColor cColor, DrawingMode nDrawingMode)`
  Zeichnet eine Linie die durch das Rechteck `cDraw` spezifiziert wird. Der Parameter `cClip` definiert ein Rechteck, gegen das die Linie geclippt wird. `cColor` gibt die Farbe an, mit der die Linie gezeichnet werden soll. `nDrawingMode` spezifiziert den Zeichenmodus.
- `public void fillRect (PixelRect cRect[], int nCount, PixelColor cColor, DrawingMode nMode)`
  Zeichnet ausgefüllte Rechtecke. Die Rechtecke werden durch das Array `cRect[]` beschrieben. `nCount` gibt an wieviele Rechtecke gezeichnet werden sollen. `cColor` gibt die Farbe an, mit der die Rechtecke gezeichnet werden soll. `nDrawingMode` spezifiziert den Zeichenmodus.
- `void bitBlt (PixelRect acOldPos[], PixelRect acNewPos[], int nCount)`
  Kopiert rechteckige Bereiche der Bitmap. Die beiden Arrays `acOldPos[]` bzw. `acNewPos[]` geben an welche Rechtecke wohin kopiert werden sollen. `nCount` spezifiziert die Anzahl der Rechtecke, die kopiert werden sollen.
- `public void drawBitmap (WBitmap cBitmap, PixelRect cDst, PixelRect cSrc, PixelRect cClp, DrawingMode nMode)`
  Zeichnet eine Bitmap. `cDst` gibt an, in welchem Bereich der Zielbitmap gezeichnet werden soll. `cSrc` gibt an welcher Bereich der Quellbitmap genommen werden soll. Es ist möglich, daß sich Breite und Höhe der beiden Rechtecke unterscheiden. Dadurch läßt sich die Quellbitmap strecken bzw. stauchen. `cClp` gibt den Bereich an, gegen den geclippt wird. `nMode` definiert den Zeichenmodus.

## 5.5 Definition sichtbarer Bereiche - Die Klasse WRegion

Um das Zeichnen möglichst effizient zu gestalten und um nur die Bereiche von Fenstern respektive Sichten zu zeichen, die wirklich sichtbar sind, muß man diese Bereiche in geeigneter Weise definieren.

Eine Region besteht aus einer Anzahl von Rechtecken die sich nicht überlappen. Abb. 5.9 zeigt ein Beispiel für eine Region, die drei Rechtecke beinhaltet. Diese drei Rechtecke definieren den sichtbaren Bereich der Region.

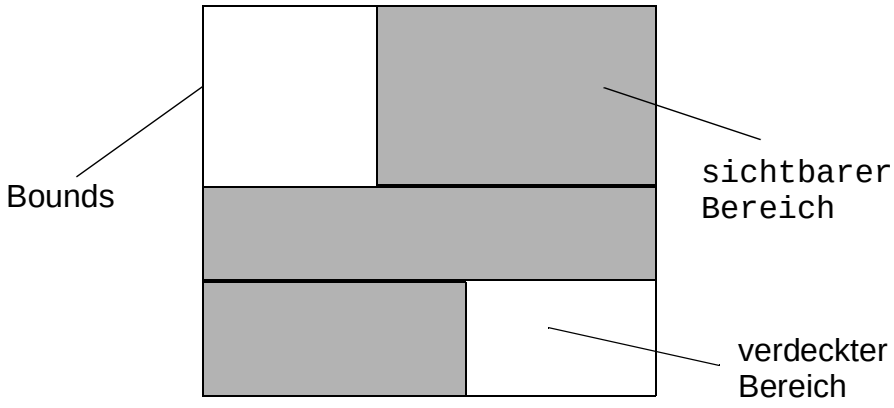



**Abb. 5.9 Eine Region**

Beim Einfügen oder bei der Herausnahme von Rechtecken aus dem sichtbaren Bereich, werden die verbleibende Rechtecke immer dahingehend optimiert, daß nur noch die minimale Anzahl an Rechtecken, die zur Definition des sichtbaren Bereichs nötig ist, benutzt werden.

Die Klasse WRegion ist für die Verwaltung solcher Regionen zuständig. Abb. 5.10 zeigt die Klasse mit ihren wichtigsten Methoden im Überblick.

