Design und Implementierung der AWT-Schnittstelle für das Java-Betriebssystem JX

Studienarbeit im Fach Informatik

vorgelegt von

Marco Winter

geb. am 13.01.1979 in Lauf a. d. Peg.

Angefertigt am

Institut für Informatik (IV)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	***Prof. Dr. W. Schröder-Preikschat***
	Dipl.-Inf. M. Felser
	Dipl.-Inf. C. Wawersich
Beginn der Arbeit:	02.05.2002
Abgabe der Arbeit:	15.10.2002

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 14.10.2002 _____

Kurzfassung

Beinahe jedes moderne Betriebssystem besitzt eine grafische Benutzeroberfläche, welche es dem Anwender erlaubt, mittels Maus und Tastatur mit grafischen Anwendungen zu interagieren, und die dabei ein spezielles, für den Benutzer intuitives Interface bietet. Die Programmiersprache Java stellt, sofern sie für ein solches Betriebssystem implementiert wurde, durch die AWT-Bibliothek eine Schnittstelle zu dessen grafischer Oberfläche zur Verfügung.

Ziel dieser Studienarbeit war es, die AWT-Schnittstelle unter dem Java-Betriebssystem JX zu implementieren, und so die einfache Entwicklung grafischer Applikationen unter JX zu ermöglichen. Dazu sollte die aktuell verfügbare Version der Classpath AWT-Bibliothek verwendet werden, die bereits den betriebssystemunabhängigen Teil der AWT implementiert. Als Grundlage stand eine Implementierung eines Windowmanagers für JX zur Verfügung.

Um die AWT-Bibliothek des Classpath Projekts zu einer funktionsfähigen AWT zu vervollständigen, mussten für deren Komponenten-Klassen die zugehörigen Peers implementiert werden. Aufbauend auf dem Windowmanager, wurden die Peers für das Menü- sowie das Fenstersystem implementiert, so dass es möglich ist, eigene fensterbasierte Anwendungen samt einer Menüleiste zu realisieren. Um diese Fenster mit Elementen füllen zu können, wurden für einen Großteil der vorhandenen AWT-Komponenten aus der Classpath-Bibliothek, wie Knöpfe oder Textzeilen, die zugehörigen Peers realisiert, sowie das Layoutmanagement der AWT implementiert. Damit die AWT-Komponenten auch aktiv Ereignisse verarbeiten können, wurde das 1.1-Ereignismodell der AWT implementiert. Daneben wurde die Systemschnittstelle, die unter der AWT in der abstrakten Klasse `Toolkit` gekapselt wird, für die JX-Umgebung grundlegend implementiert. Durch die Implementierung der `Graphics`-Klasse wird das Erstellen von benutzerdefinierten Komponenten, die z.B. von der AWT-Komponente `Canvas` abgeleitet sind, ebenfalls unterstützt. Die grundlegende Implementierung der Klasse `Image` gestattet es dem Programmierer, Grafiken von einem Dateisystem zu laden oder aus einem Speicherbereich zu erzeugen, und in seinen Anwendungen zu verwenden. Darüber hinaus wurde in den Peers die Unterstützung für einen Tastaturfokus implementiert.

Das Ergebnis dieser Arbeit war eine grundlegende Implementierung der AWT-Schnittstelle unter JX, welche einen Großteil des AWT-Standards realisiert, und ausreichend Funktionalität besitzt, um die Entwicklung AWT-basierter Anwendungen zu ermöglichen.

Abstract

Almost every up-to-date operating system contains a graphical user interface (GUI) which allows the user to interact with graphic-based applications by using the mouse and keyboard, and which provides a special, intuitive interface for the user. If implemented for such an operating system, the Java programming language features an interface to that GUI, which is realized by its AWT library.

The aim of this thesis was to implement the AWT interface for the Java-based operating system JX, and therefore simplify the development of graphic-based applications for JX. In addition, the most recent version of the Classpath AWT library had to be used, which already implements the OS-independent part of the AWT. As a fundament, an implementation of a window manager for JX was available.

To complete the AWT library of the Classpath project to a working AWT, the associated peers for the AWT component classes had to be implemented. Based on the window manager, the peers for the menu system and window system have been implemented, to make it possible to realize own window based applications, including a menu bar. To be able to fill these windows with GUI elements, the peers for the bigger part of the AWT components, e.g. buttons or text labels, have been implemented, as well as the layout management system of the AWT. The 1.1 event model of the AWT has been implemented to allow the AWT components to react to events. The system interface which resides in the abstract AWT class named `Toolkit` has been basically implemented for the JX environment. Having implemented the `Graphics` class, the creation of user defined components (e.g. which are derived from the AWT component `Canvas`) if also supported. The basic implementation of the AWT class `Image` allows the programmer to load images from a file system or create them from memory, and use them in own applications. In addition, the support for a keyboard focus has been implemented in the peers.

The result of this thesis was a fundamental implementation of the AWT interface for JX, which implements the bigger part of the AWT standard and contains enough functionality to allow the development of AWT-based applications.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Es gibt heutzutage eine Vielzahl von verschiedenen Betriebssystemen, die auf dem Markt zu finden sind, sei es für eingebettete Systeme z.B. im Auto oder im DVD-Player, oder für "vollwertige" Rechner. Und für fast jedes Betriebssystem gibt es auch eine grafische Benutzeroberfläche, die es dem Benutzer erlaubt, mit einfachen Mitteln auf einer anschaulichen Ebene mit dem System zu kommunizieren.

Solch eine Oberfläche ist meist fensterbasierend aufgebaut und stellt daneben eine Bibliothek mit grafischen Bausteinen zur Verfügung, mit denen sich die Fenster füllen lassen. Diese Bausteine können sowohl aktiv als auch passiv sein, d.h. sie können auf Ereignisse wie z.B. einen Mausklick oder einen Tastendruck reagieren. Auf diese Weise lassen sich Anwendungen erstellen, die sich über ein einfaches und komfortables Interface ansteuern lassen.

Das Vorhandensein einer solchen Oberfläche in einem Betriebssystem ist mittlerweile so selbstverständlich, dass Sun sein JDK von Anfang an mit einer Schnittstelle für den Zugriff auf die grafische Oberfläche des darunterliegenden Systems versehen hat, nämlich die AWT[1]. Diese soll in Java geschriebenen Anwendungen eine einheitliche Schnittstelle für die Gestaltung grafischer Frontends bieten.

1.1 Zielsetzung und Rahmenbedingungen

Ziel dieser Studienarbeit war es, eine AWT-kompatible Schnittstelle für das Betriebssystem JX zu entwickeln [GF+02][GF+02]. Bei JX handelt es sich um ein Java-basierendes Betriebssystem, das an der Universität Erlangen-Nürnberg entwickelt wird. Es besitzt einen kleinen Systemkern, der in C und Assembler geschrieben wurde und der den direkten Zugriff auf die darunterliegende Hardware steuert. Alle anderen Betriebssystemkomponenten wie z.B. ein Scheduler oder ein Dateisystem, sowie alle Anwendungen für JX, sind komplett in Java geschrieben. Durch die Verwendung von Java-Klassen als Betriebssystemkomponenten lässt sich JX sehr gut an die jeweilige Hardwarearchitektur anpassen. Darüber hinaus stellt es nicht so hohe Ansprüche an die Hardware wie andere Betriebssysteme, da Aspekte wie z.B. das Sicherheitskonzept einer MMU, bereits durch das sichere Klassenkonzept von Java implementiert werden [GF+02].

Es gibt für JX eine Implementierung eines Windowmanagers, die es erlaubt, eine beliebige Anzahl von Fenstern zu erstellen und zu verwalten [JO02]. Diese sehr mächtige und flexible Implementierung wurde als Basis für die Entwicklung der AWT-Schnittstelle verwendet. Daneben stellte sich heraus, dass es gar nicht notwendig ist, eine AWT-Schnittstelle von Grund auf neu zu schreiben, da es bereits andere Gruppen gibt, die auf diesem Gebiet Fortschritte erzielt haben. Für diese Arbeit wurde eine Betaversion der "non-peer AWT"-Implementierung des Classpath Projektes als Basis verwendet [CP03], die sich schon in einem relativ fortgeschrittenen Betastadium befindet, allerdings konzentriert sich die Entwicklungsarbeit bei der Classpath-Implemen-

1. Advanced Window Toolkit

tierung auf den betriebssystemunabhängigen Teil der AWT. Die Studienarbeit beschäftigte sich deswegen damit, den betriebssystemabhängigen Teil der AWT mit Hilfe des Windowmanagers von JX zu implementieren.

1.2 Legende

Im Folgenden werden die Formate und Darstellungsweisen erklärt, die in dieser Arbeit verwendet wurden.

Textbereich

Der Textbereich wurde wie folgt formatiert:

Format	Bedeutung
normaler Text	Dies ist der Standardschriftsatz, der für den normalen Text verwendet wird.
`Courier`	Wenn Klassen- oder Methodennamen benannt werden, wird diese Schrift verwendet. Beispiele hierfür sind `Component` oder `show()`.
<fetter Text>	Mit fett hervorgehobenem Text werden Tasten gekennzeichnet, wie z.B. **<Tab>** oder **<Return>**.

Zeichnungen und Skizzen

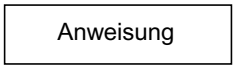
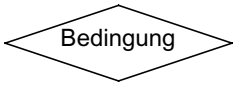
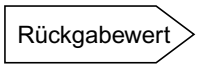
In den Grafiken wurden, falls es sich dabei um schematische Darstellungen von Klassen, Schichten und deren Beziehungen zueinander handelt, folgende Formatierungen verwendet:

Format	Bedeutung
Schicht	Mit diesem Layout werden in Grafiken die einzelnen Schichten eines Schichtenmodells voneinander abgegrenzt.
Typ	So werden die Klassen dargestellt, die nicht eindeutig spezifiziert sind, sondern nur einen bestimmten Grundtyp bzw. eine bestimmte Eigenschaft besitzen. Beispiele hierfür wären Komponente, Peer oder Kontainer.
Klasse	Soll dagegen eine spezielle Klasse benannt werden, so wird dieses Layout verwendet. Der Text im Inneren der Blase gibt den genauen Klassennamen an, z.B. **Component**, **Button** oder **Frame**.

Format	Bedeutung
	Diese Darstellung wird verwendet, wenn deutlich gemacht werden soll, dass eine Klasse von einer anderen Klasse abgeleitet ist. Die abgeleitete Klasse wird in den äußeren Bereich eingesetzt, während die Mutterklasse im inneren Bereich dargestellt wird.

Ablaufdiagramme

In Ablaufdiagrammen von Methoden wurde folgendes Format verwendet:

Format	Bedeutung
	In einem Rechteck werden normale Arbeitsschritte oder Anweisungen dargestellt, die in der Methoden durchgeführt werden.
	Die Raute steht für bedingte Verzweigungen im Code der Methode. Innerhalb der Raute steht die Bedingung, anhand der dann entschieden wird, wie der weitere Programmablauf aussieht.
	Falls die besprochene Methode in der Lage ist, einen Wert zurückzuliefern, zeigt dieses Symbol den jeweiligen Wert an, der in der aktuellen Verzweigung des Programmcodes zurückgegeben wird.
	Dieses Symbol stellt den Start bzw. das Ende des Ablaufs der Methode dar.

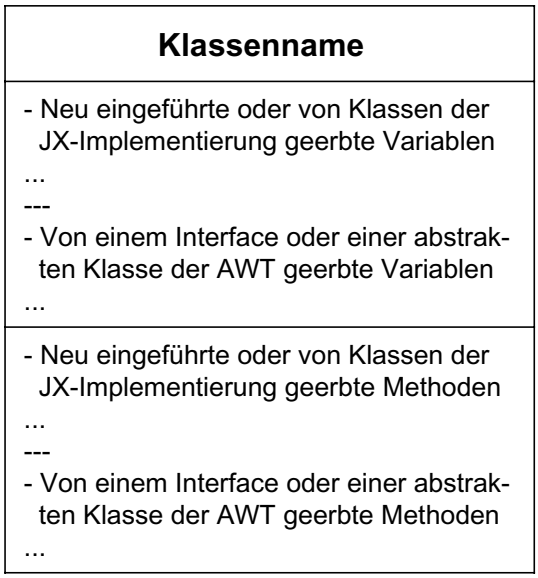
Darstellung der Klassem

Die in dieser Arbeit verwendeten Darstellungen der einzelnen Klassen halten sich weitestgehend an den UML-Standard. Es wurden daneben noch ein paar weitere Formatierungen vorgenommen:

- Die Variablen und Methoden der Klassen sind nach Herkunft in zwei Bereiche geteilt:
  - Zuerst werden diejenigen Variablen oder Methoden dargestellt, welche in dieser Klasse selbst implementiert oder von Klassen der JX-Implementierung (Peers u. a. Klassen) geerbt wurden.
  - Danach werden, durch ein “---” getrennt, all diejenigen Variablen und Methoden aufgeführt, die durch ein Interface oder eine abstrakte Klasse der AWT bekannt sind und implementiert wurden.

- Für jeden der beiden Bereiche gibt ein "..." an, dass im jeweiligen Bereich noch Variablen oder Methoden vorhanden sind, die jedoch nicht dargestellt wurden.

Die folgende Darstellung zeigt die Aufteilung der Klasse nochmals im Überblick:



2 Grundlagen

2.1 Überblick

Zum besseren Verständnis wird zuerst die AWT mit ihren Klassen und Konzepten vorgestellt, um eine Vorstellung davon zu bekommen, welche Funktionalität eine kompatible Implementierung bereitstellen muss. Anschließend werden die Grundlagen beschrieben, welche für die AWT-Implementierung unter JX verwendet wurden.

Wie bereits in der Einleitung erwähnt, baut diese Implementierung hauptsächlich auf zwei anderen Arbeiten auf:

- Der Windowmanager für JX von Jürgen Obernolte [JO02]
- Die “non-peer AWT”-Implementierung des Classpath Projektes, Version 0.03 [CP03]

Der größte Teil der Studienarbeit beschäftigte sich mit dem Ziel, eine Zwischenschicht zu konstruieren, welche beide Schichten so miteinander verbindet, dass eine funktionierende AWT-Schnittstelle entsteht. Beide Arbeiten werden im Anschluss an die Beschreibung der AWT erläutert.

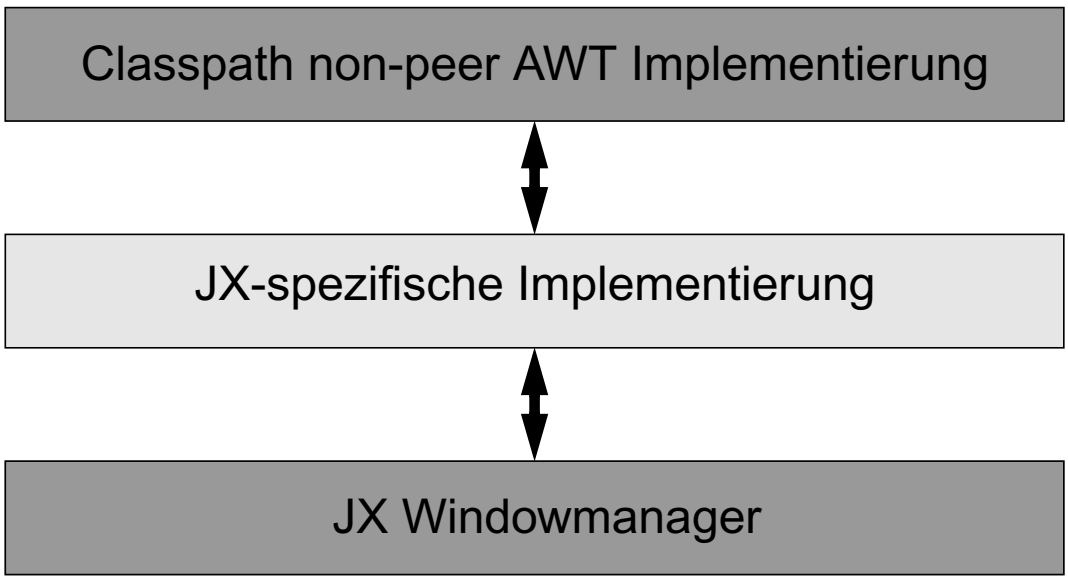


Abb. 2.1 Das Schichtenmodell der AWT-Implementierung für JX

2.2 Die AWT

Jedes Java-Entwicklungspaket, sei es das Java Development Kit (JDK) von Sun, den Erfindern von Java, oder das Kit von IBM, sowie die verschiedenen Laufzeitumgebungen für Java, bringen neben einer VM stets auch die Standardlaufzeitbibliothek von Java mit. Diese Bibliothek beinhaltet sämtliche Laufzeitklassen, die für den normalen Betrieb eines Java-Programmes notwendig sind: Grundlegende Klassen wie `System` oder `String`, Klassen zur Datenverwaltung (`Vector`, `HashTable`), Klassen für RPC, für den Dateizugriff, usw.

Ein Teil dieser Bibliothek, die sogenannte AWT, ermöglicht den Zugriff auf die grafische Oberfläche des darunterliegenden Betriebssystems und erlaubt es dem Benutzer, durch einfachen Gebrauch bestimmter AWT-Klassen, selbst fensterbasierte grafische Anwendungen zu programmieren [SM98]. Die AWT bedient sich dabei einer Vielzahl von Klassen und Konzepten, die im Folgenden kurz vorgestellt werden sollen.

2.2.1 Komponenten

Die AWT besitzt für die gebräuchlichsten GUI-Bausteine des darunterliegenden Betriebssystems eigene Klassen, z.B. für Knöpfe, Menüleisten oder ganze Fenster. Diese Klassen, die allesamt von der AWT-Klasse `Component` abgeleitet sind, repräsentieren die jeweiligen Bausteine auf der Java-Ebene und werden verwendet, um allgemeine und spezielle Eigenschaften der einzelnen Bausteine zu kontrollieren, z.B. Beschriftung, Position und Größe, oder Sichtbarkeit.

2.2.2 Kontainer

Eine Untergruppe der Komponenten sind die Kontainer, deren Mutterklasse `Container` ebenfalls von `Component` abgeleitet ist. Diese Kontainer haben die Eigenschaft, dass sie andere Komponenten (und damit auch Kontainer) aufnehmen können. Auf diese Weise kann man Komponenten hierarchisch organisieren und verwalten. Da z.B. ein Fenster unter der AWT ebenfalls ein Kontainer ist, lassen sich alle gewünschten Komponenten, die in dem Fenster zu sehen sein sollen, darin hierarchisch einbetten. Abbildung 2.2 skizziert eine mögliche Komponentenhierarchie.

2.2.3 Layoutmanagement

Durch Kontainer wird die grobe Einteilung der Komponenten gewährleistet, die genaue Größe und Position auf dem Bildschirm jedes Bausteins ist jedoch (noch) nicht bekannt. Um diese Aufgabe kümmert sich das Layoutsystem der AWT: Jedem Kontainer ist ein sogenannter Layoutmanager zugeordnet, der bei Bedarf (z.B. vor dem Sichtbarmachen des Kontainerinhalts) die Größe und Position der eingebetteten Komponenten den Umständen entsprechend anpasst. Die AWT kennt verschiedene Layoutmanager, welche für verschiedene Layouts zuständig sind. Ein

Beispiel dafür ist der Layoutmanager `BorderLayout`, welcher es ermöglicht, bis zu fünf eingebettete Komponenten in einem Kontainer nach den Richtungen Nord, Süd, Ost, West oder Mitte auszurichten.