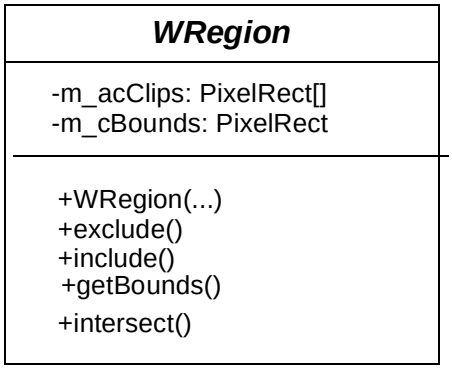



**Abb. 5.10 Die Klasse `WRegion`**

Im folgenden werden die wichtigsten Methoden kurz erläutert:

- `WRegion(), WRegion(PixelRect cRect), WRegion(WRegion cRegion)`
  Die Konstruktoren erzeugen eine leere bzw. nichtleere Region

- `void exclude(PixelRect cRect), void exclude(WRegion cRegion)`
  Das Rechteck bzw. die Region, daß durch `cRect` bzw. `cRegion` spezifiziert wird, wird aus der Region herausgenommen
- `void include(PixelRect cRect), void include(WRegion cRegion)`
  Ein Rechteck bzw. eine Region werden in den sichtbaren Bereich aufgenommen
- `PixelRect getBounds()`
  Liefert das Rechteck, daß den gesamten sichtbaren Bereich umspannt
- `intersect(WRegion cRegion)`
  Berechnet den Durchschnitt mit der angegebenen Region

## 5.6 Der Zugriff auf Sichten - Die Klasse WView

Die wohl umfangreichste Klasse des Windowmanagers ist die Klasse WView. Ein Objekt dieser Klasse spezifiziert zunächst einmal einen rechteckigen Bereich. Bei diesem Bereich kann es sich entweder um das eigentliche Fenster handeln, oder um den Arbeitsbereich des Fensters. Views sind hierarchisch organisiert, d.h. jede View kann eine beliebige Anzahl an Childviews haben. Zur Zeit gibt es aber keine Unterstützung für das Erzeugen und Verwalten von Views seitens der Applikationsseite.

Wenn der Windowmanager initialisiert wird, wird eine initiale View (die Topview) erzeugt, die Bildschirm den Bildschirm repräsentiert. Alle Fenster, die danach erzeugt werden, sind immer Childview der Topview.

Jede View besitzt einen sichtbaren Bereich (in dem gezeichnet werden darf), der durch ein Objekt der Klasse `WRegion` definiert wird. Wird eine Childview erzeugt, so wird der sichtbare Bereich der Childview aus dem sichtbaren Bereich der Topview herausgenommen. Damit wird gewährleistet, daß nur im tatsächlich sichtbaren Bereich der View gezeichnet wird.

Abb. 5.11 zeigt die Klasse `WView` mit ihren wichtigsten Methoden und Membervariablen.

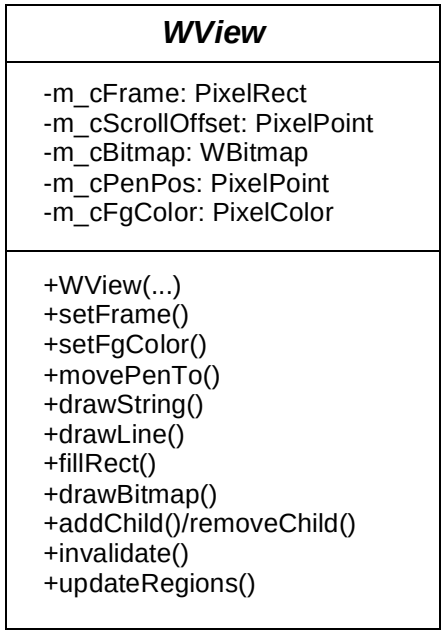



**Abb. 5.11 Die Klasse `WView`**

Jede View besitzt neben dem Frame noch einen Scrolloffset, der angibt, welche Ausschnitt der View sichtbar ist (siehe Kapitel 2.4.3 auf Seite 9). Momentan wird das Scrolling noch nicht unterstützt, da noch keine Schnittstelle dazu existiert.

### 5.6.1 Zeichnen innerhalb einer View

Damit man innerhalb der View etwas zeichnen kann, besitzt jede View ein Objekt der Klasse `WBitmap`. Normalerweise ist das immer eine Referenz auf die Bitmap der Topview (die ja den Bildschirm repräsentiert).

Jede View besitzt einen Pen (Stift), der angibt von wo aus die nächste Linie bzw. Text gezeichnet wird. Nach Abschluß der Operation wird die Position des Pens automatisch auf den neuen Stand gebracht.

Die Farbe, die zum Zeichnen verwendet wird, wird durch die aktuelle Vordergrundfarbe bestimmt.

Folgende Methoden stehen zum Zeichen zur Verfügung:

- `void setFgColor (PixelColor cColor)`
  Setzt die Vordergrundfarbe, die zum Zeichnen verwendet wird
- `void movePenTo(PixelPoint cPoint)`
  Bewegt den Pen an die angegebene Position

- `void drawString(String cString, int nLength)`
  Gibt den angebenen Text an der momemtanen Pen-Position aus
- `void drawLine(PixelPoint cToPos)`
  Zeichnet eine Linie von der momentanen Textposition zur angegebenen Position. Die Pen-Position wird danach automatisch aktualisiert.
- `void fillRect(PixelRect cRect)`
  Zeichnet ein ausgefülltes Rechteck in der aktuellen Vordergrundfarbe
- `public void drawBitmap(WBitmap cBitmap, PixelRect cSrcRect, PixelRect cDstRect)`
  Zeichnet eine Bitmap. `cSrcRect` und `cDstRect` geben das Quell- bzw. Zielrechteck an. Da sich die Dimensionen durchaus unterscheiden können, ist es auch möglich die Bitmap zu strecken oder zu stauchen.

### 5.6.2 Textausgabe

Momentan beherrscht der Windowmanager nur die Ausgabe eines einzigen Bitmapfonts. Jedes Zeichen dieses Fonts ist 9 Pixel breit und 14 Pixel hoch. Jeder Pixel eines Zeichen wird dabei durch ein Byte repräsentiert. Ein Bytewert von 0x1 bedeutet, daß dieser Pixel sichtbar ist, 0x0 dagegen daß der Pixel nicht gezeichnet werden soll.

### 5.6.3 Behandlung von Paint-Ereignissen

Jedesmal, wenn ein vorher verdeckter Bereich wieder sichtbar wird, muß dieser Bereich neu gezeichnet werden. Deshalb wird innerhalb der View nicht nur eine Region mit dem sichtbaren Bereich verwaltet, sondern auch eine Region, die die neu zu zeichnenden Bereiche definiert. Die Funktion `updateRegions()` wird aufgerufen, um diese Regionen der View neu zu berechnen (siehe auch Kapitel 2.4.4 auf Seite 10). Zur Darstellung der verschiedenen Regionen werden jeweils separate Objekte der Klasse WRegion verwendet.

## 5.7 Verwaltung von Fenstern - Die Klasse WWindowImpl

Über die Klasse `WWindowImpl` werden Fenster erzeugt und verwaltet. Das Fenster besteht aus zwei Views. Die erste View (m_cWndBorder) enthält die Fensterkontrollen (Titelleiste, Buttons, ...). Die zweite View (m_cTopView) repräsentiert den Arbeitsbereich des Fensters. Die Klasse stellt außerdem die Schnittstelle zur Applikation dar.

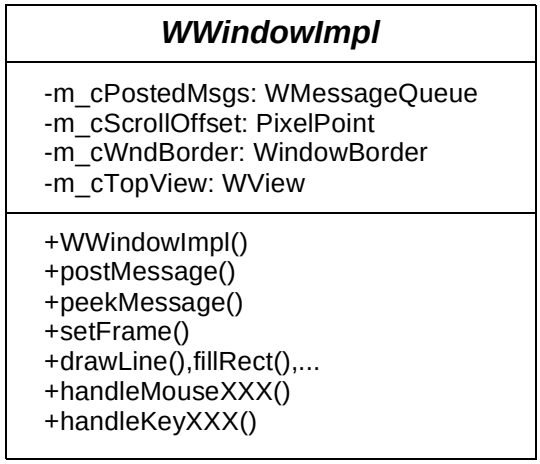



**Abb. 5.12 Die Klasse `WWindowImpl`**

Ereignisse, die die Borderview betreffen (Paint-Ereignisse, Maus- und Tastatur) werden vom Windowmanager selbst verarbeitet. Ereignisse, die den Arbeitsbereich betreffen, werden an die Applikation weitergeleitet.