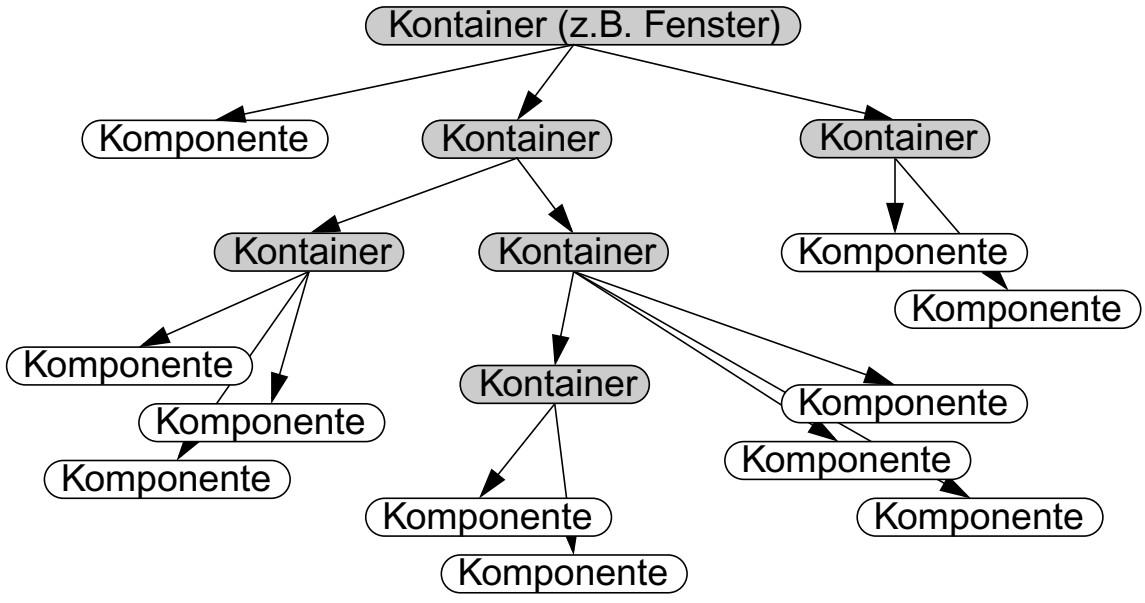


Abb. 2.2 Skizze einer Komponentenhierarchie unter der AWT

2.2.4 Ereignisbehandlung

Eine AWT-Komponente kann auf verschiedene Ereignisse reagieren, z.B. auf einen Mausklick, einen Tastendruck, das Wechseln des Tastaturfokus, usw. Genauso kann sie auch verschiedene Ereignisse auslösen. Wie man auf Ereignisse reagieren kann, hängt vom verwendeten Ereignismodell ab. Die AWT kennt zwei Modelle:

- Beim 1.0-Modell besitzt jede Komponente von Haus aus bestimmte Methoden, die je nach Eintritt eines bestimmten Ereignisses aufgerufen werden. Um hier auf bestimmte Ereignisse zu reagieren, muss man sich eine neue Klasse von der gewünschten Komponente ableiten und die jeweilige Methode mit eigenem Programmcode überschreiben, der beim Auftreten des Ereignisses ausgeführt werden soll.
- Das flexiblere 1.1-Modell dagegen führt sogenannte “Eventlistener” sowie eine Ereigniswarteschlange ein. Treten Ereignisse auf, so werden diese in der Warteschlange eingereiht, und von dort durch einen eigenen Verteiler-Thread an die entsprechenden Komponenten geschickt. Die Listener werden bei der jeweiligen Komponente registriert, und deren entsprechende Methoden ausgeführt, sobald ein passendes Ereignis bei der Komponente eintritt. Jede Komponente kann hier mehrere Listener verwalten, und jeder Listener kann bei mehreren Komponenten lauschen [PR98].

2.2.5 Benutzerdefinierte Komponenten

Die AWT bietet neben der Möglichkeit, bereits vordefinierte Komponenten zu verwenden, auch das Erstellen von eigenen Komponenten an. Obwohl sich theoretisch alle AWT-Komponenten um bestimmte Merkmale und Fähigkeiten ergänzen lassen, benötigt der Benutzer meist so etwas wie einen freien Platz in einem Fenster, in dem er tun und zeichnen kann, was er möchte. Zu diesem Zweck stellt die AWT die Komponente `Canvas` zur Verfügung, welche im Endeffekt einen solchen freien Bereich auf dem sichtbaren Fenster darstellt.

Aus dieser Komponente kann der Benutzer durch Ableitung eigene Komponenten schaffen, die komplett selbst gestaltet werden können. So kann man neben dem freien Design des Aussehens auch eigene Eventlistener anhängen, die auf bestimmte Ereignisse reagieren und beispielsweise das Aussehen der Komponente ändern. Einen eigenen Knopf zu erstellen, der seine Erscheinung verändert, wenn man mit der Maus darüberwandert, ist ebenso möglich, wie z.B. eine Zeichenfläche, auf der mittels Mausklicks und -bewegungen Linien gemalt werden können.

2.2.6 Das Zeichnen an sich

Zum Zeichnen in eine Komponente, u.a. auch bei `Canvas`, wird v.a. die Methode `paint()` verwendet, die jeder Komponente vererbt ist. Diese Methode wird von der AWT aufgerufen, sobald die Komponente sich neu zeichnen soll. Durch Ableiten einer Komponente und Überschreiben dieser Methode lässt sich das Aussehen einer Komponente beeinflussen bzw. komplett selbst gestalten.

Eine Komponente kennt an sich keine eigenen Methoden, um grafische Elemente wie Punkte, Kreise, Linien o.ä. zu zeichnen. Um dennoch Zeichenoperationen durchführen zu können, erhält die Methode `paint()` als Parameter ein Objekt des Type `Graphics`. Dieses Objekt besitzt Methoden, die es erlauben, entsprechende Formen mit der gewünschten Farbe und evtl. dem gewünschten Füllmuster zu zeichnen. Durch diese Kapselung von Zeichenfähigkeiten in ein spezielles Objekt ist es leicht möglich, verschiedene Derivate des Objektes für verschiedene Ausgabegeräte zu erstellen. So kann man ein `Graphics`-Objekt erzeugen, welches für die Bildschirmausgabe zuständig ist, während eine andere Instanz den Speicher eines Druckers ansteuert.

2.2.7 Grafiken

Die AWT benutzt selbst keine Grafiken für die Darstellung ihrer Komponenten, dennoch bietet sie dem Benutzer die Möglichkeit, Grafiken in eigenen Komponenten zu verwenden. Zur Verwaltung von Grafiken oder Bitmaps wird die Klasse `Image` verwendet. Diese abstrakte Klasse kapselt eine Grafik für die weitere Verwendung unter der AWT. Die AWT bietet einige Möglichkeiten, was die Verwendung von Grafiken in einer AWT-Anwendung anbelangt. Es ist z.B. möglich, Grafiken direkt vom Dateisystem oder von einer URL zu laden, sofern die Bilddaten

in einem unterstütztem Format vorliegen (im Normalfall nur JPEG oder GIF). Darüber hinaus lassen sich Grafiken mittels der Klasse `MemoryImageSource` auch aus Speicherbereichen herstellen.

Eine Besonderheit unter Java ist dabei, dass das Laden einer Grafik von einer externen Quelle (Datei oder URL) asynchron verläuft, d.h. der eigentliche Methodenaufruf stößt den Ladevorgang nur an, während die Methode sofort wieder zurückkehrt. Es gibt die Möglichkeit, sich durch andere Klassen über den Ladefortschritt zu informieren: Die Klasse `MediaTracker` beispielsweise kann verschiedene `Image`-Objekte registrieren, und mittels `statusID()` oder `statusAll()` den Status eines bzw. aller registrierten Objekte zurückliefern.

Will man eine bereits erzeugte Grafik manipulieren, so gibt es u.a. die Möglichkeit, ein `Graphics`-Objekt zu erzeugen, das seine Ausgaben in die Grafik zeichnet. Daneben bietet die AWT eine Reihe von sogenannten Filtern, die eine Grafik drehen, kippen, verdunkeln o.ä. können.

Die Ausgabe einer Grafik ist relativ einfach zu bewerkstelligen: Die Klasse `Graphics` besitzt einige Methoden, die es erlauben, Grafiken auch skaliert darzustellen. Will man also Grafiken in einer eigenen Komponente verwenden, genügt es, einen entsprechenden Methodenaufruf in der `paint()`-Methode der Komponente zu formulieren.

2.2.8 Erzeuger und Verbraucher bei Grafiken

Das Konzept von Erzeuger und Verbraucher wird in der AWT im Bereich “Grafiken” häufiger verwendet, und in der vorliegenden Implementierung miteinbezogen, deswegen wird es hier kurz erläutert.

Die AWT erlaubt es, Erzeuger und Verbraucher für Bilddaten zu erzeugen. Diese Objekte sind für den Austausch von Bilddaten verantwortlich: Ein Erzeuger liest Bilddaten aus der ihm zugeordneten Quelle und gibt diese an die registrierten Verbraucher weiter, welche die Daten dann in die ihnen zugeordneten Ziele speichern. Quelle und Ziel können dabei `Image`-Objekte oder auch andere Objekte sein.

Dieses Konzept macht v.a. Sinn, wenn es um Bildmanipulationen geht: In die Verbindung zwischen Erzeuger und Verbraucher lassen sich Filter, Multiplexer und andere Klassen einbauen, welche die Daten einlesen, evtl. modifizieren und weiterreichen.

Jede Klasse, die einen Erzeuger realisieren möchte, muss dafür das Interface `ImageProducer` implementieren, bei Verbrauchern trägt das Interface den Namen `ImageConsumer`. Die darin definierten Methoden steuern die Kommunikation zwischen den einzelnen Objekten so, dass die Verbraucher sich beim jeweiligen Erzeuger registrieren und ihm mitteilen müssen, wann er mit der Übertragung der Bilddaten beginnen soll. Dieser wiederum verwendet die Methoden der Verbraucher, um ihnen seine Daten zu übermitteln.

Eine häufig verwendete Implementierung des `ImageProducer`-Interfaces stellt die Klasse `MemoryImageSource` dar, welche Bilddaten aus einem Speicherbereich heraus übermittelt (vgl. vorherigen Abschnitt).

2.2.9 Peers

Das grafische Erscheinungsbild einer AWT-Anwendung sieht unter jedem Betriebssystem anders aus. Das liegt daran, dass die einzelnen vordefinierten Komponenten der AWT sich nicht selbst zeichnen, sondern auf die grafische Bibliothek des darunterliegenden Betriebssystem zurückgreifen, um sich selbst darzustellen. Ein Knopf, der unter Windows in einem Java-Fenster gezeichnet wird, kommt aus der selben Grafikbibliothek wie ein Knopf aus einem Windows-Fenster. Die AWT besitzt also eine Schnittstelle, die es gestattet, die darunterliegende grafische Bibliothek anzusprechen.

Die AWT, und davon speziell die Gruppe der Komponenten, lässt sich folglich grob in zwei Teile gliedern: Zum Einen gibt es einen betriebssystemunabhängigen Teil, der komplett in Java geschrieben ist und diejenigen Bestandteile enthält, die vom darunterliegenden System unabhängig funktionieren, wie z.B. das Layoutmanagement eines Fensters. Daneben existiert noch der Teil, der den direkten Zugriff auf das darunterliegende grafische Subsystem steuert und deswegen für jedes OS eigens geschrieben werden muss. Um diesen Teil ebenfalls einheitlich unter Java ansprechen zu können, wurden die sogenannten Peers eingeführt. Diese Klassen repräsentieren den betriebssystemabhängigen Teil der AWT-Komponenten (z.B. eines Knopfes) unter Java und besitzen ein definiertes Interface, über das es dem systemunabhängigen Teil gestattet ist, die Möglichkeiten der darunterliegenden grafischen Oberfläche zu nutzen.

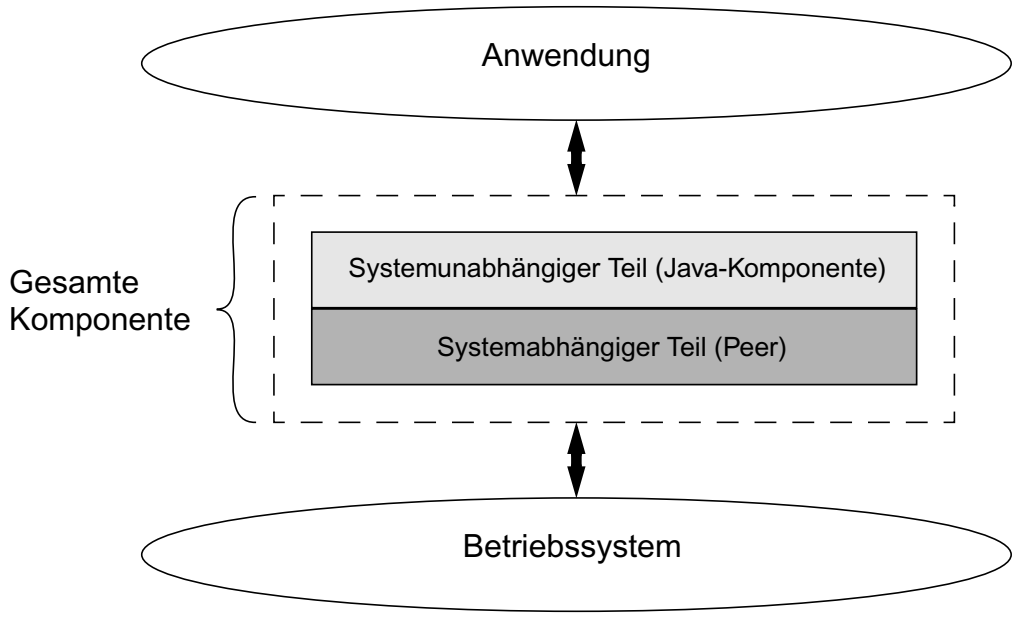


Abb. 2.3 Die zwei Teile einer AWT-Komponente

Eine AWT-Komponente besteht somit im Allgemeinen aus zwei Teilen: Einem Java-Teil, der die Komponente nach oben hin präsentiert, und dem zugehörigen Peer, der die Schnittstelle in das Betriebssystem realisiert. In der Praxis handelt es sich bei den Peers meist um einfache

Wrapper-Klassen, die jeden Methodenaufruf durch das Java Native Interface (JNI) direkt an eine native Bibliothek weiterleiten, welche dann die eigentlichen Funktionsaufrufe an der grafischen Oberfläche des Betriebssystems durchführt.

2.3 Die Classpath AWT

Das Classpath Projekt beschäftigt sich mit der Aufgabe, eine möglichst genaue Kopie der Laufzeitbibliothek von Java aufzubauen. Eine solche Bibliothek wird zwar von Sun oder IBM kostenlos ausgeliefert, die Benutzung ist aber an lizenzrechtliche Bestimmungen gebunden. Ziel von Classpath ist es, eine Open-Source-Implementierung der Bibliothek zu liefern, die jedermann ohne weiteres zugänglich ist.

Auch für die AWT gibt es bereits eine grundlegende Implementierung in der Classpath Bibliothek. Die Arbeit des Classpath Projektes konzentriert sich vor allem auf die Fertigstellung des betriebssystemunabhängigen Teils, der auch für diese Studienarbeit verwendet wurde. Zum Zeitpunkt dieser Arbeit war von der Classpath Bibliothek nur eine Betaversion verfügbar, die aber bereits sehr ausgereift war und stabil arbeitete.

Im Folgenden werden einige Details besprochen, die für diese Studienarbeit von besonderem Interesse waren oder einfach wichtig sind, um ein tieferes Verständnis für die Materie zu bekommen.

Anmerkung: Wenn im Folgenden von der AWT gesprochen wird, ist damit immer der betriebssystemunabhängige Teil gemeint. Desweiteren wird hier immer von der AWT-Implementierung des Classpath Projektes ausgegangen. Auch wenn diese mit dem Ziel programmiert wird, vollständig kompatibel zum Original von Sun zu sein, können kleine Abweichungen nicht ausgeschlossen werden, vor allem da die verwendete Version sich noch im Betastadium befand.

2.3.1 Struktur der AWT

Die AWT ist hierarchisch aufgebaut, wie man anhand der Abbildung 2.4 erkennen kann. Die AWT-Klassen lassen sich dabei grob in vier Kategorien einteilen:

- Zum Einen gibt es die Klasse `Component` und deren Ableitungen. Diese stellen die eigentlich sichtbaren Komponenten dar, mit denen ein Fenster gefüllt wird. Diese Gruppe lässt sich wiederum unterteilen in "normale" darstellbare Klassen und solche, die selbst weitere Klassen beherbergen können, ähnlich einem Kontainer. Entsprechend besitzt die zweite Untergruppe auch eine Mutterklasse namens `Container`. (Im Folgenden wird immer dann von einem Kontainer gesprochen, wenn es sich um eine Klasse handelt, die andere AWT-Komponenten beinhalten kann, ganz egal, ob es sich dabei z.B. um ein Fenster oder ein Panel handelt.) Durch das Kontainer-Prinzip lässt sich eine beliebig tief verschachtelte Hierarchie erreichen: Die oberste Komponente ist immer ein Kontainer (ge-

nauer: `Dialog`, `Frame` oder `Applet`), der mehrere Komponenten beinhalten kann, die ihrerseits wieder Kontainer sein können, die wiederum weitere Komponenten enthalten können, usw.

- Als davon unabhängig wird die zweite Kategorie um die Klasse `MenuComponent` und deren Derivate betrachtet. Diese Klassen stellen das sichtbare Menüsystem dar, dass manche Anwendungen bereitstellen.
- Die dritte Gruppe besteht aus den (teilweise abstrakten) Klassen `Font`, `Image`, `Graphics` und `Toolkit`. Diese stellen Schnittstellen zu verschiedenen Bereichen des darunterliegenden Grafiksystems dar: `Font` kapselt eine beliebige Bildschirmschriftart für die Verwendung unter der AWT, `Image` beinhaltet eine systeminterne Darstellung einer Grafik, die mit der AWT ausgegeben oder bearbeitet werden kann, und `Graphics` stellt eine Schnittstelle für grundlegende Zeichenoperationen zur Verfügung. Die Bedeutung von `Toolkit` wird später noch genauer erläutert.
- In die vierte und letzte Sparte fallen alle restlichen Klassen. Beispiele dafür sind z.B. die Layoutmanager, die einem Kontainer das gewünschte Layout geben, oder Klassen, die zur internen Darstellung von Rechtecken, Punkten, Polygonen, Farben, Rändern, o.ä. verwendet werden.

Da die ersten beiden erwähnten Gruppen offensichtlich auf das darunterliegende Betriebssystem samt seiner grafischen Oberfläche zugreifen müssen, sind es auch diese beiden Gruppen, für die entsprechende Peers entwickelt werden müssen, die den Zugriff regeln. Dazu sind von der AWT Interface-Klassen für jede Komponente vorgesehen worden, die der jeweilige Peer entsprechend implementieren muss.

Bei der dritten Gruppe ist der Zugriff auf das native System uneinheitlich geregelt: Während die Klasse `Font` ebenfalls ein Peer-Interface besitzt, welches ein nativer Peer implementieren muss, sind die anderen Klassen `Image`, `Graphics` und `Toolkit` als abstrakt definiert, d.h. es müssen davon eigene Klassen abgeleitet werden, die den Austausch mit dem Betriebssystem steuern.

Die restlichen Klassen zeichnen sich dadurch aus, dass sie komplett in Java geschrieben sind und somit keinerlei betriebssystemspezifische Anpassung benötigen. Sie stellen folglich den systemunabhängigen Teil der AWT dar.