### 5.7.1 Behandlung von Nachrichten

Jedes Fenster verwaltet eine Nachrichtenschlange, in der Nachrichten für die Applikation gespeichert werden. Die Methode `postMessage()` dient dazu, eine Nachricht an das Fenster zu schicken. Die Methode `peekMessage()` wird dazu benutzt, eine Nachricht aus der Nachrichtenschlange zu entnehmen. Ist die Warteschlange leer, so wird der aufrufende Thread blockiert, bis eine Nachricht in der Nachrichtenschlange eintrifft.

### 5.7.2 Behandlung von Mausereignissen

Tritt ein Mausereignis ein, so wird überprüft der Windowmanager zunächst, welches Fenster dafür zuständig ist. Gibt es ein Fenster das unter dem Mauszeiger liegt, so werden die entsprechenden Methoden des Fensters aufgerufen (`handleMouseXXX()`). Im nächsten Schritt wird überprüft, ob die Borderview oder die Clientview unter dem Mauszeiger liegt. Betrifft das Mausereignis den Arbeitsbereich, so wird es einfach an die Applikation weitergeleitet.

Bei der Behandlung von Mausereignissen, die die Borderview betreffen, werden folgende Fälle unterschieden:

- Mausbutton wird gedrückt
  Je nachdem welcher Button betätigt wird, werden folgende Aktionen ausgeführt:

a) Wird der Close-Button gedrückt, dann wird das Fenster geschlossen

b) Wird die Titelleiste gedrückt, und ist das Fenster noch nicht das aktive Fenster, so wird es in den Vordergrund gebracht und als aktives Fenster gesetzt.

c) Zoom-Button: Momentan nicht genutzt

d) Depth-Button: Momentan nicht genutzt

e) Resize-Button: Die Position der Maus wird sich gemerkt.

- Maus wird bewegt
  Wurde zuvor ein Button oder die Titelleiste gedrückt, so lassen sich folgende Aktionen daraus ableiten:

  a) Titelleiste: Das Fenster wird verschoben.

  b) Resize-Button: Die Größe des Fensters wird verändert.

### 5.7.3 Behandlung von Tastatureingaben

Momentan werden alle Tastatureingaben direkt an die kontrollierende Applikation des aktiven Fenster weitergeleitet. Innerhalb des Windowmanager findet keine Verarbeitung von Tastatureingaben statt.

### 5.7.4 Zeichnen

Alle Zeichenoperationen, die die Applikation tätigt, werden momentan einfach an die Topview weitergeleitet. Im Grunde stehen dieselben Methoden zum Zeichen bereit, wie bei der Klasse `WView`.

## 5.8 Verwaltung des Mauszeigers

Der Mauszeiger wird als Sprite dargestellt. Ein Sprite ist eine Bitmap, die sich, unabhängig von irgendwelchen Fenstern oder Views, irgendwo auf dem Bildschirm anzeigen und verschieben läßt.

Bevor das Sprite gezeichnet wird, wird der Hintergrund in dem Bereich, in dem das Sprite gezeichnet werden soll, gesichert. Wird das Sprite verschoben, so wird der Hintergrund wiederhergestellt und die ganze Prozedur wiederholt.

Ein Sprite wird durch ein Objekt der Klasse `WSprite` repräsentiert. Es hat neben der Bitmap, die angezeigt werden soll, noch eine Bitmap, die den Hintergrund aufnimmt. Diese Hintergrundbitmap hat dabei dieselbe Farbtiefe wie der Bildschirm. Desweiteren wird noch ein sogenannter Hotspot (in Sprite-Koordinaten) definiert, der einen einzigen

Pixel angibt, der aktiviert wird, wenn der Benutzer z.B. einen Mausbutton drückt. Um z.B. einen Fensterbutton zu drücken, muß sich der Hotspot innerhalb des Buttons befinden.

Der Windowmanager verwaltet das Sprite, das den Mauszeiger darstellt. Wird die Maus bewegt, so wird das Sprite entsprechend auf dem Bildschirm verschoben.