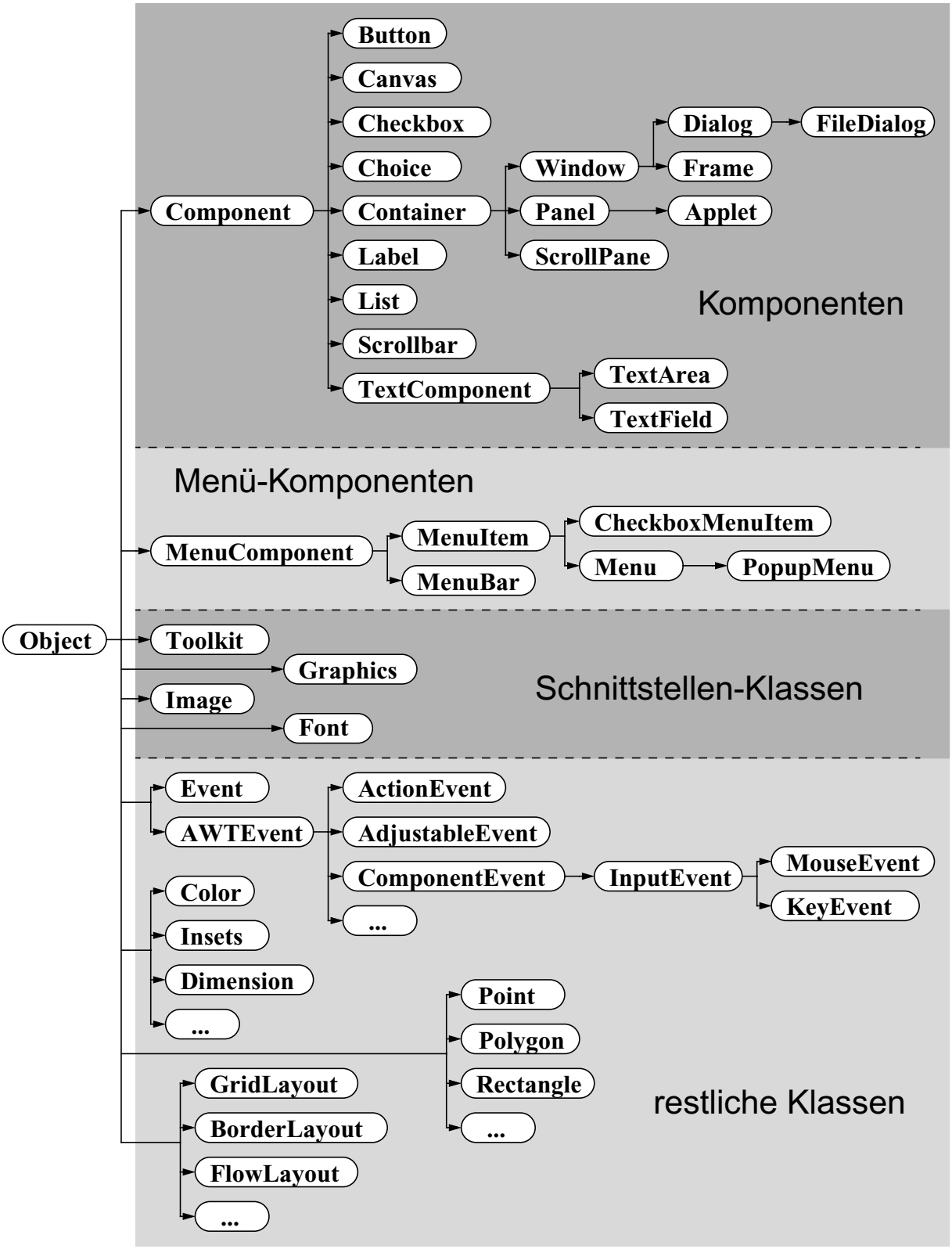


Abb. 2.4 Die Ableitungshierarchie der AWT (enthält nicht alle Klassen)

2.3.2 Verwaltung der Peers

Nach dieser kurzen Einteilung der Klassen stellt sich v.a. die Frage, wie die AWT die Erzeugung und die Kommunikation mit den notwendigen Peers regelt. Dazu wird folgendes Beispielprogramm einmal genauer untersucht:

```
import java.awt.*;
public class MyDemo {
    public static void main(String args[]) {
        Frame f = new Frame("A simple application");
        Button b = new Button("My Button");

        f.add(b);
        f.pack();
        f.show();
    }
}
```

Abb. 2.5 Ein einfaches Beispiel für eine AWT-Anwendung

Für den halbwegs geübten AWT-Programmierer sollte der Ablauf des Programmes klar ersichtlich sein: Die `main()`-Methode, die bekanntlich beim Laden einer Klasse gestartet wird, erzeugt zwei neue Objekte vom Typ `Frame` und `Button`, also ein Fenster mit dem Titel "A simple application" und einen Knopf mit der Aufschrift "My Button". Mittels `add()` wird der Knopf in das Fenster eingebunden, und durch `pack()` erstellt der Layoutmanager des Fensters ein passendes Layout für das Fenster, basierend auf den Komponenten, die das Fenster beinhaltet (momentan also nur den Knopf). Die Methode `show()` schließlich zeigt das gewünschte Ergebnis: Ein kleines Fenster, das einen einzelnen Knopf beherbergt, der nach Belieben gedrückt werden kann.

Abb. 2.6 Das fertige Fenster unter einer Windows-Plattform

Soweit der Einblick in die Benutzerschnittstelle. Nun geht es um die Frage, was im Inneren der AWT bei der Ausführung dieses Programms abläuft.

Die Erzeugung der Peers

Bei der Erzeugung der beiden AWT-Objekte werden diese erst einmal mit definierten Startwerten initialisiert. Der folgende Aufruf der Methode `add()` bewirkt, dass der Knopf in einer internen Verwaltungsstruktur des Fensters registriert wird. Der eigentlich relevante Aufruf findet mit `pack()` statt: Die Methode überprüft vor dem Layoutdurchgang, ob das entsprechende Objekt (im diesem Fall das Fenster) überhaupt einen Peer besitzt, und wenn nicht, startet sie den

Erzeugungsprozess durch Aufruf der Methode `addNotify()`. Diese Methode, die allen Komponenten vererbt ist, beginnt zuerst den Peer für das Fenster zu erstellen, läuft danach seine interne Verwaltungsstruktur durch und fordert alle registrierten Komponenten (hier nur den Knopf) mittels `addNotify()` auf, ebenfalls deren Peers zu erstellen. Letztendlich ist es also Aufgabe der Methode `addNotify()`, die entsprechenden Peers zu erzeugen.

Allgemein läuft der Erzeugungsprozess folgendermaßen ab: Gestartet wird normalerweise durch den ersten Aufruf der Methode `pack()`. Diese ruft dann die Methode `addNotify()` auf, die nach folgendem rekursiven Prinzip arbeitet:

- Der eigene Peer wird erstellt.
- Sollte das aktuelle Objekt von `Container` abgeleitet sein, so werden die `addNotify()`-Methoden der mittels `add()` registrierten Komponenten aufgerufen.

Durch diesen rekursiven Algorithmus wird sichergestellt, dass alle Komponenten ihre Peers einrichten, so dass nur noch die Frage übrigbleibt, wie ein Peer konkret eingerichtet wird. Hier kommt die Klasse `Toolkit` ins Spiel: Diese stellt die Schnittstelle dar zu allem, was von der AWT benötigt werden könnte, aber nicht betriebssystemunabhängig implementiert werden kann. Dazu zählen neben den Peers z.B. auch das Laden von Grafiken oder das Abfragen der Bildschirmauflösung. Diese abstrakte Klasse stellt u.a. für jede AWT-Komponente eine entsprechende Methode bereit, über die sie ihren zugehörigen Peer beziehen kann. Diese Methoden besitzen die allgemeine Form

```
protected abstract XXXPeer createXXX(XXX target);
```

, wobei `XXX` für die zugehörige Komponente steht, also z.B. `Button`, `Scrollbar` oder `Frame`. In der eigentlichen Klasse `Toolkit` sind dabei nur die Methodenrümpfe definiert; Wie die Methoden letztendlich arbeiten, bleibt der jeweiligen systemabhängigen Implementierung von `Toolkit` vorbehalten.

Im obigen Beispiel würde die Klasse `Button` also den Peer mittels des Methodenaufrufs `createButton(this)` erhalten, welcher dann ein Objekt des Typs `ButtonPeer` zurückliefert. Dieser zurückgegebene Objekttyp ist ein Interface, das bestimmte Methoden definiert, welche der zurückgelieferte Peer implementiert. Durch die Übergabe der eigenen Referenz an die Methode wird sichergestellt, dass der erstellte Peer seine zugehörige Komponente kennt, so dass die Komponente auch von Änderungen informiert werden kann, die im Peer durch Ereignisse des grafischen Subsystems stattgefunden haben.

Hier tritt zum ersten Mal ein wichtiger Punkt auf, der noch an einigen Stellen von Bedeutung sein wird: Wenn ein Peer erstellt wird, kennt er nur seine zugehörige Java-Komponente. Das heißt, er kennt weder andere Java-Komponenten, noch kennt er andere Peers! Das kann problematisch sein, da das Betriebssystem vielleicht nicht nur einfach GUI-Elemente frei zur Verfügung stellt, sondern auch eine eigene Hierarchie dieser Elemente verwalten möchte. Und es ist insofern von Nachteil, als ein Peer z.B. dann gar nicht wissen kann, welchem Fenster er angehört, und wohin er sich zeichnen darf (außer natürlich, er ist der Peer für das Fenster selbst). Diese Problematik lässt sich jedoch lösen: Jeder Peer kennt seine Java-Komponente, und jede Komponente besitzt eine Referenz auf den sie beinhaltenden Kontainer (die oberste Komponen-

te der Hierarchie, z.B. ein Fenster, besitzt hier eine Null-Referenz). Durch das Verfolgen dieser Referenzen kommt man schließlich zum obersten Element der Hierarchie, und von da aus könnte man wieder rekursiv den ganzen Hierarchiebaum durchsuchen. Somit ließe sich feststellen, wie der neue Peer eingeordnet ist, und diese Information sich auch an das darunterliegende System weiterleiten. Abbildung 2.7 zeigt anhand einer Beispielskizze, welchen "Bekanntheitsgrad" die einzelnen Objekte in der AWT-Hierarchie besitzen.

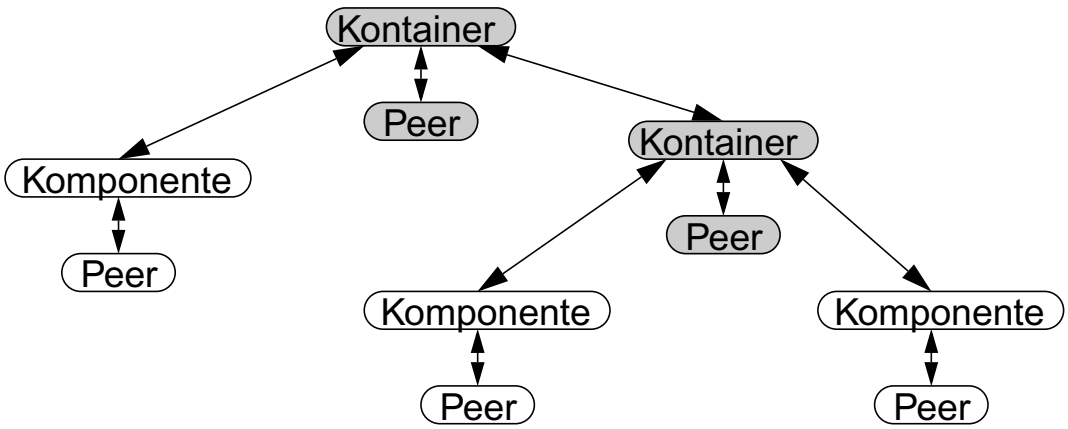


Abb. 2.7 Der Bekanntheitsgrad von Objekten in einer AWT-Hierarchie

Kommunikation mit den Peers

Jede Komponente besitzt Methoden, die der Anwender benutzt, um die Komponente seinen Wünschen entsprechend gestalten oder steuern zu können; Beispiele hierfür sind `setBounds()`, `setEnabled()` oder `show()`. Viele dieser Methoden rufen im Endeffekt nur entsprechende Methoden der darunterliegenden Peers auf, um das Betriebssystem anzuweisen, die entsprechenden Änderungen durchzuführen. Umgekehrt hat auch der Peer die Möglichkeit, Methoden seiner zugehörigen Java-Komponente aufzurufen und diese ebenfalls von Änderungen auf Betriebssystemebene zu informieren. Im Normalfall sollte ein Peer von dieser Art der Kommunikation aber keinen Gebrauch machen, da die wenigsten Methoden der Komponenten für die Benutzung der zugehörigen Peers ausgelegt sind, und es dabei leicht zu Feedbacks bzw. Endlosschleifen kommen kann.

Die übliche Methode der Peers, mit der AWT zu kommunizieren, besteht darin, dieser Nachrichten zu schicken, und sie damit über aufgetretene Ereignisse zu informieren. Je nach verwendetem Ereignismodell werden dabei entweder die entsprechenden Handlermethoden der Ziel-Komponente direkt aufgerufen, oder ein entsprechendes Ereignis in die Ereigniswarteschlange eingeschoben (vgl. Abschnitt 2.2.4).

2.3.3 Zeichnen einer Komponente

Wie in Abschnitt 2.2.5 bereits erwähnt, erlaubt die AWT dem Entwickler, neben den vom Betriebssystem vorgegebenen Komponenten eigene zu entwickeln bzw. das Aussehen der vorhandenen Komponenten in gewissen Grenzen zu manipulieren. Die meisten Modifikationen an den Komponenten sind dabei nur auf Ebene der AWT durch Übermalen mit benutzerspezifischem Zeichencode möglich, da das darunterliegende Betriebssystem im Normalfall keine direkte Manipulation des Aussehens der grafischen Elemente erlaubt. Damit diese Modifikationen, die auf Komponentenebene vorgenommen wurden, auch immer sichtbar sind, d.h. der Peer vom Betriebssystem nicht einfach mit dem Standarddesign übermalt wird, muss die AWT angewiesen werden können, bei einem Neuzeichnen des Peers die Modifikationen erneut auf diesen anzuwenden. Dies ist vor allem bei der Komponente `Canvas` von Bedeutung, da diese eine Zeichenfläche für benutzerspezifischen Zeichencode darstellt, und dementsprechend von der korrekten Ausführung dieses Zeichencodes abhängig ist.

Im Normalfall geschieht das Zeichnen einer Komponente selbstständig, d.h. das Pendant auf Betriebssystemebene zeichnet sich wann immer nötig selbst. Um dabei auch das Neuzeichnen des benutzerspezifischen Teils zu gewährleisten, sendet der Peer ein sogenanntes “Paint-Event” (sofern er das benutzerspezifische Zeichnen unterstützt), das in die Ereigniswarteschlange eingetragen wird. Dieses Ereignis enthält Informationen über die Komponente und deren Ausschnitt, der gezeichnet werden soll. Erreicht ein solches Ereignis eine Komponente, so wird deren `paint()`-Methode aufgerufen, welche den benutzerdefinierten Teil zeichnet.

Die `paint()`-Methode erhält ein `Graphics`-Objekt als Parameter, welches die notwendigen Zeichenoperationen zur Verfügung stellt. Dieses Objekt wird dafür kurz zuvor mittels der `getGraphics()`-Methode der Komponente erstellt. Die Methode ist dabei folgendermaßen implementiert (vgl. Abbildung 2.8):

- Falls kein Peer existiert, wird eine Null-Referenz zurückgeliefert.
- Falls er existiert, wird dessen Methode `getGraphics()` aufgerufen. Liefert diese ein gültiges Objekt, wird es zurückgegeben.
- Wird eine Null-Referenz geliefert, so wird der “Vater” der Komponente gesucht, also der Kontainer, der die Komponente beinhaltet. Ist keiner vorhanden, so wird eine Null-Referenz zurückgeliefert.
- Ist der Vater vorhanden, so laufen folgende Schritte ab:
  - Die Methode `getGraphics()` des Vaters wird aufgerufen.
  - Das zurückgelieferte `Graphics`-Objekt der Methode wird so eingerichtet, dass der Clipbereich dem Bereich dieser Komponente und der Koordinatenursprung der linken oberen Ecke dieser Komponente entspricht.
  - Das Objekt wird zurückgegeben.

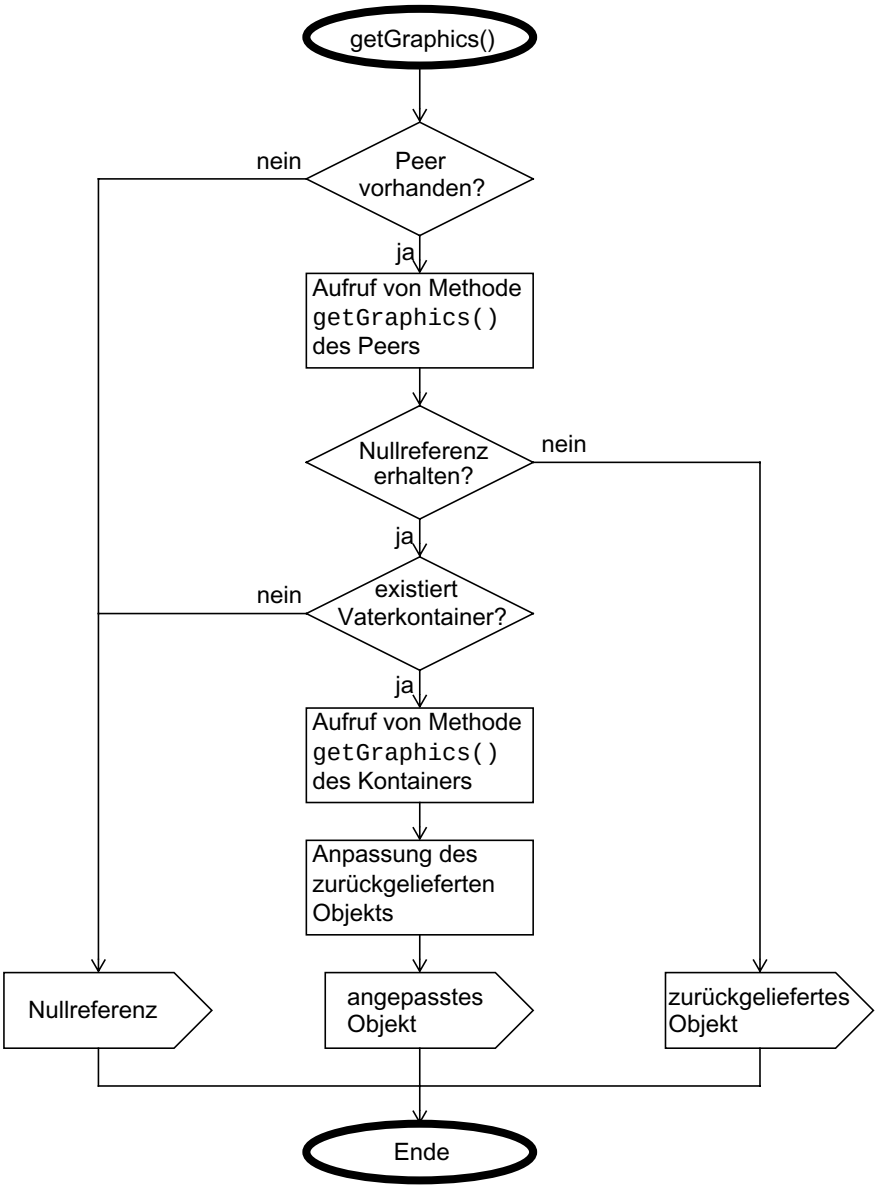


Abb. 2.8 Die Methode getGraphics()

Dieser rekursive Algorithmus erlaubt es, dass nicht alle vorhandenen Peers eine Möglichkeit kennen müssen, ein `Graphics`-Objekt zu erzeugen, solange zumindest das oberste Element in der Komponentenhierarchie ein solches Objekt erzeugen kann. Der Algorithmus wandert den Hierarchiebaum solange nach oben, bis er ein gültiges `Graphics`-Objekt erhält, und beim Herabsteigen wird jeweils der Clipbereich und der Koordinatenursprung des Objekts angepasst, so dass jede Komponente ein für sie maßgeschneidertes `Graphics`-Objekt bekommt.

Neben der Möglichkeit, dass eine Komponente durch ein Paint-Event neu gezeichnet wird, kann man jede Komponente auch direkt anweisen, sich selbst neu zu zeichnen. Dies geschieht mit Hilfe der `repaint()`-Methode. Je nach Belieben kann man mit dieser Methode die ganze Komponente oder nur einen Teil davon zeichnen, sowie den Zeitpunkt angeben, wann dies geschehen soll.

2.3.4 Verwaltung des Tastaturfokus

Um einen Tastendruck einer bestimmten Komponente zuzuordnen, wird der sogenannte Tastaturfokus verwendet. Nur diejenige Komponente, welche den Tastaturfokus besitzt, empfängt Tastaturevents von der darunterliegenden grafischen Oberfläche. Welche Komponente gerade den Fokus hat, ist normalerweise an der besonderen Zeichnung der Komponente zu erkennen.

Die AWT bietet bereits auf Komponentenebene einige Methoden zur Unterstützung eines Tastaturfokus an, die jede Komponente kennt. Zum einen ist da die Methode `isFocusTraversable()`, die für jede Komponente angibt, ob diese den Fokus bekommen kann oder nicht. Neben dieser Methode, die sich der gleichnamigen Methode des Peers bedient, sind vor allem zwei weitere Methoden interessant: `transferFocus()` und `requestFocus()`.

Die Methode `transferFocus()` ist vollständig auf der AWT aufgebaut und dafür zuständig, den Nachfolger dieser Komponente bezüglich des Tastaturfokus zu finden (d.h. die nächste Komponente, die den Fokus bekommen kann) und dieser dann den Fokus zu übertragen. Dazu bedient sie sich der Methode `requestFocus()`, welche der zugehörigen Komponente den Fokus überträgt. Diese Methode benutzt dazu wiederum die gleichnamige Methode des Peers. Somit ist das “wie” und “wann” des Fokustransfers auf Peer-Ebene zu regeln.

2.4 Der Windowmanager von JX

Der Windowmanager basiert auf einer Studienarbeit von Jürgen Obernolte und stellt eine flexible und einfach zu handhabende Bibliothek dar, die es ermöglicht, unter JX Fenster zu erzeugen und zu verwalten, diese mittels Maus zu verschieben und in der Größe zu verändern, sowie deren Inhalt durch bereitgestellte Methoden frei zu gestalten. Die zum Zeitpunkt dieser Arbeit verfügbare Version arbeitete zwar noch ohne Double-Buffering und ohne besondere Hardwarebeschleunigung durch den darunterliegenden Grafiktreiber, ermöglichte aber dennoch ein flüssiges Arbeiten mit dem System und brauchte sich vom Leistungspotential her nicht hinter anderen, kommerziellen Lösungen zu verstecken.

Wie die meisten anderen Betriebssystemkomponenten von JX ist auch der Windowmanager zu 100% in Java geschrieben. Er besitzt eine große Anzahl von Klassen, von denen allerdings nur eine Handvoll für den Benutzer wichtig sind.

2.4.1 Erzeugung eines Fensters

Die wichtigste Klasse ist `WWindow`, da diese die eigentliche Schnittstelle zum Windowmanager implementiert. Jede Instanz dieser Klasse stellt ein eigenes Fenster dar, das über die bereitgestellten Methoden von `WWindow` beliebig gesteuert werden kann. Um ein neues Fenster zu erzeugen, ist es ausreichend, eine neue Instanz von `WWindow` zu erzeugen und dessen `show()`-Methode aufzurufen. Danach besitzt man ein Fenster, das sich mit der Maus mühelos verschieben, vergrößern und verkleinern lässt.

2.4.2 Reaktion auf Ereignisse

In der Klasse `WWindow` sind verschiedene Methoden zur Ereignissteuerung vorhanden, die vom Windowmanager aufgerufen werden, wenn ein bestimmtes Ereignis eintritt, z.B. wenn ein Fenster aktiv, eine Taste gedrückt oder mit der Maus verschoben wird (vgl. das 1.0-Ereignismodell der AWT). Durch Überschreiben dieser Methoden kann die Anwendung auf das jeweilige Ereignis wie gewünscht reagieren.

2.4.3 Zeichnen in das Fenster

Auch zum Zeichnen sind verschiedene Methoden in der Klasse `WWindow` vorhanden. Momentan erlaubt die Klasse das Zeichnen von Linien, gefüllten und leeren Rechtecken, Grafiken sowie von Texten. Daneben werden noch unterschiedliche Farbmodelle (8-bit, 16-bit, 32-bit) sowie verschiedene Zeichenmodi (normal oder XOR) unterstützt. Außerdem ist es möglich, einen Clipping-Bereich anzugeben.

Der Aufruf an das Fenster, sich selbst zu zeichnen, geschieht ähnlich wie in der AWT: Falls ein Fenster nach Meinung des Windowmanagers neu gezeichnet werden muss, wird die `paint()`-Methode des entsprechenden Fensters aufgerufen, die als Parameter den neu zu zeichnenden Bereich übergeben bekommt. Da ein Fenster seine eigenen Zeichenmethoden besitzt, ist es hier im Gegensatz zur AWT nicht notwendig, eine Art `Graphics`-Objekt zu übergeben.

Es ist natürlich auch auf Anwenderseite möglich, das Fenster anzuweisen, etwas zu zeichnen, indem einfach entsprechende Zeichenoperationen aufgerufen werden. Sinnvoller ist es jedoch, diese Operationen in der `paint()`-Methode durchführen zu lassen, ansonsten sind die sichtbaren Ergebnisse nach dem nächsten Update des Windowmanagers verschwunden.

2.5 Zusammenfassung

Das eigentliche Ziel der Arbeit wurde genauer herausgearbeitet, nämlich eine Verbindungsschicht zwischen der non-peer Classpath AWT und dem Windowmanager von JX aufzubauen. Daneben wurde die AWT vorgestellt, sowohl ihre allgemeinen Konzepte, als auch deren konkrete Implementierung durch die Classpath AWT. Dabei lag ein Schwerpunkt auf dem Konzept der Peers und seiner Bedeutung für die AWT. Zum Abschluss wurde auf den Windowmanager eingegangen, der die grundlegende Schicht für diese Arbeit bildet.

3 Konzepte

3.1 Überblick

Dieses Kapitel beschreibt die Konzepte, die der Implementierung der Peers zugrundeliegen.

3.2 Die grundlegende Hierarchie: Peers und Konnektoren

Bei der Gestaltung der einzelnen Peers wurde versucht, die Ableitungseigenschaften der Sprache Java so gut wie möglich zu nutzen, um möglichst wenig Code mehrmals schreiben zu müssen. Daher schien es am sinnvollsten, die Hierarchie der Peers analog zu der der AWT-Komponenten aufzubauen, da diese in Hinblick auf Ökonomie der Methoden bereits sehr effizient gestaltet worden ist. Ein weiterer Grund für die Analogie der Hierarchien lag in der Tatsache, dass jede der AWT-Komponenten, auch die abstrakten Klassen, einen eigenen Peer besitzen kann. So lag es auf der Hand, für jede Komponente einen eigenen Peer zu erstellen und gleichzeitig die Ableitungshierarchie beizubehalten. Abbildung 3.1 zeigt die Ableitungshierarchie der Peers an einem Beispiel.

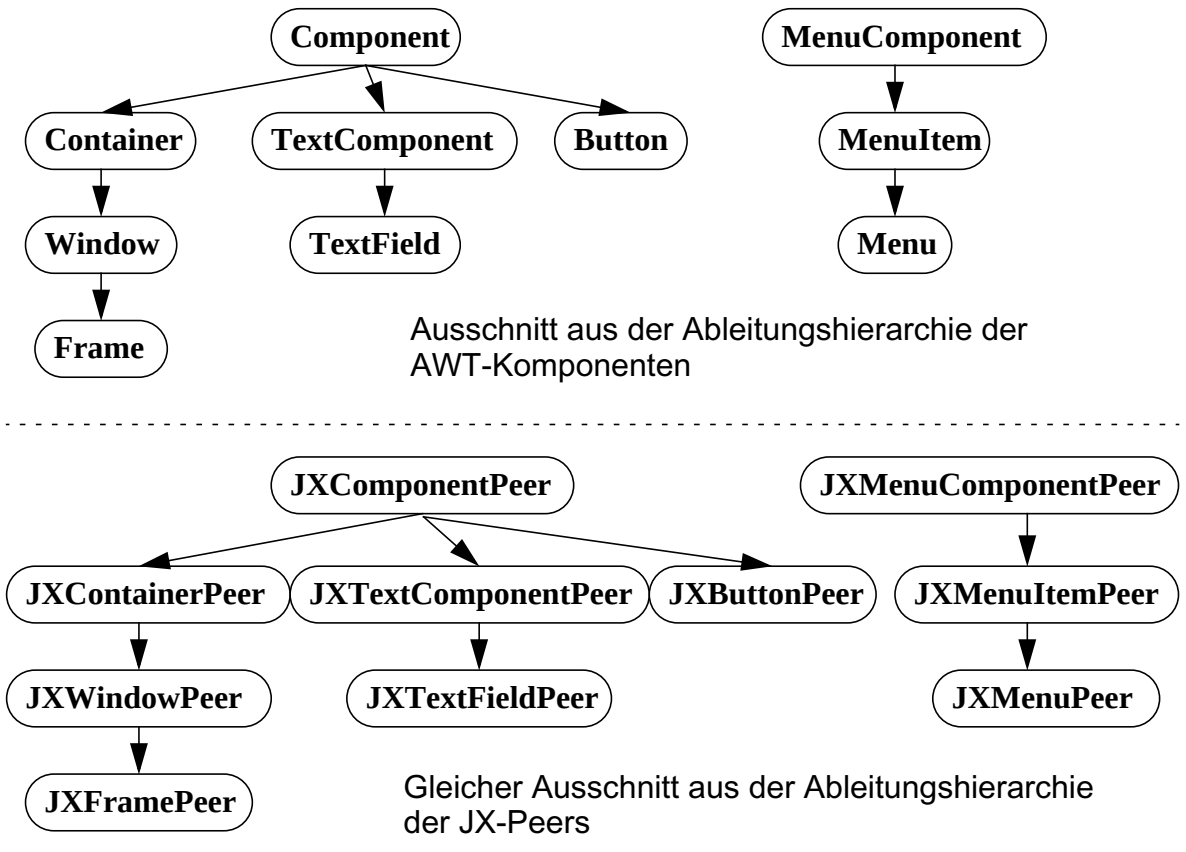


Abb. 3.1 Ein beispielhafter Vergleich beider Ableitungshierarchien

Ein Peer kennt zum Zeitpunkt seiner Erzeugung nur seine zugehörige AWT-Komponente. Es wäre zwar möglich, durch ein aufwendiges Suchverfahren alles über die anderen AWT-Komponenten und deren Peers herauszufinden, jedoch wurde in der vorliegenden Implementierung darauf verzichtet, da einerseits der Aufwand der Aktualisierung dieser Informationen viel zu groß wäre, und andererseits auch vollkommen unnötig ist. Tatsächlich ist es in dieser Implementierung für die einzelnen Peers nicht notwendig, sich untereinander zu kennen oder miteinander zu kommunizieren. Die einzige Ausnahme bildet hier der Algorithmus zum Zeichnen der Peers, der noch weiter unten besprochen wird.

Darüber hinaus gibt es in der vorliegenden Implementierung zweierlei Arten von Peers: Solche, die nur Platz in bestehenden Fenstern benötigen, und solche, die selbst Fenster repräsentieren bzw. zur Verfügung stellen. Letztere müssen sich der vom Windowmanager zur Verfügung gestellten `WWindow`-Klasse bedienen, um ihre Fenster darstellen und verwalten zu können. Der besseren Übersicht und Wartbarkeit wegen wurden für diese Peers die sogenannten Konnektoren geschaffen, die von der Klasse `GeneralConnector` abgeleitet sind (vgl. Abbildung 3.2), welche wiederum von `WWindow` abgeleitet ist. Jeder Peer, der ein eigenes Fenster benötigt, z.B. für ein Menü, verfügt über eine Instanz des zugehörigen Konnektors und kann über ihn auf die Fähigkeiten des Windowmanagers zugreifen.

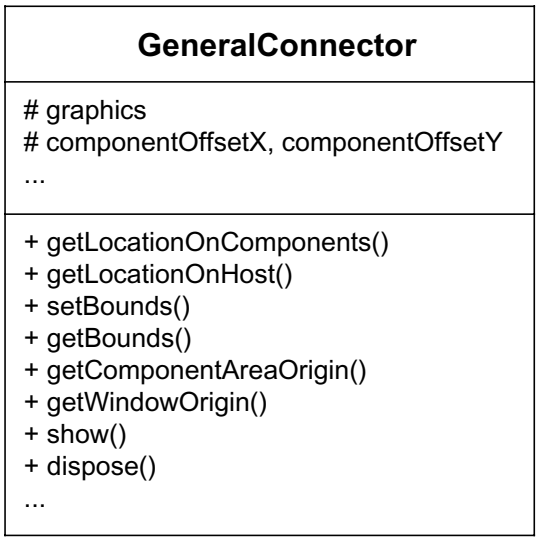


Abb. 3.2 Die Klasse GeneralConnector

Diese Konnektoren bilden damit die Schnittstelle zum Windowmanager und erlauben es, über ihre geerbten Fähigkeiten ein Fenster zu erzeugen und zu verwalten. Durch diese Trennung in Peer und Konnektor lassen sich einige Ressourcen sparen: Genauso wie eine Komponente ihren Peer erst erzeugt, wenn er zum ersten Mal dargestellt werden soll, so erzeugt ein Peer seinen Konnektor auch erst dann, wenn er benötigt wird. Auf diese Weise wird der Windowmanager nicht mit der Verwaltung von unsichtbaren und unnötigen Fenstern aufgehalten. Darüber hinaus lassen sich so die AWT- oder Windowmanager-spezifischen Teile der JX-Implementierung voneinander getrennt warten und erweitern.

Somit ergibt sich für das interne Zusammenspiel der einzelnen Schichten untereinander ein Bild, wie in Abbildung 3.3 skizziert. Man erkennt auch hier, dass jeder Peer nur seine zugehörige AWT-Komponente kennt, und seinen zugehörigen Konnektor, falls er einen besitzt.

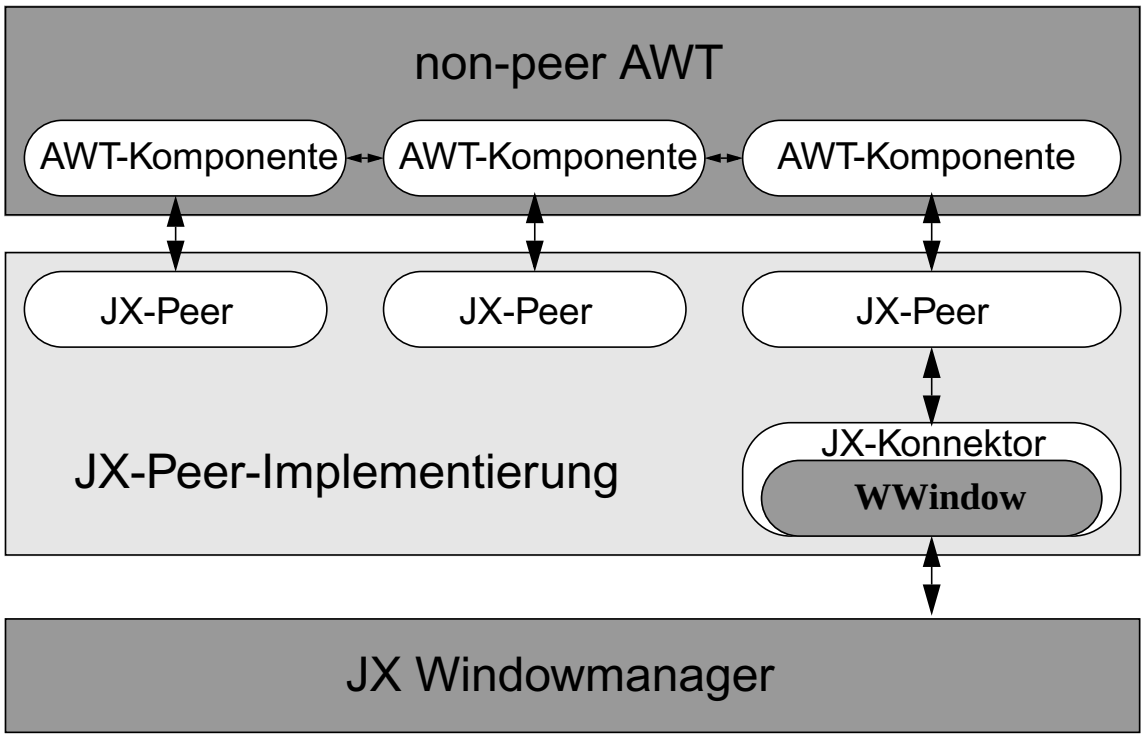


Abb. 3.3 Der interne Programmfluss zwischen den drei Schichten

3.3 Interaktion mit den Peers

Ein Peer kann von zwei Seiten angesprochen werden: Einerseits von der Komponentenebene, andererseits von Seiten des Windowmanagers. Die Komponentenebene kommuniziert normalerweise dann mit dem Peer, wenn es der Programmcode so erfordert, d.h. die Kommunikation erfolgt vorherbestimmt durch den Java-Code. Der Windowmanager dagegen spricht den Peer nur an, wenn bestimmte Ereignisse aufgetreten sind, über die der Peer informiert werden muss. Dessen Kommunikation erfolgt quasi “zufällig”.

Die Komponentenebene kommuniziert mit der Peer-Schicht durch die Peer-Interfaces der AWT, der Windowmanager dagegen durch die Handlermethoden der Klasse `WWindow`. Je nachdem, welche der beiden Schichten mit der Peer-Schicht kommuniziert, werden die ankommenden Ereignisse unterschiedlich behandelt:

Trifft ein Ereignis von Seiten der AWT ein, so erreicht dieses direkt den betroffenen Peer. Dieser kann das Ereignis intern speichern, weiterverarbeiten und, falls notwendig (wenn es sich z.B. um eine Größenänderung eines Fensters handelt), seinen zugehörigen Konnektor entsprechend informieren. Kommt das Ereignis dagegen vom Windowmanager, so erreicht es als erstes den

Konnektor, in dessen Fenster das Ereignis aufgetreten ist. Dieser verarbeitet das Ereignis intern, und leitet es gegebenfalls an den Peer weiter, für den das Ereignis bestimmt ist. Dieser Peer, der nicht notwendigerweise der zugehörige Peer des Konnektors sein muss, verarbeitet das Ereignis noch einmal für sich, und sendet evtl. ein AWT-Event an die Komponentenschicht (vgl. Abbildung 3.4).

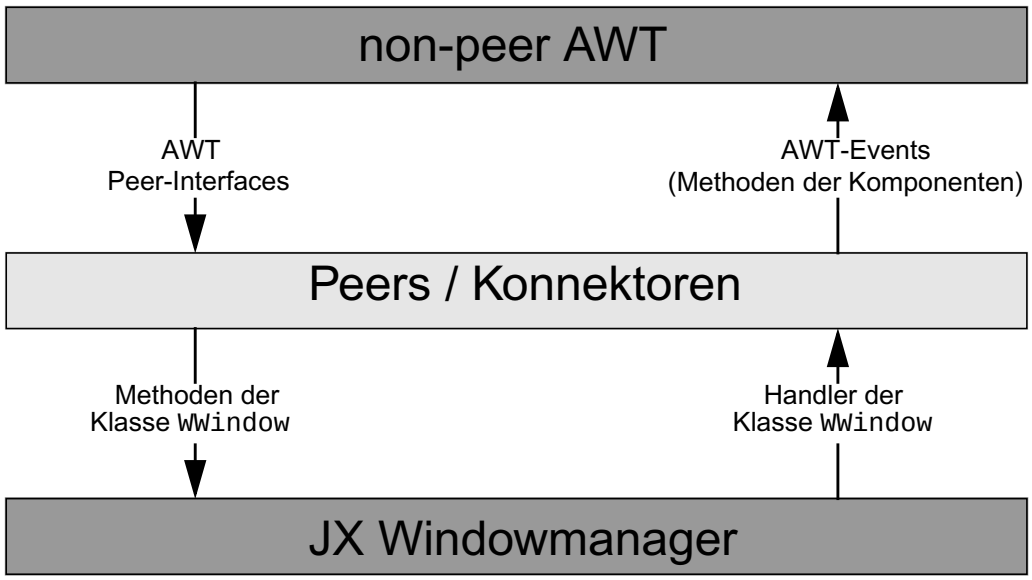


Abb. 3.4 Interaktion der Peers/Konnektoren mit den anderen Schichten

Wie gerade erläutert wurde, müssen die Peers auch in der Lage sein, Ereignisse an die AWT oder den Windowmanager weiterzuleiten, damit diese darauf reagieren können. Bei der AWT bedienen sich die Peers zu diesem Zweck der durch die AWT eingeführten Ereigniswarteschlange. Hier schieben die Peers die jeweiligen Events hinein, so dass der Verteiler-Thread der AWT diese an die entsprechenden Komponenten bzw. deren Eventlistener weiterleiten kann. Es ist für die Peers auch möglich, mit einzelnen Komponenten zu kommunizieren, indem sie deren Methoden aufrufen, jedoch geschieht dies nur in seltenen Fällen, etwa wenn sich ein neu erzeugter Peer initialisiert.