## 5.9 Der Windowmanager - Die Klasse WWindowManagerImpl

Der eigentliche Windowmanager wird durch ein Objekt der Klasse `WWindowManger-Impl` repräsentiert. Die Klasse implementiert dabei das Interface `WindowManager`, daß die Schnittstelle für Applikationen bildet. Das Interface stellt folgende Methoden zur Verfügung:

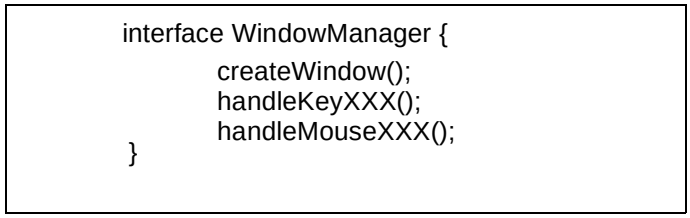



**Abb. 5.13 Das `WindowManager-Interface`**

Die einzige, für Applikationen interessante Methode, ist die Methode `createWindow()`. Sie erzeugt ein Fenster.

Die Methoden, die mit `handleKeyXXX()` und `handleMouseXXX()` bezeichnet sind, sind Methoden, die für Eingabegeräte gedacht sind.

### 5.9.1 Initialisierung

Der erste Schritt der Initialierung besteht darin, das Portal zum Windowmanager zu registrieren. Im nächsten Schritt wird nach einer verfügbaren Grafikkarte gesucht, und diese initialisiert (d.h. der Bildschirmmodus wird umgeschaltet). Danach wird die Topview erzeugt und initialisiert. Die Topview stellt die Wurzel der View-Hierarchie dar. Alle anderen Views, die danach erzeugt und angezeigt werden, sind immer Childviews der Topview. Das Erzeugen des Mauszeigers beendet die Initialisierungsphase.

### 5.9.2 Eingabegeräte und Behandlung von Eingabeevents

Möchte ein Eingabegerät dem Windowmanager ein Event mitteilen, so benutzt es die Methoden des WindowManager-Interfaces. Im wesentlichen sind es folgende Methoden, die benutzt werden: `handleMouseDown(), handleMouseUp(), handleMouseMoved(), handleMousePosition(), handleKeyDown(), handleKeyUp()`.

Um den Entwickler von Treibern nicht zu sehr mit den Details des Windowmanagers zu belasten, steht in der Klassenbibliothek die Klasse EventListener bereit, die die Details vor dem Entwickler verbirgt.

## 5.10 Die Schnittstelle für Applikationen - Die Klasse WWindow

Damit die Applikationsentwickler nicht zu sehr mit irgendwelchen Implementierungsdetails belastet werden, und um das Entwickeln von Applikationen zu vereinfachen, existiert die Klasse WWindow. Die Klasse verbirgt dabei im wesentlichen das Handling von Nachrichten und ruft stattdessen Memberfunktionen auf, die von der Applikation überschrieben werden können.

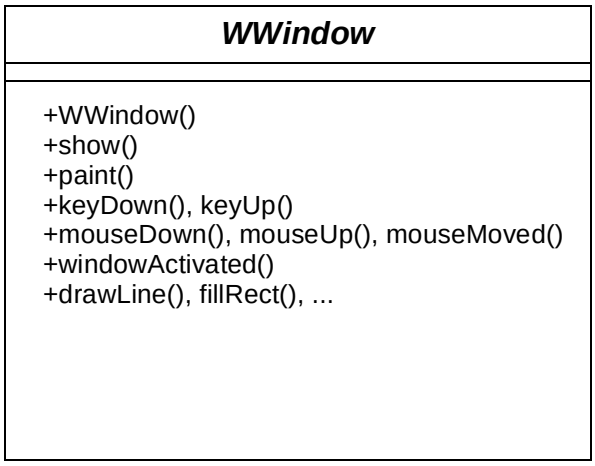



**Abb. 5.14 Die Klasse `WWindow`**

### 5.10.1 Reaktion auf Events

Möchte eine Applikation auf bestimmte Events reagieren, so überschreibt sie einfach die gewünschte Memberfunktion. Folgende Events stehen zur Auswahl:

- `keyDown(), keyUp()`
  Wird aufgerufen, wenn eine Taste gedrückt oder losgelassen wurde