Mit dem Windowmanager kommunizieren die Peers durch die Schnittstelle, welche die Konnektoren von der Windowmanager-Klasse `WWindow` geerbt haben. Durch diese Schnittstelle lassen sich alle Eigenschaften eines Fensters wie Größe, Position oder andere Attribute einstellen.

3.4 Sonstige Konzepte

Der folgende Abschnitt beinhaltet eine Reihe weniger offensichtlicher Konzepte, die in der vorliegenden Implementierung zu finden sind. Bei vielen von ihnen handelt es sich nicht um neue Ideen, sondern einfach um Fortführungen von Konzepten aus der Komponentenebene der AWT. Das Fortsetzen dieser Konzepte war teils sinnvoll, teils unvermeidlich.

3.4.1 Das Zeichnen eines Peers

Das Zeichnen eines Peers ist insofern problematisch, als der Peer wissen muss, in welchem Fenster er gezeichnet werden soll, und dementsprechend auch, wie er sich selbst zeichnen soll, da er selbst keine Methoden zum Zeichnen kennt. Dieses Problem wird ganz analog zur Problematik des Neuzeichnens der AWT-Komponenten gelöst, d.h. durch Einführung einer eigenen Klasse, welche die notwendigen Zeichenoperationen zur Verfügung stellt (vgl. Abschnitt 2.3.3).

Ein Objekt einer solchen Klasse kann anfangs nur ein Konnektor besitzen, da nur er direkten Zugriff auf die Zeichenfläche seines Fensters hat. Das bedeutet, dass jeder Peer, der sich selbst zeichnen möchte, zuerst ein solches Objekt bei dem Konnektor seines Mutterfensters holen muss. Durch einen Algorithmus, der noch in Abschnitt 4.2.2 erläutert wird, bekommt er eine angepasste Kopie des Objekts, welche es ihm ermöglicht, Zeichenoperationen durchzuführen.

3.4.2 Strikte Trennung von Menü- und Zeichenbereich

Die AWT beinhaltet eine strikte Trennung zwischen Komponenten, die den Menübereich ausmachen, und denen, welche die restliche Fläche eines Fensters für sich beanspruchen. Dieses Zwei-Klassen-Konzept setzt sich bis zu den entsprechenden Peer-Interfaces durch, und wird deswegen von der vorliegenden Implementierung der Peers fortgesetzt: Die Peers für “normale” Komponenten und für Menü-Komponenten unterscheiden sich grundlegend in ihrer Implementierung. Die genauen Unterschiede werden im Abschnitt 4.6 deutlich.

3.4.3 Globale Objekte

Die Klasse `Toolkit` der AWT stellt die Schnittstelle zum betriebssystemabhängigen Teil der AWT dar. Daneben zeichnet sie sich gegenüber den anderen Klassen noch durch ein weiteres Detail aus: Während der gesamten Laufzeit wird von ihr nur eine einzige Instanz verwendet. Um dieses zu gewährleisten, und auch um diese Instanz ansprechen zu können, besitzt die Klasse eine statische Methode namens `getDefaultToolkit()`, die beim ersten Aufruf eine neue Instanz einer auf das darunterliegende Betriebssystem zugeschnittenen `Toolkit`-Klasse erstellt und zurückgibt. Eine interne Referenz verweist auf diese Instanz, so dass weitere Aufrufe der Methode `getDefaultToolkit()` nur diese zurückliefern.

Diese Auszeichnung ermöglicht es, die eigene Implementierung der Klasse `Toolkit` neben ihrer gewohnten Funktion als System-Schnittstelle noch als Aufhängepunkt für andere globale Instanzen zu verwenden, d.h. um andere Objekte zu referenzieren, die einzigartig in der AWT sind.

Bei diesen Objekten handelt es sich meist um Handler, die einen globalen Zustand in der AWT-Implementierung speichern und verwalten müssen, wie z.B. die Klasse `FocusHandler` (vgl. nächsten Abschnitt). Um diese Objekte ebenfalls ansprechen zu können, wurden in der eigenen `Toolkit`-Implementierung noch Schnittstellen für diese Objekte geschaffen.

3.4.4 Unterstützung des Tastaturfokus

Auch das Konzept des Tastaturfokus wird von den Peers implementiert: Peers können die AWT informieren, ob sie in der Lage sind, einen Fokus zu erhalten, und sich entsprechend zeichnen. Daneben können sie den Fokus für sich anfordern.

Die vorliegende Implementierung arbeitet nach dem Prinzip, dass immer nur eine Komponente zur gleichen Zeit den Fokus besitzen bzw. sich auch dementsprechend kennzeichnen kann. Eine solche Komponente kann es nur im aktiven Fenster geben; Ist kein Fenster aktiviert, so hat auch keine Komponente den Fokus.

Auf Peer-Ebene wurde dafür eine globale Verwaltungsstruktur in der Klasse `FocusHandler` geschaffen, welche immer eine Referenz auf die aktuell fokussierte Komponente hält. Bei dieser Klasse handelt es sich um eines der gerade erwähnten “globalen Objekte”: Da diese Klasse eine einzige Referenz für alle Fenster halten muss, darf es auch nur eine einzige Instanz davon geben, um Inkonsistenzen zu vermeiden.

3.5 Zusammenfassung

Es wurden die wichtigsten Konzepte erläutert, auf denen die vorliegende Implementierung der AWT für JX aufbaut. Es wurde auf die Unterteilung der implementierten Klassen in Peers und Konnektoren eingegangen, und es wurde die Art und Weise skizziert, wie die Peers mit den beiden anderen Schichten interagieren. Zum Schluss wurden noch weitere Konzepte beschrieben, die in der Implementierung realisiert sind.

4 Details & Implementierung

4.1 Überblick

Dieses Kapitel behandelt die Implementierung der Peers und aller sonstigen Klassen, die für eine funktionsfähige Version der AWT notwendig waren. Dabei werden zuerst die grundlegenden Klassen behandelt, anschließend werden die Peers und Konnektoren beschrieben, welche für die Realisierung von AWT-Komponenten, dem Menüsystem sowie dem Fenstersystem implementiert wurden. Danach wird noch auf die Realisierung der Unterstützung von Grafiken eingegangen. Abschließend werden die zusätzlichen Erweiterungen des AWT-Standards erläutert.

4.2 Grundlegende Klassen

4.2.1 Die Peer-Schnittstelle: Die Klasse JXToolkit

Um eine Schnittstelle zu Systemfunktionen und den eigenen Peers zu erhalten, war es notwendig, eine eigene Implementierung der Klasse `Toolkit` zu schaffen. Diese Klasse namens `JXToolkit` ist u.a. für die Erzeugung der nötigen Peers verantwortlich (Abbildung 4.1 zeigt die Klasse im Überblick). Die dafür notwendigen Methoden sind alle nach dem folgenden Schema implementiert:

```
protected yyyPeer createyyy(java.awt.yyy target) {
        return new JXyyyPeer(target, this);
    }
```

, wobei `yyy` wiederum für die jeweilige Komponente steht, wie z.B. `Button`, `Panel` oder `Frame` (die zugehörigen Peer-Klassen heißen somit `JXButtonPeer`, `JXPanelPeer` oder `JXFramePeer`). Die jeweilige Methode erzeugt einfach eine neue Instanz des zugehörigen JX-Peers, übergibt ihm dabei eine Referenz auf sich selbst sowie auf die zugehörige AWT-Komponente, und gibt den neuen Peer zurück.

Neben diesen Methoden, welche die `JXToolkit`-Klasse implementieren muss, bietet sie auch eigene, implementierungsspezifische Erweiterungen an. Da von dieser Klasse zur Laufzeit nur ein einziges globales Objekt vorhanden ist, bot es sich an, hier auch Referenzen für weitere, global angelegte Klassen zu schaffen. Es handelt sich dabei um folgende Klassen, die später noch in den jeweiligen Kapiteln genauer erläutert werden:

- `JXMenuThread`
- `MenuHandler`
- `FocusHandler`

- `SlaveWindowHandler`

Allen gemein ist nur die Tatsache, dass auch diese Klassen nur einmal global instanziiert werden dürfen. Dies wird von der Klasse `JXToolkit` erledigt, und diese bietet auch Methoden zur Referenzierung dieser Klassen an.

Als letzte eigene Erweiterung erlaubt es die Klasse, mittels der Methode `loadAlternateColors()`, ein neues Farbschema für die einzelnen Peers zu laden. Dazu wird die Klasse `JXColors` verwendet, die in Abschnitt 4.8.2 noch genauer erläutert wird.

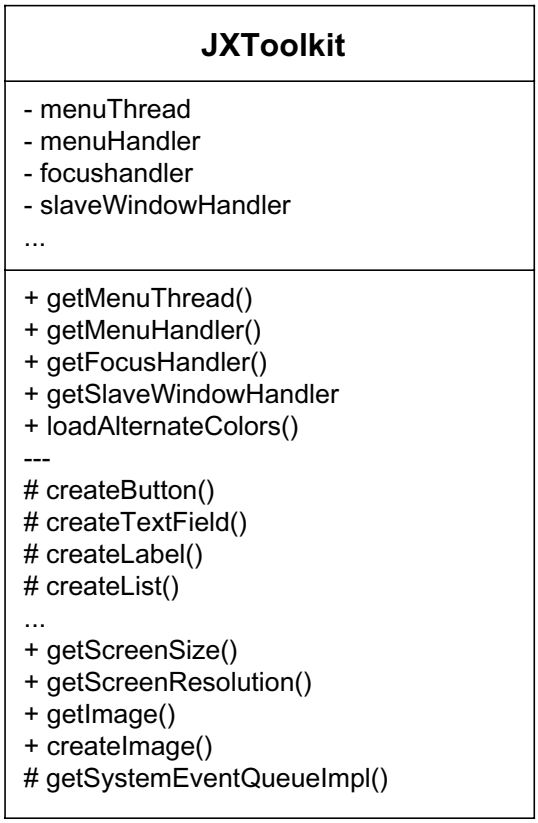


Abb. 4.1 Die Klasse JXToolkit

4.2.2 Das Zeichnen eines Peers: Die Klasse JXGraphics

Wie in Abschnitt 3.4.1 beschrieben, verwendet ein Peer zum Zeichnen ein Objekt einer Klasse, welche speziell für diesen Zweck erstellt wurde. Diese Klasse hat den Namen `JXGraphics`. Jeder Konnektor besitzt eine Referenz auf eine eigene Instanz einer solchen Klasse, und gibt bei Anfrage eine Kopie davon zurück. Diese Instanz des `JXGraphics`-Objekts sowie deren Kopien besitzen selbst wieder eine Referenz auf den zugehörigen Konnektor. Auf diese Weise können, da jeder Konnektor die notwendigen Zeichenoperationen für sein eigenes Fenster beherrscht, Zeichenoperationen einer `JXGraphics`-Klasse in Zeichenoperationen des zugehörigen Konnektors umgesetzt werden. Abbildung 4.2 zeigt die Klasse im Überblick.

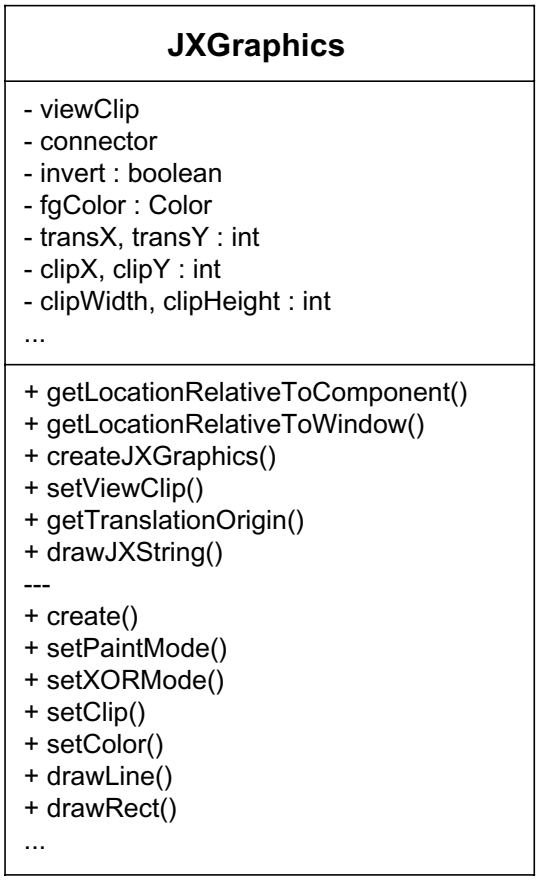


Abb. 4.2 Die Klasse JXGraphics

Die Klasse `JXGraphics` ist von der AWT-Klasse `Graphics` abgeleitet, so dass sie ohne weiteres auch für das Zeichnen von benutzerspezifischen Teilen der Komponenten verwendet werden kann. Dementsprechend sind auch die verwendeten Befehle zum Zeichnen eines Peers mit denen zum Zeichnen des benutzerspezifischen Teils einer Komponente identisch. Auch der Erhalt einer solchen Klasse erfolgt ähnlich dem Verfahren aus Abschnitt 2.3.3: Um eine gültige Instanz dieser Klasse zu erhalten, wird die Methode `getJXGraphics()` eines Peers aufgerufen. Diese Methode arbeitet nach folgendem Algorithmus, der in Abbildung 4.3 nochmals dargestellt wird:

– Zuerst wird die zugehörige Komponente geprüft: Falls sie nicht von der Klasse `Frame` abstammt, wird deren Vaterkontainer gesucht und getestet, usw., bis die Instanz einer Klasse vom Typ `Frame` gefunden wurde, die den aufrufenden Peer bzw. seine zugehörige Komponente beherbergt.

– Von dieser Instanz der Klasse `Frame` wird der Peer gesucht und an diesem die Methode `getGraphics()` aufgerufen. Diese wiederum ruft die Methode `getComponent-Graphics()` seines Konnektors auf (ein Peer der Komponente `Frame` muss einen Konnektor besitzen!).

– Der Konnektor erstellt daraufhin eine neue Kopie des `JXGraphics`-Objekts für dieses Fenster und gibt es nach oben zurück. Diese Kopie erreicht schließlich den Peer, dessen `getJXGraphics()`-Methode aufgerufen wurde.

– Im letzten Schritt werden noch der Koordinatenursprung und der Clippingbereich des `JXGraphics`-Objekts eingestellt, so dass das Objekt einen zeichenbaren Bereich genau in der Größe des Peers und den Koordinatenursprung an der linken oberen Ecke der Position des Peers besitzt.

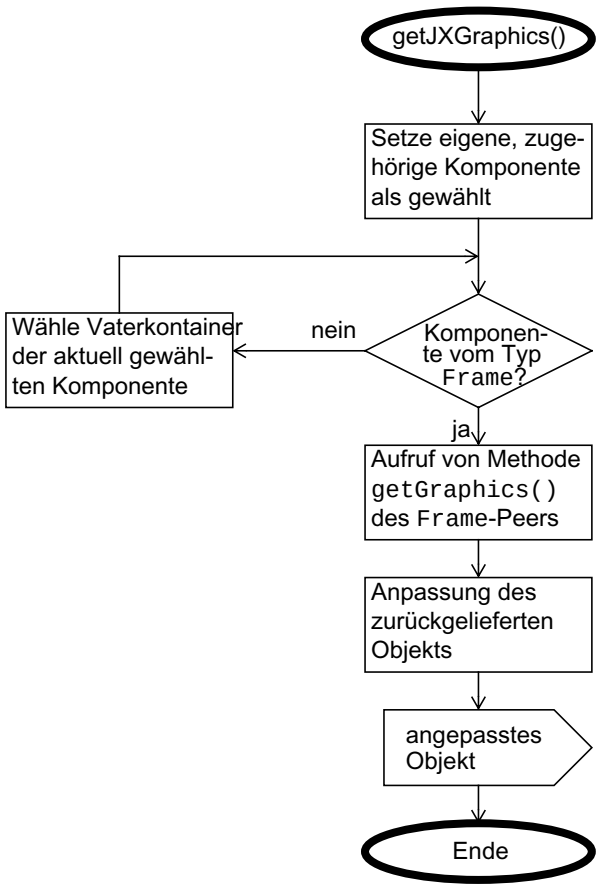


Abb. 4.3 Die Methode getJXGraphics()

Normalerweise wird die Methode `getJXGraphics()` eines Peers jedoch nicht direkt aufgerufen. Wenn ein Peer gezeichnet werden soll oder sich selbst neu zeichnen möchte, wird dessen Methode `redraw()` verwendet. Diese erzeugt sich eine neue gültige Instanz eines `JXGraphics`-Objekts mittels `getJXGraphics()` und ruft anschließend die Methode `paint()` des Peers mit dem erstellten Objekt als Parameter auf. Diese Methode beinhaltet den eigentlichen Zeichencode für einen Peer, und ist somit die bevorzugte Methode, wenn es darum geht, einem Peer ein anderes Aussehen zu geben.

Im Folgenden soll die Funktionsweise der gerade vorgestellten Methoden verdeutlicht werden. Als Beispiel soll diesmal wieder das Listing von Abbildung 2.5 dienen. Angenommen, das Programm steht gerade vor der letzten Anweisung, der Ausführung der Methode `show()`. Dann

stellt sich die Frage, was als Nächstes in den einzelnen Schichten vor sich geht, bis das fertige Fenster schließlich sichtbar auf dem Bildschirm steht, und vor allem, wie die Problematik des Zeichnens an sich dabei gehandhabt wird. Dazu werden nun alle Schritte aufgeführt, welche im Einzelnen dabei ablaufen:

- Die Komponente `Frame` ruft bei der Ausführung ihrer `show()`-Methode die Methode `show()` ihres Peers auf.
- Der Peer wiederum ruft die `show()`-Methode seines Konnektors auf, welche direkt ins Innere des Windowmanagers führt.
- Der Windowmanager erzeugt das Fenster und startet den zugehörigen Thread des Fensters. Dieser Thread zeichnet den Fensterrahmen auf den Bildschirm und ruft die `paint()`-Methode des Konnektors auf, um ihn zum Neuzeichnen seiner Selbst zu bewegen (Man beachte, dass der Konnektor von `WWindow` abgeleitet ist!).
- Der Konnektor ruft die Methode `redraw()` seines `Frame`-Peers auf.
- Der Peer sucht eine Instanz eines gültigen `JXGraphics`-Objekts, erhält sie von seinem Konnektor und ruft damit seine `paint()`-Methode auf, um sich selbst zu zeichnen.
- Danach geht der Peer die Verwaltungsstruktur seiner zugehörigen `Frame`-Komponente durch, findet die Instanz der Klasse `Button` und ruft die `redraw()`-Methode deren Peers auf.
- Der Peer der Klasse `Button` sucht selbst wiederum ein gültiges `JXGraphics`-Objekt. Dessen Methode `getJXGraphics()` findet als Halter eines gültigen Objekts den Peer der Klasse `Frame` und liefert nach Ende des Algorithmus eine neue Kopie der gleichen Instanz, die auch vom Peer der Klasse `Frame` als Vorlage verwendet wurde. Mit diesem Objekt ruft der Peer nun seine `paint()`-Methode auf, um sich seinerseits neu zu zeichnen.

Nach Ablauf dieser Vorgänge ist das Fenster samt seinem Knopf sichtbar auf dem Schirm dargestellt. Hätte das Fenster eine komplexere Hierarchie aus mehreren Komponenten und Kontainern, so würden sich die letzten beiden Schritte wiederholen: Jeder Peer eines Kontainers, dessen `redraw()`-Methode aufgerufen wird, zeichnet sich erst selbst neu und ruft dann die `redraw()`-Methoden seiner eingebetteten Komponenten bzw. deren Peers auf. Und jeder Peer muss sich eine eigene Kopie des `JXGraphics`-Objekts des Fensters holen, um sich selbst zeichnen zu können. Diese Vorgehensweise ist zwar nicht sehr förderlich für die Geschwindigkeit, erlaubt aber eine größere Flexibilität, was das Handling eines vorhandenen Menüsystems oder die Steuerung durch die Klasse `ScrollPane` betrifft (vgl. Abschnitt 4.5.7).

4.3 Allgemeine Eigenschaften eines Peers

So wie jede AWT-Komponente entweder von `Component` oder `MenuComponent` abgeleitet ist, ist jeder Peer von `JXComponentPeer` oder `JXMenuComponentPeer` abgeleitet. Dadurch besitzen alle Peers derselben Gruppe auch einige gemeinsame Eigenschaften, die hier etwas genauer erläutert werden sollen.

4.3.1 Die Initialisierung eines Peers

Allen Peers gemein ist die Art und Weise, wie ein Peer initialisiert wird: Wird ein Peer erzeugt, so erhält er eine Referenz auf die zugehörige AWT-Komponente. Jeder Peer ruft während der Initialisierung durch seinen Konstruktor alle relevanten Methoden seiner Komponente auf, und speichert deren Ergebnisse in eigenen Strukturen. So ruft z.B. ein Peer der Klasse `JXLabelPeer` die Methode `getText()` der Klasse `Label` auf, um den aktuellen Text des Label zu erhalten. Auf diese Art und Weise kennt der Peer nach seiner Initialisierung alle wichtigen Daten seiner Komponente.

Dieses Vorgehen ist möglich, da die Daten des Peers von seiner Komponente ständig aktualisiert werden: Betrachtet man den Quellcode der Classpath AWT, so stellt man fest, dass beinahe jede Methode einer Komponente, die Änderungen an den internen Daten vornimmt, auch eine entsprechende Methode des zugehörigen Peers aufruft, um ihn über diese Änderungen zu informieren. Eben diese entsprechenden Methoden werden auch im Konstruktor benutzt, um die Daten des Peers richtig einzustellen.

Nachdem der Konstruktor durchlaufen worden ist, wird am Ende das `ready`-Flag gesetzt. Dieses Flag gibt an, dass der Peer bereit ist, gezeichnet zu werden. Das Flag darf erst am Ende des Initialisierungsvorgangs gesetzt werden. Das liegt daran, dass der Peer während der Initialisierung die gleichen eigenen Methoden verwendet, die normalerweise von der zugehörigen Komponente benutzt werden, um den Peer über Zustandsänderungen zu informieren. Diese Methoden beinhalten meist eine `redraw()`-Anweisung, um die Änderungen auch gleich sichtbar zu machen. Da jedoch während der Initialisierung noch kein Zeichnen des Peers möglich ist (es gibt den Peer zu diesem Zeitpunkt noch nicht), testet die `redraw()`-Methode das `ready`-Flag und bricht bei gelöschtem Flag ab.

Es wäre statt dieser Lösung auch möglich gewesen, für jeden Peer noch weitere zusätzliche Methoden zu schreiben, welche die Zustandsänderungen nur speichern, und den Peer grafisch nicht aktualisieren. Diese könnten dann im Konstruktor verwendet, und somit das `ready`-Flag vermieden werden. Diese Idee führt dann allerdings zu einer Verdoppelung einiger Teile des Codes, was den Code selbst wiederum schlechter wartbar macht.

4.3.2 Die Klasse JXComponentPeer

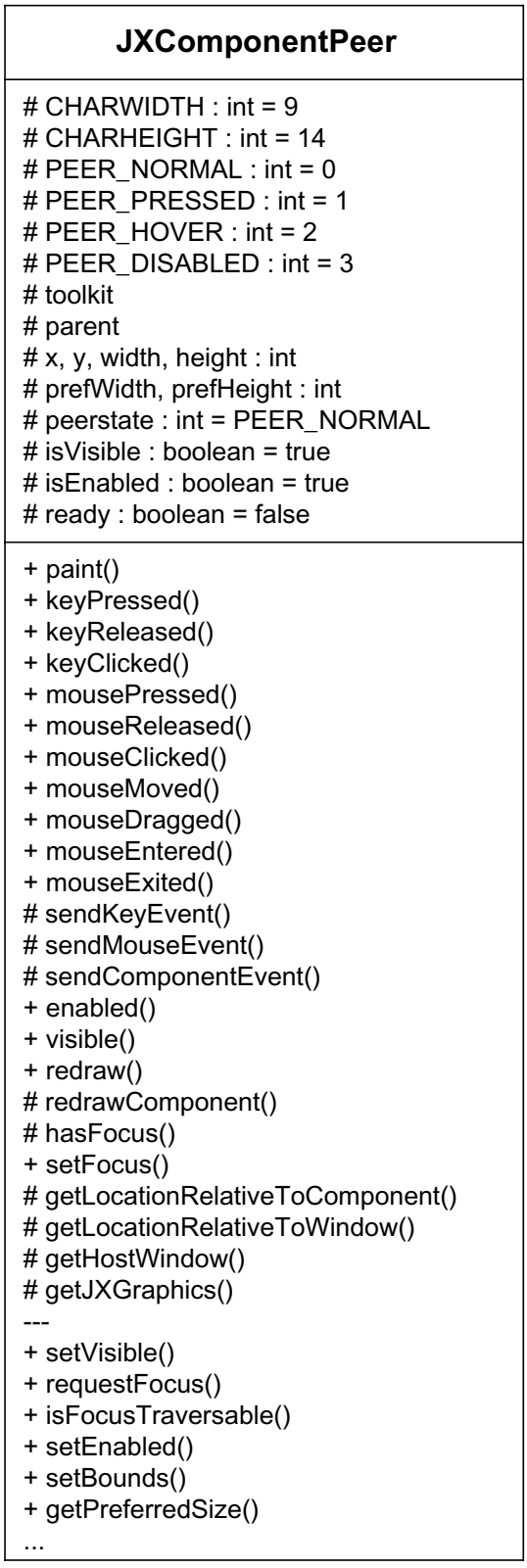


Abb. 4.4 Die Klasse JXComponentPeer

Durch das relativ umfangreiche Interface, das den Peers vorgegeben wird, welche von der Klasse `JXComponentPeer` abstammen, und wegen dem damit verbundenen Verwaltungsaufwand besitzen diese Peers auch einige gemeinsame Eigenschaften. Diese sollen im Weiteren näher erläutert werden. Abbildung 4.4 zeigt dazu die Klasse `JXComponentPeer` im Überblick.

Anmerkung: Wenn im Folgenden Sätze wie “Ein Peer besitzt...” oder “Ein Peer kennt...” o. ä. formuliert werden, so bedeutet dies, dass die jeweilige Fähigkeit von der Klasse `JXComponentPeer` geerbt wurde.

4.3.2.1 Allgemeine Eigenschaften

Die momentane Implementierung des Windowmanagers für JX kennt nur einen einzigen Zeichensatz für die Darstellung von ASCII-Zeichen. Unter der AWT gibt es normalerweise die Klassen `Font` bzw. `FontMetrics`, um Zeichensätze zu kapseln. Diese wurden jedoch nicht implementiert, weil die Unterstützung durch den Windowmanager nicht ausreichend war. Um dennoch nicht völlig auf den einzigen Zeichensatz von JX fixiert zu sein, wurden die Konstanten `CHARWIDTH` und `CHARHEIGHT` eingeführt. Diese beinhalten Breite und Höhe des aktuellen Zeichensatzes. Jeder Peer kennt diese Werte und verwendet diese für seine internen Berechnungen, so dass zumindest Zeichensätze mit fester Breite leicht durch Anpassung dieser Konstanten unterstützt werden können.

Weiterhin kennt jeder Peer einen “Peer state”, einen Grundzustand, in dem sich ein Peer befinden kann. Dieser Zustand bestimmt, wie der Peer auf dem Bildschirm dargestellt werden soll. Momentan werden vier Grundzustände unterstützt:

- PEER_NORMAL: Der Normalzustand, d.h. keiner der anderen Zustände
- PEER_PRESSED: Der Peer wird “gedrückt” dargestellt (z.B. wegen Mausklick)
- PEER_HOVER: Die Maus befindet sich gerade über der zugehörigen Komponente
- PEER_DISABLED: Der Peer wird deaktiviert dargestellt

Der aktuelle Zustand wird durch die jeweiligen Methoden zur Ereignisbehandlung gesetzt (vgl. Abschnitt 4.3.2.3).

Als weitere interne Informationen kennt jeder Peer noch seine aktuelle Größe und Position, seine ideale Größe (relevant für einige Layoutmanager der AWT), eine Referenz auf die globale Instanz von `JXToolkit`, eine Referenz auf die zugehörige Komponente, Flags, die angeben, ob der Peer sichtbar bzw. aktiviert[1] ist, sowie das bereits erwähnte `ready`-Flag. Diese internen Strukturen werden durch den Konstruktor der Klasse `JXComponentPeer` initialisiert.

1. Dieses Flag entspricht im eigentlichen Sinne nicht dem Grundzustand PEER_DISABLED: Das Flag gibt an, ob der Peer aktiviert ist oder nicht, während der Zustand bestimmt, ob der Peer aktiviert gezeichnet werden soll oder nicht. Zwar ist der Zustand PEER_DISABLED des Peers in der vorliegenden Implementierung direkt vom Flag abhängig, dennoch wird hier bewusst zwischen Flag und Zustand unterschieden, da beide im Grunde verschiedene Konzepte repräsentieren.

4.3.2.2 Methoden zum Zeichnen des Peers

Jeder Peer besitzt die bereits mehrfach erwähnte Methode `paint()`. Diese abstrakte Methode muss von jedem Peer mit eigenem Code überschrieben werden, welcher den Peer auf den Bildschirm zeichnet. Die Implementierung dieser Methode sieht bei allen Peers fast gleich aus:

- Der Peer wertet die ihm zur Verfügung stehenden Informationen aus: Welchen Grundzustand besitzt er, und hat er den Tastaturfokus oder nicht?
- Mit den relevanten Informationen wird eine eigene Methode zum Zeichnen des Peers aufgerufen, welche den Peer entsprechend den Daten auf dem Schirm darstellt (Bei der Klasse `JXLabelPeer` heißt diese Methode z.B. `paintLabel()`).

Die Methode `getJXGraphics()` wurde bereits ausführlich in Abschnitt 4.2.2 besprochen und soll hier nicht noch einmal vertieft werden.

Auch die Methode `redraw()` wurde bereits erwähnt und ist allen Peers eigen. Sie testet, ob das `ready`-Flag gesetzt ist, und ruft im positiven Fall die Methode `redrawComponent()` auf. Diese Methode wiederrum ist für die Arbeiten rund um das Zeichnen des Peers verantwortlich: Sie sucht zuerst ein neues, gültiges `JXGraphics`-Objekt mittels `getJXGraphics()`. Dann testet sie, ob der Peer sichtbar ist oder nicht. Ist er unsichtbar, wird die Methode `paintInvisibleComponent()` aufgerufen, welche eine unsichtbare Komponente "zeichnet", indem sie den Zeichenbereich der Komponente mit einer Standardfarbe (z.B. grau) füllt. Ansonsten wird die gerade erwähnte `paint()`-Methode aufgerufen, und anschließend ein AWT-Paint-Event für die zugehörige Komponente in die Ereigniswarteschlange geschoben.

Der Ablauf beider Methoden wird nochmals in Abbildung 4.5 verdeutlicht.

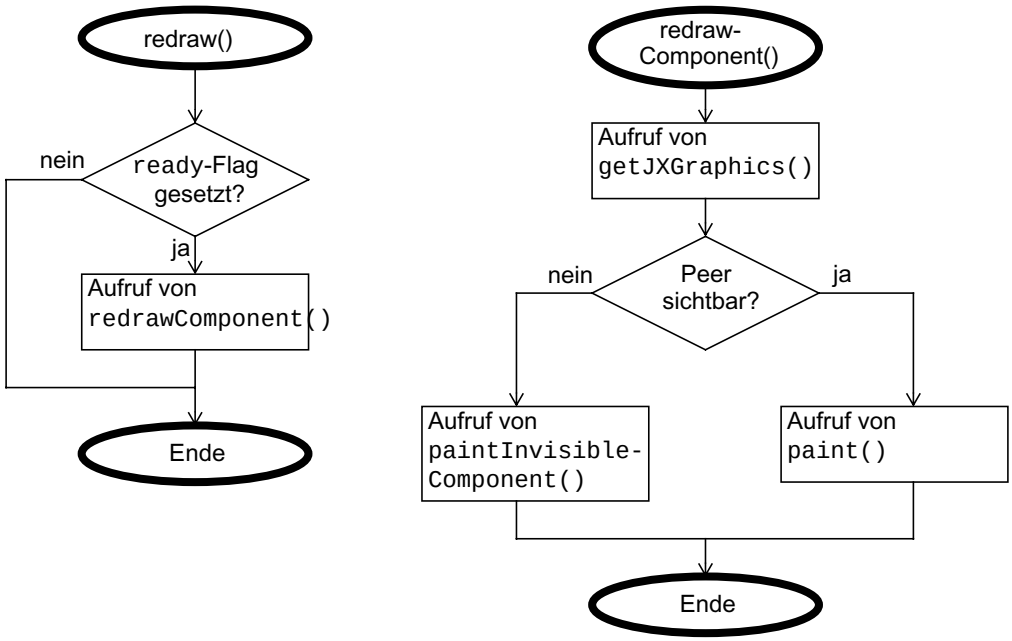


Abb. 4.5 Die Methoden redraw() und redrawComponent()

4.3.2.3 Behandlung von Ereignissen

Jeder Peer besitzt Methoden für die Reaktion auf unterschiedlichste Ereignisse, seien es Tastatur-, Maus- oder Paint-Events. Je nach Ereignis wird die entsprechende Methode des Peers aufgerufen, damit dieser auf das Ereignis reagieren kann. Abbildung 4.6 zeigt die verschiedenen Methoden und ihre zugehörigen Ereignisse im Überblick.

Methode	**Ereignis**
`keyPressed()`	Der Peer besitzt den Tastaturfokus, und eine Taste wurde gedrückt
`keyReleased()`	Der Peer besitzt den Tastaturfokus, und eine Taste wurde losgelassen
`keyClicked()`	Der Peer besitzt den Tastaturfokus, und eine Taste wurde betätigt
`mousePressed()`	Die Maus befindet sich über dem aktuellen Peer, und eine Maustaste wurde gedrückt
`mouseReleased()`	Die Maus befindet sich über dem aktuellen Peer, und eine Maustaste wurde losgelassen
`mouseClicked()`	Die Maus befindet sich über dem aktuellen Peer, und eine Maustaste wurde betätigt
`mouseEntered()`	Die Maus hat soeben den Bereich des Fensters betreten, in dem der Peer gezeichnet wird
`mouseExited()`	Die Maus hat soeben den Bereich des Fensters verlassen, in dem der Peer gezeichnet wird
`mouseMoved()`	Die Maus hat sich im Bereich des Peers bewegt
`mouseDragged()`	Die Maus hat sich im Bereich des Peers bewegt, dabei wird eine Maustaste gedrückt gehalten
`redraw()`	Der Peer muss neu gezeichnet werden

Abb. 4.6 Die Methoden zur Ereignisbehandlung

Aufgerufen werden die Methoden durch den Konnektor des Mutterfensters, in dem sich der Peer befindet. Der Konnektor empfängt selbst Nachrichten vom Windowmanager, falls bestimmte Ereignisse eingetreten sind, sucht dann die davon betroffene Komponente in der Komponentenhierarchie seiner `Frame`-Komponente heraus, und ruft die entsprechenden Methoden deren Peers auf (vgl. auch Abschnitt 3.3).

Neben der Verarbeitung von Ereignissen muss ein Peer diese meist auch an die nächsthöhere Schicht, also an die AWT weiterleiten, schließlich soll diese laut Definition ebenfalls imstande sein, Ereignisse zu verarbeiten. Um dies zu bewerkstelligen, bedient sich die AWT im 1.1-Ereignismodell der sogenannten Ereigniswarteschlange, welche vom Verteiler-Thread der AWT

verarbeitet wird, der wiederum die Events an die jeweiligen Eventlistener schickt. Um die AWT von Ereignissen in Kenntnis zu setzen, genügt es also, die entsprechenden Events in die Warteschlange einzufügen, alles Weitere erledigt die AWT.

Obwohl es in der Classpath AWT bereits eine fertige Implementierung dieser Warteschlange gibt, konnte diese nicht zuverlässig unter JX benutzt werden. Das lag daran, dass JX das `synchronized`-Konstrukt (noch) nicht unterstützt, welches aber von der Warteschlange gefordert wird, damit es zu keinen sogenannten "Race-Conditions" kommt, d.h. zu instabilen Zuständen der Schlange bei gleichzeitigem Ein- und Austragen durch mehrere Threads. Um dennoch eine zuverlässige Warteschlange benutzen zu können, musste eine neue Implementierung mit einigen JX-spezifischen Elementen erstellt werden.

Bei dieser neuen Implementierung handelt es sich letztendlich um eine Kapselung der Klasse `MultiThreadList`, welche eine Hilfsklasse darstellt, die von einigen Teilen des JX-Betriebssystems verwendet wird. Diese Klasse implementiert eine sichere Warteschlange für Objekte unter gleichzeitiger Verwendung durch einen "Verbraucher" und mehrere "Erzeuger". Die neue Version der Klasse `EventQueue` bietet alle benötigten Methoden der Originalversion und leitet deren Aufrufe zum Ein- und Aushängen von Objekten einfach an die entsprechenden Methoden der Klasse `MultiThreadList` weiter.

4.3.2.4 Handling des Tastaturfokus

Wie in Abschnitt 2.3.4 beschrieben, muss ein Peer einer Komponente einen Teil der Fokusverwaltung übernehmen. So ist z.B. zu regeln, wann ein Fokuswechsel überhaupt stattfinden soll. Die vorliegende Implementierung regelt die Vergabe des Fokus nach drei Bedingungen:

- Einerseits bekommt diejenige Komponente den Fokus, auf die gerade mit der Maus geklickt wurde, und die dazu in der Lage ist, den Fokus zu erhalten.
- Andererseits wird mit der <**Tab**>-Taste die Methode `transferFocus()` der aktuell fokussierten Komponente aufgerufen, d.h. der Fokus wird, wenn möglich, weitergereicht (vgl. Abschnitt 2.3.4).
- Die dritte Möglichkeit dient der Handhabung des Fensterwechsels: Wenn ein Fenster inaktiv wird, so speichert dessen Konnektor eine Referenz auf die aktuell fokussierte Komponente seines Fensters. Wird ein Fenster dagegen aktiv, so wird getestet, ob eine solche Referenz gespeichert wurde. Falls das zutrifft, so wird die Methode `requestFocus()` der gespeicherten Komponente aufgerufen, damit diese wieder den Fokus erhält, ansonsten erhält das aktive Fenster den Fokus.

Wie bereits beschrieben, dient zur Verwaltung des Tastaturfokus eine globale Instanz der Klasse `FocusHandler` (vgl. Abbildung 4.7). Diese Instanz speichert lediglich eine Referenz auf die momentan fokussierte Komponente des momentan aktiven Fensters, und ist notwendig, falls es darum geht, diese Komponente anzusprechen oder dafür zu sorgen, dass sich nur diese Komponente entsprechend kennzeichnet. Tatsächlich besitzt nämlich jedes Fenster zu jedem Zeitpunkt eine fokussierte Komponente, aber nur diejenige im aktiven Fenster soll sich auch so darstellen dürfen.

Weder irgendein Peer noch ein Konnektor kennt die momentan fokussierte Komponente eines Fensters direkt. Jeder Peer kann jedoch für sich herausfinden, ob der den Tastaturfokus besitzt oder nicht. Dazu kennt er die Methode `hasFocus()`, die sich einfach der Referenz der Klasse `FocusHandler` bedient, um diese Frage zu beantworten.

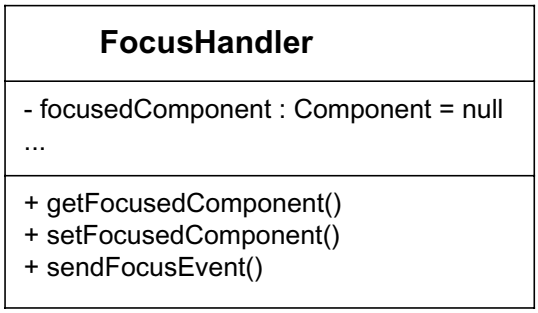


Abb. 4.7 Die Klasse FocusHandler

Damit ein Peer überhaupt den Fokus bekommen kann, muss er der AWT erst mitteilen, dass er dazu in der Lage ist. Dafür besitzt jeder Peer die Methode `isFocusTraversable()`. Ein Peer, der den Fokus bekommen möchte, muss hier `true` zurückliefern, ansonsten `false`.