- `mouseDown(), mouseUp(), mouseMoved()`
  Zeigt einen Mausevent an (Button gedrückt oder losgelassen, Maus bewegt)
- `windowActivated()`
  Wird aufgerufen, wenn das Fenster zum aktiven Fenster gemacht wurde, d.h. den Eingabefokus erhalten hat

### 5.10.2 Zeichnen und Textausgabe

Möchte die Applikation etwas zeichnen oder Text ausgeben, so stehen dieselben Funktionen, wie bei der Klasse `WWindowImpl` zur Verfügung. Tatsächlich macht die Klasse auch nichts anderes, als die entsprechenden Methoden der Klasse `WWindowImpl` aufzurufen.

## 5.11 Zusammenfassung

Dieses Kapitel gab einen umfassenden Überblick über die Implementierung. Es wurden die verschiedenen Klassen und Interfaces beschrieben, die entwickelt wurden. Vor allem die Schnittstellen für Gerätetreiber wurden genauer betrachtet. Ebenso intensiv wurde die Klasse WView betrachtet, da sie das eigentliche Herzstück des Windowmanagers bilden. Die Implementierung des Windowmanagers und die Erläuterung der Applikations-Schnittstelle, rundeten dieses Kapitel ab.

# 6 Schwierigkeiten und Details

## 6.1 Überblick

In diesem Kapitel werden auf die Details und Schwierigkeiten bei der Entwicklung eingegangen.

## 6.2 Entwickeln unter einer nicht-stabilen Laufzeitumgebung

Ein großes Hindernis, daß sich im Laufe der Entwicklung immer wieder herausgestellt hat, ist das Betriebssystem selbst. Da sich JX noch in der Entwicklung befindet, und alle Komponenten noch nicht 100%ig ausgereift sind, hat sich die Entwicklung immer wieder durch Fehler im Betriebssystem verzögert.

## 6.3 Fehlende Dokumentation

Ein weiterer großer Minuspunkt ist der Mangel an Dokumentation. Vor allem die im vorigen Kapitel angesprochenen Probleme hätten vermieden werden können, wenn es entsprechende Dokumentationen gegeben hätte. So war man gezwungen, entweder alles mittels Try and Error in Erfahrung zu bringen, oder ständig Rücksprache mit den Betreuern zu halten.

Desweiteren wäre eine Klassenübersicht nützlich gewesen, die die Betriebssystemspezifischen Klassen zeigen. Auch die Dokumentation der Klassen an sich ist nur als mangelhaft zu bezeichnen, weswegen ich des öfteren dazu gezwungen war, mir die entsprechenden Quellcodes anzuschauen, um die Funktionsweise zu verstehen.

## 6.4 Vorgehensweise bei der Implementierung

JX kennt zwei Betriebsmodi. Zum einen kann es direkt auf der Hardware laufen, zum anderen gibt es noch die Möglichkeit, daß System als eigenständigen Prozeß unter Linux zu starten. Da die Entwicklung des Windowmanagers unter anderem den Zugriff auf die Grafikhardware braucht, konnte ich allerdings nicht auf den zweiten Betriebsmodus zurückgreifen. Dadurch verzögert sich die Entwicklung ziemlich, da man ständig gezwungen ist, den Rechner neu zu booten, um seinen Code zu testen.

Um diesen Vorgang ein wenig zu beschleunigen, habe ich mich dazu entschieden, einen PC-Emulator (VMWare) zu verwenden. Der PC-Emulator emuliert einen kompletten PC mit Festplatte, Maus, Tastatur und Grafikkarte. Die Entwicklungsschritte sind zwar im wesentlichen immer noch die selben, doch kann man jetzt alles an einer Maschine machen, da der PC-Simulator ein ganz normales Programm ist.

Erst gegen Ende der Studienarbeit kam die Möglichkeit hinzu, den zweiten Betriebsmodus zu nutzen. Möglich wurde dies durch die Entwicklung einer sogenannten Framebuffer-Emulation. Wird JX als Prozeß unter Linux gestartet, so wird ein Fenster erzeugt, daß eine Grafikkarte emuliert.