Außerdem besitzt er eine Implementierung der Methode `requestFocus()`. Diese regelt die eigentliche Übergabe des Fokus an diesen Peer auf einfache Art und Weise: Der aktuelle Fokusbesitzer wird von der globalen Instanz der Klasse `FocusHandler` ermittelt. Dann wird die Referenz des Handlers auf die eigene Komponente gesetzt, und der alte und neue Fokusbesitzer neu gezeichnet, um den Fokuswechsel sichtbar zu machen.

4.3.3 Die Klasse JXContainerPeer

Der Unterschied zwischen Kontainer und normaler Komponente ist auf der Peer-Ebene nur marginal. Dies liegt vor allem daran, dass der betriebssystemunabhängige Teil der AWT die meiste Arbeit eines Kontainers bereits implementiert.

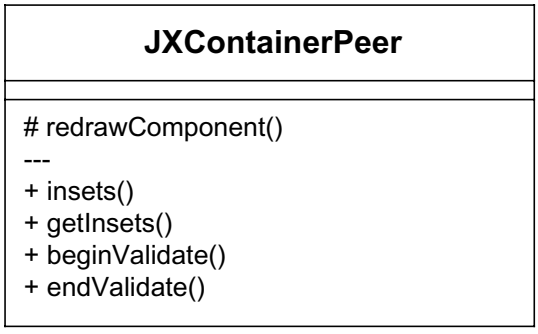


Abb. 4.8 Die Klasse JXContainerPeer

Der wesentliche Unterschied zwischen einem normalen Peer und einem Kontainer-Peer liegt darin, dass die `redrawComponent()`-Methode eines Kontainer-Peers erweitert ist: Außer dass der Peer sich wie gewohnt selbst zeichnet, ruft er noch die `redrawComponent()`-Methoden aller Peers auf, deren Komponenten in seine eingebettet sind. Auf diese Weise wird das rekursive Neuzeichnen eines Zweiges oder des ganzen Hierarchiebaumes ermöglicht.

Daneben kennt ein Kontainer noch die sogenannten “insets”. Dabei handelt es sich um Durchmesserangaben, welche einen Rahmen um die eingebetteten Komponenten definieren, in welchen nur der Kontainer selbst zeichnen kann. Der gesamte Platzbedarf des Kontainers ergibt sich hier durch den Platz der eingebetteten Komponenten sowie den umfassenden Rahmen.

4.3.4 Die Klasse JXMenuComponentPeer

Die Peers, welche das Menüsystem eines Fensters repräsentieren, unterscheiden sich grundlegend von den vorhergehenden Peers. Das liegt daran, dass bei der AWT die Schnittstelle zum Menüsystem sehr klein gehalten ist, und es der nativen Implementierung vollständig überlassen bleibt, wie das Menüsystem gestaltet und bedient wird.

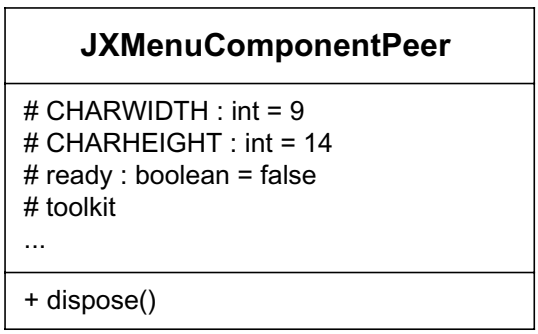


Abb. 4.9 Die Klasse JXMenuComponentPeer

Die Peers des Menüsystems besitzen in der vorliegenden Implementierung keine nennenswerten gemeinsamen Eigenschaften, außer dem bereits erwähnten `ready`-Flag und den Dimensionen des Standardzeichensatzes des Windowmanagers. Mehr ist auch nicht notwendig, da die Schnittstelle keine weiteren speziellen Eigenschaften erforderlich macht. Deswegen wurde die Implementierung dieser Gruppe so einfach und übersichtlich wie möglich gehalten, wie in Abschnitt 4.6 noch ausführlicher behandelt wird.

4.4 Peers für einfache eingebettete Komponenten

Dieser Abschnitt beschäftigt sich mit der Implementierung der Peers für die grundlegenden AWT-Komponenten. Mit “grundlegend” sind dabei diejenigen Komponenten gemeint, welche ohne besondere Ressourcen auskommen, außer dass sie einen gewissen Bereich innerhalb des Mutterfensters für sich beanspruchen.

Anmerkung: Die folgenden Beschreibungen enthalten meist keine besonderen Hinweise über die Implementierung von peertypischen Methoden, d.h. der Methoden, welche die Peers laut ihrem Interface implementieren müssen. Dies liegt daran, dass diese Methoden in den meisten Fällen entweder nicht implementiert sind, weil unnötig für diese Implementierung, oder weil sie einfach die übergebenen Parameter in eigene Strukturen übernehmen und sich evtl. daraufhin aktualisieren und neu zeichnen. Die wenigen Methoden, welche nicht in diese beiden Kategorien fallen, werden jedoch genauer beschrieben, falls deren Verständnis notwendig ist.

4.4.1 Die Klasse JXCanvasPeer

Der Peer der Klasse `Canvas` lässt sich ohne großen Aufwand realisieren, da er nur eine freie Fläche darstellt und auf keinerlei Interaktion auf nativer Ebene ausgelegt sein muss. Dementsprechend begnügt sich der hier realisierte Peer damit, von `JXComponentPeer` abgeleitet zu sein und damit alle allgemeinen Eigenschaften eines Peers zu beherrschen.

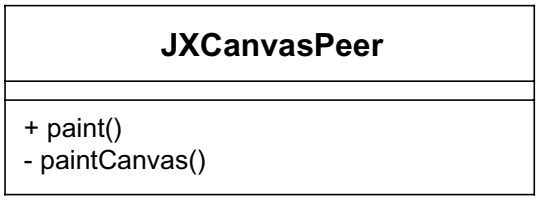


Abb. 4.10 Die Klasse JXCanvasPeer

4.4.2 Die Klasse JXPanelPeer

Da die speziellen Eigenschaften eines Kontainers bereits auf der Komponentenebene durch die AWT realisiert werden, sind hier keine weiteren Arbeiten oder Implementierungen von Seiten des Peers notwendig. Die vorliegende Implementierung des Peers beschränkt sich deswegen darauf, die `paint()`-Methode zu implementieren, die den Bereich des Kontainers neu zeichnet bzw. einfach löscht.

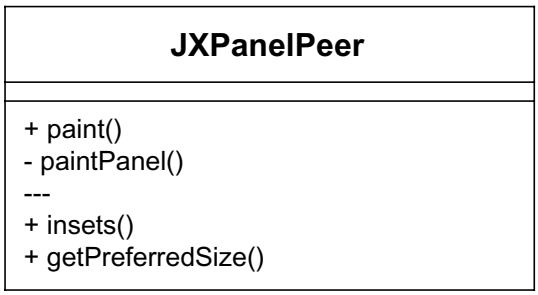


Abb. 4.11 Die Klasse JXPanelPeer

Diese Implementierung des Peers verwendet keinen extra Rahmen für sich, so dass dadurch keine Zeichenfläche unnötig verschwendet wird (vgl. Abschnitt 4.3.3). Falls ein solcher gewünscht wird, kann man die `ExtendedPanel`-Klasse verwenden. Diese wird in Abschnitt 4.10.2 noch genauer beschrieben.

4.4.3 Die Klasse JXLabelPeer

Hier wird erstmals der Peerzustand mit berücksichtigt: Abhängig davon, ob die Komponente `Label` als aktiviert oder deaktiviert markiert ist, wird der zu zeigende Text beim Zeichnen entsprechend gefärbt. Dann wird der Text auf den vorher gelöschten Zeichenbereich geschrieben.

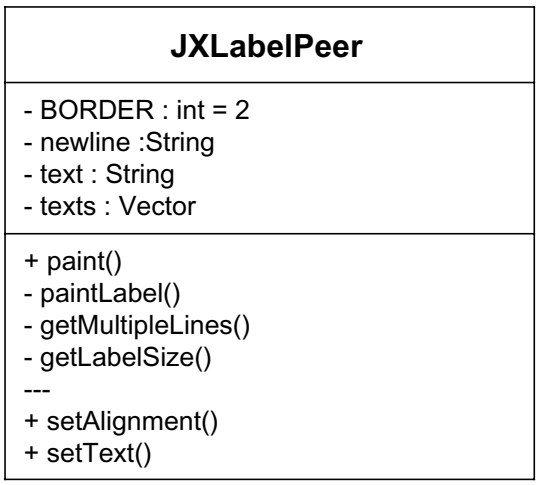


Abb. 4.12 Die Klasse JXLabelPeer

Diese Klasse beinhaltet darüber hinaus die Unterstützung für die Klasse `ExtendedLabel`, welche in Abschnitt 4.10.1 beschrieben wird.

4.4.4 Die Klassen JXButtonPeer und JXCheckboxPeer

Beide Klassen repräsentieren einfache Peers, die auf Maus- sowie auf Tastaturereignisse reagieren können, sowie alle vorhandenen Peerzustände interpretieren. Dementsprechend verläuft die Ereignisbehandlung bei beiden Peers auch fast gleich: Bei jedem Maustastendruck oder dem Drücken der <**Space**>-Taste wird der Grundzustand aktualisiert und der Peer neu gezeichnet. Nur die Implementierung von `keyClicked()` und `mouseClicked()` ist verschieden: Während die `Button`-Implementierung ein "Action-Event" nach oben an die AWT weiterreicht, sendet die Klasse `JXCheckboxPeer` hier ein "Item-Event" und aktualisiert dazu noch das `selected`-Flag (siehe unten).

Beiden Peers ist außerdem gemein, dass sie den Fokus bekommen können, also die Methode `isFocusTraversable()` entsprechend implementiert haben.

Beide Peers zeichnen sich entsprechend ihrem jeweiligen Grundzustand neu: Die vorliegenden Implementierungen zeichnen sich selbst neu, sobald sie aktiviert oder deaktiviert, gedrückt oder losgelassen werden, oder die Maus ihren Bereich betritt oder verlässt. Ihre `paint()`-Methoden werten diesen Zustand aus und rufen die Methoden entsprechend auf.

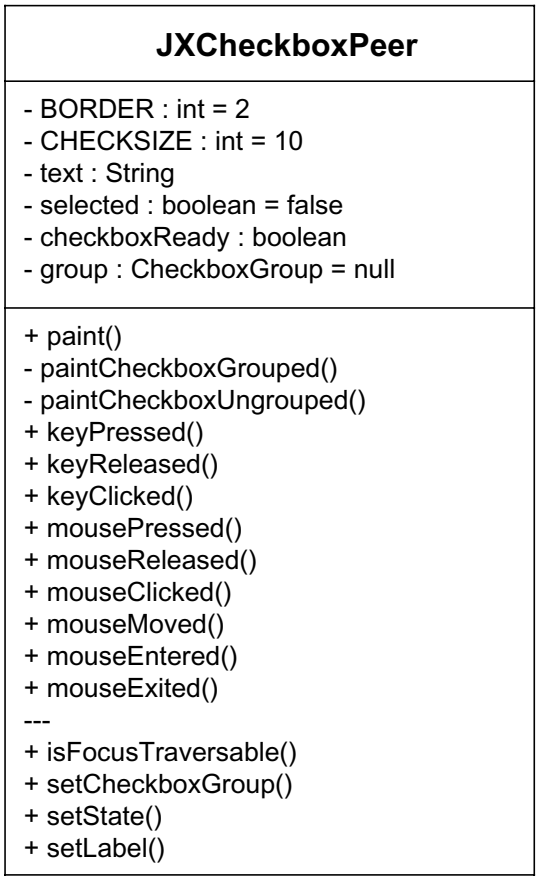


Abb. 4.13 Die Klasse JXCheckboxPeer

Erwähnenswert ist, dass in `JXCheckboxPeer` gleich zwei Methoden zum Zeichnen vorhanden sind: `paintCheckboxGrouped()` und `paintCheckboxUngrouped()`. Das liegt daran, dass es auch zwei Arten von Checkboxes gibt, eine für Gruppierungen und eine "alleinstehende" Version. Die beiden Methoden zeichnen die Checkbox entsprechend ihrer Art.

Die Klasse `JXCheckboxPeer` besitzt neben diesen Eigenschaften noch weitere: So kennt sie das bereits erwähnte `selected`-Flag. Dies ist notwendig, da eine Checkbox einen Zustand speichern muss, der angibt, ob sie gerade ausgewählt ist oder nicht, und sich entsprechend darstellen muss. Falls ihre zugehörige AWT-Komponente eine Referenz auf ein Objekt der Klasse `CheckboxGroup` besitzt (es sich also um eine gruppierte Checkbox handelt), so besitzt diese Referenz auch der Peer, und informiert dieses Objekt, sobald sich das `selected`-Flag ändert.

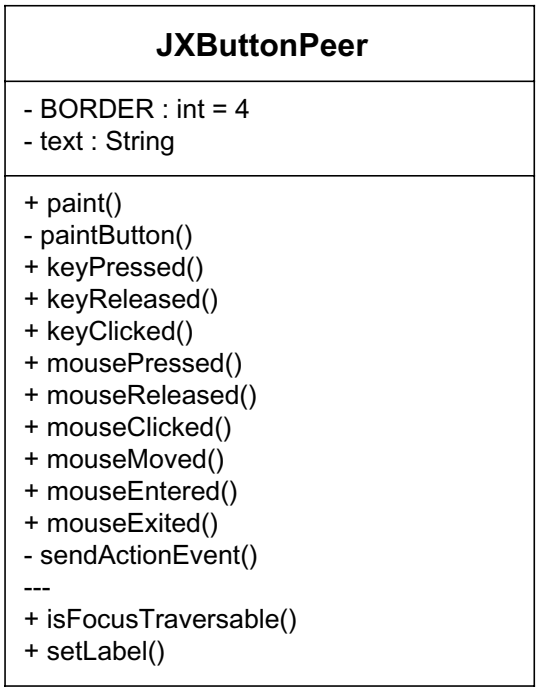


Abb. 4.14 Die Klasse JXButtonPeer

4.4.5 Die Klasse JXTextComponentPeer

So wie unter der AWT die Klassen `TextField` und `TextArea` von `Textcomponent` abgeleitet sind, und diese wiederum von `Component`, so ist `JXTextComponentPeer` von `JXComponentPeer` abgeleitet und kennt die Unterklassen `JXTextFieldPeer` und `JXTextAreaPeer` (vgl. Abbildung 4.15). Die abstrakte Klasse `JXTextComponentPeer` beinhaltet die gemeinsamen Methoden ihrer beiden Unterklassen, und daneben alles, was zur Verwaltung eines Textbereiches notwendig ist.

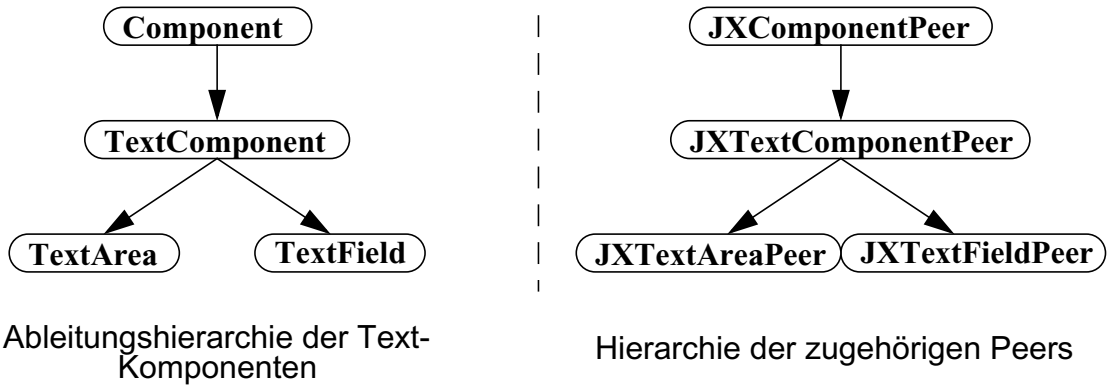


Abb. 4.15 Die Ableitungshierarchien der Text-Komponenten und deren Peers

Die Klasse, welche in Abbildung 4.16 dargestellt ist, beinhaltet folgende wesentlichen Datenstrukturen:

- Ein Objekt vom Typ `Stringbuffer`, das den Text beinhaltet.
- Ein Objekt vom Typ `String`, welches das “new line”-Zeichen beinhaltet, also das Zeichen, an dem der Text gebrochen wird (unter JX ’\n’).
- Drei Indexe, welche auf den Text zeigen: Zwei Markierungen für Anfang und Ende einer Auswahl, sowie die aktuelle Position des Textcursors (das sog. “caret”).
- X- und Y-Koordinate eines Offsets, der die Verschiebung des Textbereichs auf dem Bildschirm angibt.
- Das `editable`-Flag. Dieses gibt an, ob der Text durch Tastatureingaben durch den Benutzer manipulierbar ist oder nicht.

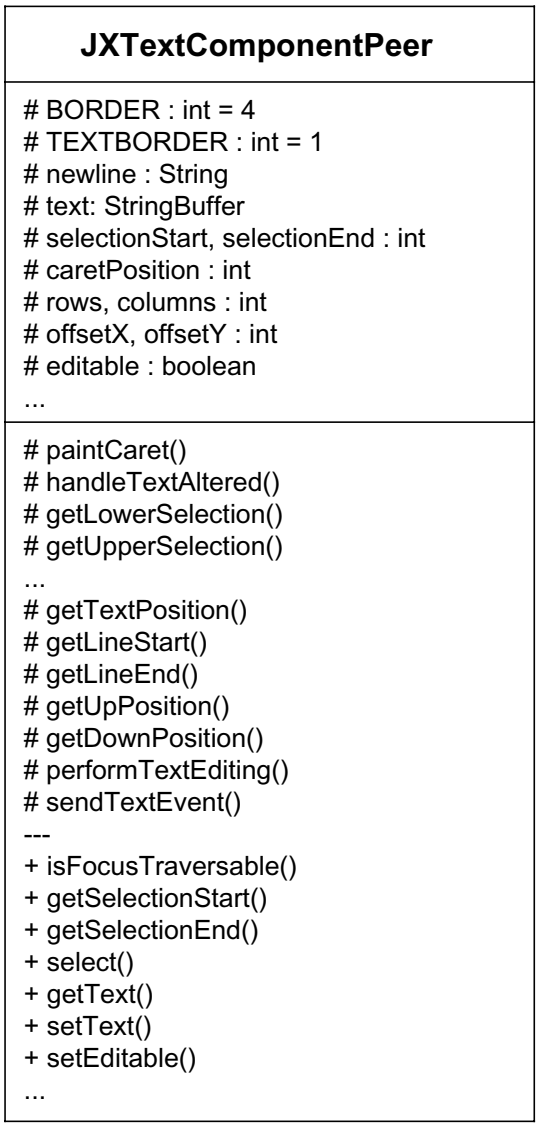


Abb. 4.16 Die Klasse JXTextComponentPeer

Die Klasse `JXTextComponentPeer` besitzt die abstrakte Methode `handleTextAltered()`. Diese Methode muss von den Unterklassen implementiert werden und wird immer dann aufgerufen, wenn sich der Inhalt des Textes ändert. Der Sinn dahinter ist, dass es einer Unterklasse ermöglicht werden soll, den Text in einer eigenen internen Form zu speichern, welche für die jeweilige Klasse am besten geeignet ist, und durch diese Methode immer die eigene Struktur aktualisieren zu können.

Daneben besitzt die Klasse noch einige Methoden, um einen zweidimensionalen Textbereich mit einem eindimensionalen Puffer verwalten zu können: Methoden, die den Index von Zeilenanfang oder Zeilenende der aktuellen Zeile finden, oder den Index, auf welchem der Cursor stehen würde, wenn man ihn eine beliebige Anzahl Zeilen nach oben oder unten bewegt. Auch Konvertierungsroutinen zum Umrechnen von Textdimensionen in Pixel und zurück sind vorhanden, sowie eine Methode namens `getTextPosition()`, welche zu den angegebenen Koordinaten im zweidimensionalen Textbereich den Index des Buchstabens angibt, der sich gerade darunter befindet.