## 6.5 Zugriff auf Hardwareregister unter Java

Beim Entwickeln von Hardwaretreibern ist man früher oder später mit dem Problem konfrontiert, auf die Register der Hardware zuzugreifen. Verwendet man eine Programmiersprache wie C oder C++, so hat man die Möglichkeit, diese Zugriffe mit Assemblerbefehlen zu gestalten. Da Java jedoch eine Plattformunabhängige Programmiersprache ist, und keine Assemblerbefehle kennt, muß man sich dafür einen anderen Weg einfallen lassen. Unter JX gibt es folgende zwei Möglichkeiten, um auf Hardwareregister zuzugreifen:

- Die Klasse `Ports` gibt einem die Möglichkeiten direkt auf Hardwareregister zuzugreifen. Die Klasse stellt einem dabei die Methoden `inb(int nPort)`, `inw(int nPort)` und `inl(int nPort)` zur Verfügung, um ein Register auszulesen. Die Methoden `outb (int nPort, byte nValue)`, `outw (int nPort, short nValue)`, `outl (int nPort, int nValue)` dienen dazu, ein Register zu beschreiben.
- Mit Hilfe der Klasse `DeviceMemory` lassen sich Hardwareregister, aber auch Hardwarespeicher, in den Hauptspeicher des PCs einblenden, und wie ein normales Memory-Objekt ansprechen.

## 6.6 Zusammenfassung

Das Kapitel gab einen Überblick über die aufgetretenen Schwierigkeiten während der Entwicklung.

# 7 Abschließende Betrachtungen

Dieses Kapitel beschreibt noch mal kurz die erreichten Ergebnisse und stellt mögliche Erweiterungen für die Zukunft vor.

Der Windowmanager bietet in vielen Bereichen volle Funktionalität. Allerdings konnte aufgrund der Komplexität der Aufgabe, nicht alles realisiert werden. Besonders die Applikations-Schnittstelle muß noch erheblich erweitert werden. Das Konzept der Views läßt sich noch ausbauen und ebenfalls in die Schnittstelle aufnehmen.

Während der Entwicklung wurde stets darauf geachtet, alles möglichst einfach und übersichtlich zu halten, was zu einem großen Teil gelungen ist.

Das implementierte System zeigt, daß es auch unter Java möglich ist komplexe Programm-Systeme zu entwerfen und zu entwickeln. Natürlich läßt sich die Performance nicht mit anderen Windowmanagern vergleichen, da zum einem keine Zeit mehr zur Verfügung stand, um entsprechende Tests zu entwickeln, und zum anderen viele Grafik-Funktionen von der Software übernommen werden.

# 8 Literaturangaben

Weiterführende Literatur zum Thema grafische Benutzeroberfläche:


[1] Apple. “*Aqua User Interface*”. Verfügbar auf *http://developer.apple.com/techpubs/macosx/Carbon/HumanInterfaceToolbox/Aqua/aqua.html*

[2] Be Inc. *“BeOS Multimedia Betriebssystem”*. Verfügbar auf *http://www.be.com/*

[3] KDE Project. *“KDE Desktop Environment”*. Verfügbar auf *http://www.kde.org/*

[4] The XFree86 Project,Inc. *“The XFree86 Project,Inc.”*. Verfügbar auf *http://www.xfree.org/*

[5] Microsoft. *“Microsoft Windows”*. Verfügbar auf *http://www.microsoft.com/windows/*


Papers über JX:


[6] M. Golm, J. Kleinöder, F. Bellosa. *“Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System”*. Verfügbar am Lehrstuhl für Betriebssysteme an der Universität Erlangen.

[7] M. Golm, J. Kleinöder. *“JX: Eine adaptierbare Java-Betriebssystemarchitektur”*. Verfügbar am Lehrstuhl für Betriebssysteme an der Universität Erlangen.

[8] M. Golm. *"JX flexible Operating System Architecture"*. Verfügbar auf *http://www4.informatik.uni.erlangen.de/Projects/JX/*

[9] M. Golm, M. Felser, C. Wawerisch, J. Kleinöder. *“The JX Operating System’”*. Verfügbar am Lehrstuhl für Betriebssysteme an der Universität Erlangen.


Weiterführende Links zu Java:


[10] Sun Inc. *“Java Developer Connection”*. Verfügbar auf *http://developer.java.sun.com/*

[11] Sun Inc. “Java Remote Method Invocation (RMI)“. Verfügbar auf http://java.sun.com/docs/books/tutorial/rmi/