Eine wichtige Methode der Klasse stellt die Methode `performTextEditing()` dar: Diese kümmert sich um die gesamten Textmanipulationen, welche durch Eingaben des Benutzers entstehen können. Als Parameter bekommt sie den Tastencode übergeben, der ausgewertet werden soll, und bearbeitet den Text dann entsprechend. Bislang werden folgende Tasten unterstützt:

- Cursortasten: Bewegen den Textcursor je nach Taste um eine Position nach links, rechts, oben oder unten. Falls der Cursor dabei aus dem erlaubten Bereich herausfallen sollte, wird die Position automatisch korrigiert. Wird dabei die <**Shift**>-Taste gedrückt gehalten, so wird die Auswahl entsprechend korrigiert.
- <**Home**>, <**End**>: Bewegen den Cursor zum Anfang bzw. Ende der Zeile. Wird dabei die <**Shift**>-Taste gedrückt gehalten, so wird die Auswahl entsprechend korrigiert.
- <**PgUp**>, <**PgDn**>: Bewegen den Cursor um die Höhe des sichtbaren Textbereichs nach oben oder unten, falls der Bereich mehrzeilig ausgelegt ist. Die Cursorposition wird dabei, falls nötig, angepasst. Bei gedrückter <**Shift**>-Taste wird die Auswahl korrigiert.
- <**BackSpace**>: Falls eine Auswahl besteht, wird diese gelöscht, ansonsten das Zeichen, welches vor dem Textcursor steht. Der Cursor wird danach an den Anfang des gerade manipulierten Bereichs gesetzt.
- <**Delete**>: Falls eine Auswahl besteht, wird diese gelöscht, ansonsten das Zeichen, welches gerade “unter” dem Textcursor steht.
- <**Enter**> bzw. <**Return**>: Falls der Textbereich einzeilig ausgelegt ist (z.B. bei `JXTextFieldPeer`, siehe nächsten Abschnitt), so wird ein “Action-Event” an die AWT gesendet. Falls es sich um einen mehrzeiligen Textbereich handelt, so wird das Zeichen einfach als “normales” Zeichen behandelt.

- "normale" Zeichen: In der vorliegenden Implementierung fallen Zeichen, die nicht bereits vorher bearbeitet wurden, in diese Kategorie. Diese werden einfach an die aktuelle Cursorposition eingefügt, wobei eine bestehende Auswahl vorher gelöscht wird. Der Cursor wird danach hinter das neue Zeichen gesetzt.

Die Methode wertet bei ihren Arbeiten das `editable`-Flag aus: Handelt es sich um eine Taste, welche den Textinhalt manipuliert, so wird diese nur dann verarbeitet, wenn das `editable`-Flag gesetzt ist. Bei allen Arbeiten, welche den Textinhalt manipulieren, wird dabei die Methode `handleTextAltered()` aufgerufen, um der Unterklasse Gelegenheit zu geben, ihre Strukturen zu aktualisieren (vgl. oben). Außerdem werden auch sog. "Text-Events" an die AWT versandt, um sie von den Änderungen in Kenntnis zu setzen.

4.4.6 Die Klasse JXTextFieldPeer

Dieser Peer ist der Erste der beiden abgeleiteten Peers von `JXTextComponentPeer`. Er implementiert einen einzeiligen Textbereich, der mit der Maus und der Tastatur bearbeitet werden kann. Abbildung 4.17 zeigt den Peer im Überblick.

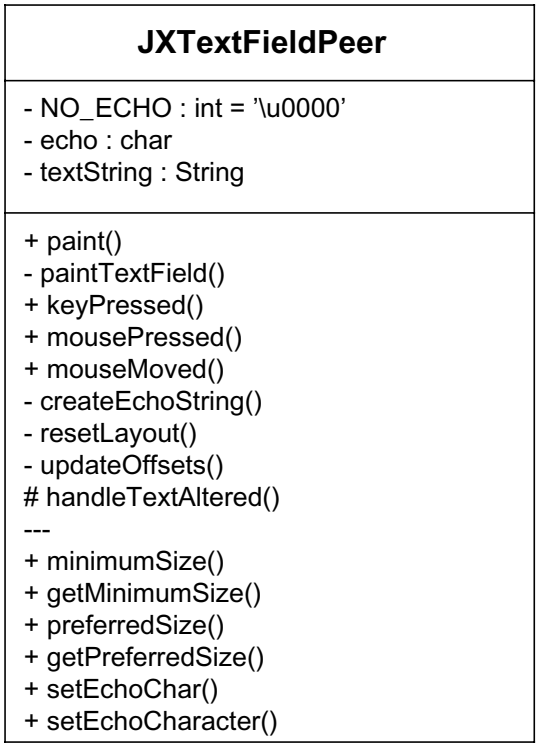


Abb. 4.17 Die Klasse JXTextFieldPeer

Die Maus dient lediglich der Modifikation der Auswahl: Drückt der Benutzer auf eine Maustaste, so wird sowohl der Textcursor als auch der Anfang der Auswahl auf die Position des Zeichens unterhalb der Maus gesetzt. Hält der Benutzer die Maustaste gedrückt und bewegt die Maus, so wird dies innerhalb der Ereignismethode `mouseMoved()` erkannt, und die Auswahl wird auf den Bereich zwischen Markierungsanfang und der aktuellen Mausposition gesetzt.

Die Bedienung mit der Tastatur wurde auf einfache Weise implementiert: Bei jedem Tastendruck wird die gedrückte Taste an die geerbte Methode `performTextEditing()` weitergeleitet, und diese kümmert sich um alles Weitere (vgl. vorherigen Abschnitt).

Die Implementierung der Methode `handleTextAltered()` wandelt den Zeichenpuffer in ein `String`-Objekt um, welches entweder den Textinhalt oder einen "Echo-String" enthält, d.h. eine Zeichenkette in der Länge des Textes, die mit "Echo-Zeichen" gefüllt ist (Ein typisches Beispiel dafür ist ein Passwort-Textfeld). Dies ist abhängig davon, ob in der zugehörigen Komponente ein Echo-Zeichen definiert ist oder nicht.

Dieses `String`-Objekt wird dann verwendet, wenn der Peer gezeichnet wird. In der vorliegenden Implementierung berücksichtigt der Peer dabei, ob er aktiviert oder deaktiviert ist, sowie das `editable`-Flag. Je nach Gesamtzustand zeichnet sich der Peer anders.

4.5 Peers für erweiterte eingebettete Komponenten

Nun sollen die Peers für die erweiterten AWT-Komponenten betrachtet werden. Diese Peers waren mit einem umfangreicheren Implementierungsaufwand verbunden, nicht zuletzt deshalb, weil sie auf einige zusätzliche Ressourcen angewiesen waren, die im Folgenden kurz erläutert werden sollen. Anschließend werden die Peers selbst beschrieben.

4.5.1 Verwendung von internen Scrollbalken: Die Klasse InternalScrollbar

Die Peers der Klassen `Choice`, `List`, `Scrollbar`, `ScrollPane` sowie `TextArea` sind alle auf die Verwendung von Scrollbalken angewiesen. Eine Möglichkeit wäre, diese Balken auf Anwenderebene zu realisieren, indem man einfach die notwendige Anzahl von Instanzen der Klasse `Scrollbar` erzeugt, und mittels entsprechender Eventlistener die abhängigen Komponenten bzw. deren Peers manipuliert. Diese Idee wäre zwar mit viel Aufwand realisierbar, jedoch bringt die Klasse `Scrollbar` einen Overhead an Code und Daten mit sich, der für die anderen Klassen meist überflüssig ist und diese unnötig verlangsamt. Abgesehen davon widerspricht die Verwendung einer AWT-Komponente auf Peer-Ebene dem Prinzip der Schichtentrennung.

Aus diesem Grund wurde eine dedizierte, auf das Nötigste beschränkte Klasse namens `InternalScrollbar` eingefügt, die, wie der Name schon sagt, einen Scrollbalken für die interne Verwendung innerhalb eines Peers zur Verfügung stellt. Diese Klasse ist kein eigenständiger Peer, kann aber in JX-Peers miteingebunden werden, wie z.B. in den Peer der Komponente `Scrollbar` (vgl. Abschnitt 4.5.3). Sie unterstützt sowohl die vertikale als auch die horizontale Darstellung, und lässt sich vom benutzenden Peer unabhängig auch deaktivieren. Zum Einstellen aller Parameter, die das Aussehen und das Verhalten des Scrollbalkens beeinflussen, bietet die Klasse alle notwendigen Methoden, sowie auch eine Methode zum Auslesen des momentan dargestellten Wertes. Abbildung 4.18 zeigt die Klasse im Überblick.

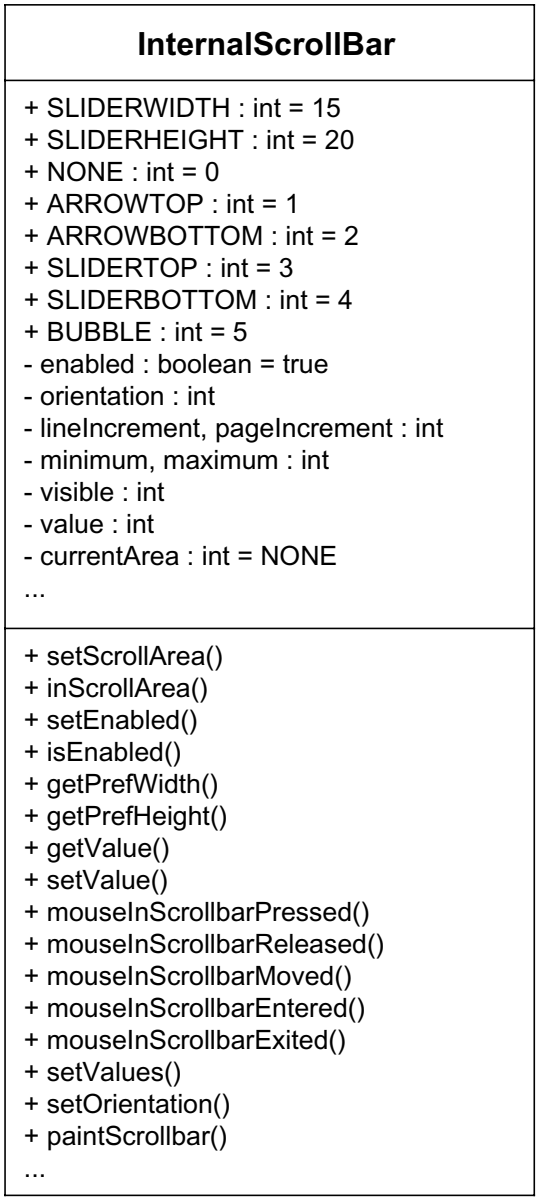


Abb. 4.18 Die Klasse InternalScrollbar

Die Verwendung in einem Peer lässt sich relativ leicht bewerkstelligen, wenn man die folgenden Regeln dabei berücksichtigt:

- Die Klasse `InternalScrollbar` muss wissen, wo im Zeichenbereich des Peers “ihr” Bereich liegt, in dem sie sich zeichnen darf, und wo sie auf Ereignisse warten kann. Dies muss ihr durch ihre Methode `setScrollArea()` mitgeteilt werden.
- Die Klasse besitzt eigene Ereignishandler für verschiedene Ereignisse, wie in Abbildung 4.19 dargestellt. Diese aktualisieren die internen Datenstrukturen und das Layout der Klasse. Jedoch wird die Klasse nicht direkt über diese Ereignisse informiert, da nur Peers Ereignisnachrichten erhalten. Diese Methoden müssen dementsprechend vom Peer aufgerufen werden, sobald das jeweilige Ereignis eingetreten ist.

- Die `paint()`-Methode des Peers muss die Methode `paintScrollbar()` der Klasse `InternalScrollbar` aufrufen. Auf diese Weise wird sichergestellt, dass die Scrollbar ebenfalls immer im richtigen Augenblick gezeichnet wird. Es muss jedoch darauf geachtet werden, dass der Scrollbalken nicht vom Peer übermalt wird, d.h. dass die Methode `paintScrollbar()` möglichst erst nach dem Zeichencode des Peers aufgerufen wird.

Man beachte, dass die Klasse keine Unterstützung für Tastaturevents besitzt. Dies wäre auch gar nicht direkt umsetzbar, da es bei einem Peer nicht eindeutig sein muss, welche Taste für die Scrollbar gedacht ist und welche nicht. Tatsächlich muss der Peer selbst entscheiden, auf welche Tasten er wie reagieren soll, und die Scrollbar gegebenenfalls selbst auf den gewünschten Wert setzen.

Methode	**Ereignis**
`mouseInScrollbarPressed()`	Eine Maustaste wurde in der Scrollbar gedrückt
`mouseInScrollbarReleased()`	Eine Maustaste wurde in der Scrollbar losgelassen
`mouseInScrollbarMoved()`	Die Maus wurde im Bereich der Scrollbar bewegt
`mouseInScrollbarEntered()`	Die Maus hat den Scrollbarbereich betreten
`mouseInScrollbarExited()`	Die Maus hat den Scrollbarbereich verlassen

Abb. 4.19 Die Ereignishandler der Klasse InternalScrollbar

Intern unterteilt die Klasse ihren Zeichenbereich in fünf Teile, welche die unterschiedlichen Bauteile des Scrollbalkens darstellen: Zwei Knöpfe, einen Balken, der mit der Maus gezogen werden kann, und zwei freie Flächen, über die der Balken bewegt wird. Beinahe jeder der gerade erwähnten Ereignishandler gibt beim Aufruf einen Wert zurück, der wiederspiegelt, wo sich die Maus gerade auf dem Scrollbalken befindet. Insgesamt können dabei folgende Werte zurückgegeben werden:

- NONE: Die Maus ist nicht im Bereich des Scrollbalkens
- ARROWTOP: Die Maus befindet sich über dem oberen/linken Knopf
- ARROWBOTTOM: Die Maus befindet sich über dem unteren/rechten Knopf
- SLIDERTOP: Die Maus befindet sich über dem oberen/linken freien Bereich
- SLIDERBOTTOM: Die Maus befindet sich über dem unteren/rechten freien Bereich
- BUBBLE: Die Maus befindet sich über dem Balken, der mit der Maus gezogen werden kann

4.5.2 Verwendung von Zusatzfenstern: Die Klasse SlaveWindowHandler

Die Komponente `Choice` hat die Fähigkeit, bei einem Mausklick eine Liste auszuklappen, aus welcher der Benutzer mittels Maus und Tastatur eine Auswahl treffen kann. Diese Liste muss notwendigerweise als eigenes Fenster realisiert werden, da die Liste durchaus über den Fensterrand des Mutterfensters hinausreichen könnte. Trotzdem darf das Listenfenster nicht mit anderen Fenstern gleichgestellt sein, und z.B. keinen typischen Fensterrahmen besitzen. Darüber hinaus darf es nur solange sichtbar sein, bis ein Mausklick außerhalb des Fensters stattfindet, es ist also von seiner Umgebung abhängig.

Zur Verwaltung der Abhängigkeit solcher Fenster wurde die Klasse `SlaveWindowHandler` geschrieben. Diese Klasse wird allgemein verwendet, um eine Komponente zu registrieren, die noch Aufräumarbeiten zu verrichten hat, sobald sie oder ihr Mutterfenster den Fokus verliert. Benötigt eine Komponente diese Unterstützung, so registriert sie sich bei der globalen Instanz der Klasse `SlaveWindowHandler`. Wenn ein Konnektor nun ein Ereignis erhält, dass sein Fenster aktiviert oder deaktiviert wurde, fragt er zuerst bei der globalen Instanz nach, ob Komponenten registriert sind, und falls ja, wird die entsprechende Aufräumprozedur für jede Komponente gestartet und anschließend die Registrierung der Komponenten gelöscht.

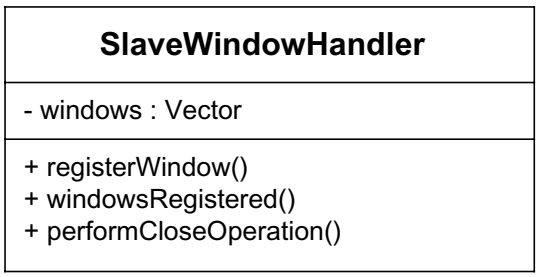


Abb. 4.20 Die Klasse SlaveWindowHandler

Auch die Klasse `SlaveWindowHandler` wird nur einmal instanziiert, und dann als globales Objekt an das `JXToolkit`-Objekt angehängt. Momentan bietet die Klasse `SlaveWindowHandler` nur Unterstützung für `Choice`-Klassen, ist aber beliebig erweiterbar.

Die Klasse wird nicht verwendet, um die offenen Fenster eines evtl. vorhandenen Menüsystems zu verwalten. Da das Menüsystem intern ganz anders funktioniert als die “normalen” Komponenten, wurden dafür auch eigene Mechanismen entwickelt, welche die anfallenden Aufräumarbeiten des Menüsystems durchführen (vgl. Abschnitt 4.6.4.1).

4.5.3 Die Klasse JXScrollbarPeer

Dieser Peer ist intern als eine Art Wrapper-Klasse für ein `InternalScrollbar`-Objekt aufgebaut: Wenn er ein Ereignis empfängt, ruft er den entsprechenden Ereignishandler des Objekts auf, und reagiert entsprechend auf dessen Rückgabewert. So schiebt er, wenn der Balken gezogen oder die anderen Bereiche angeklickt wurden, entsprechende “Adjustment-Events” in die AWT-Ereigniswarteschlange und informiert so über Wertänderungen der Scrollbar.

Die Implementierung der `paint()`-Methode ist ebenso einfach gehalten: Der Peer zeichnet einfach das `InternalScrollbar`-Objekt neu, wobei vorher noch der Grundzustand des Peers ausgewertet wird.

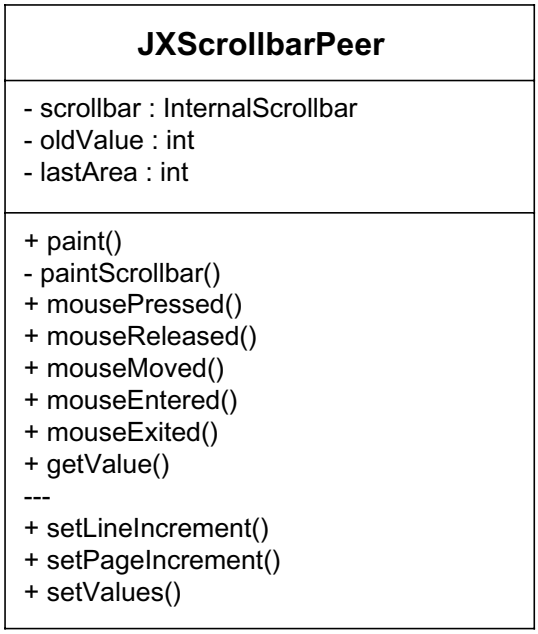


Abb. 4.21 Die Klasse JXScrollbarPeer

4.5.4 Die Klasse JXListPeer

Eine Liste kann auf zwei verschiedene Funktionsweisen eingestellt werden: Entweder man kann nur genau einen Eintrag auf einmal markieren, oder es ist eine beliebige Untermenge der Listenelemente auswählbar. Um beide Arten realisieren zu können, bedient sich der Peer der Hilfsklasse `JXListElement`. Diese stellt im Grunde nur einen eigens definierten Datentyp dar, der einen Text sowie ein `selected`-Flag speichern kann. Der Peer verwaltet intern einen Vektor dieses Datentyps, in dem alle Listeneinträge sowie deren Flags gespeichert sind. Durch entsprechende Verwaltung lassen sich so beide Funktionsweisen des Peers realisieren.

Anmerkung: Die beiden verschiedenen "Betriebsarten" werden im Folgenden als Einfach- bzw. Mehrfachmodus bezeichnet.

Die Handlermethode `keyPressed()` des Peers regelt alle Aktionen, welche durch bestimmte Tastendrücke ausgeführt werden müssen. Um auf der Liste mit der Tastatur arbeiten zu können, gibt es einen sichtbaren Cursor, der mit den Cursortasten bewegt werden kann. Befindet sich die Liste im Einfachmodus, so wird auch gleich die aktuelle Auswahl auf den Cursor gesetzt und somit ebenfalls bewegt. Im Mehrfachmodus lässt sich die Auswahl nur modifizieren, wenn man die <**Space**>-Taste drückt: Damit wird das `selected`-Flag des Eintrags unter dem Cursor invertiert. Beim Drücken auf <**Enter**> bzw. <**Return**> wird dagegen einfach ein "Action-Event" an die AWT weitergeschickt.

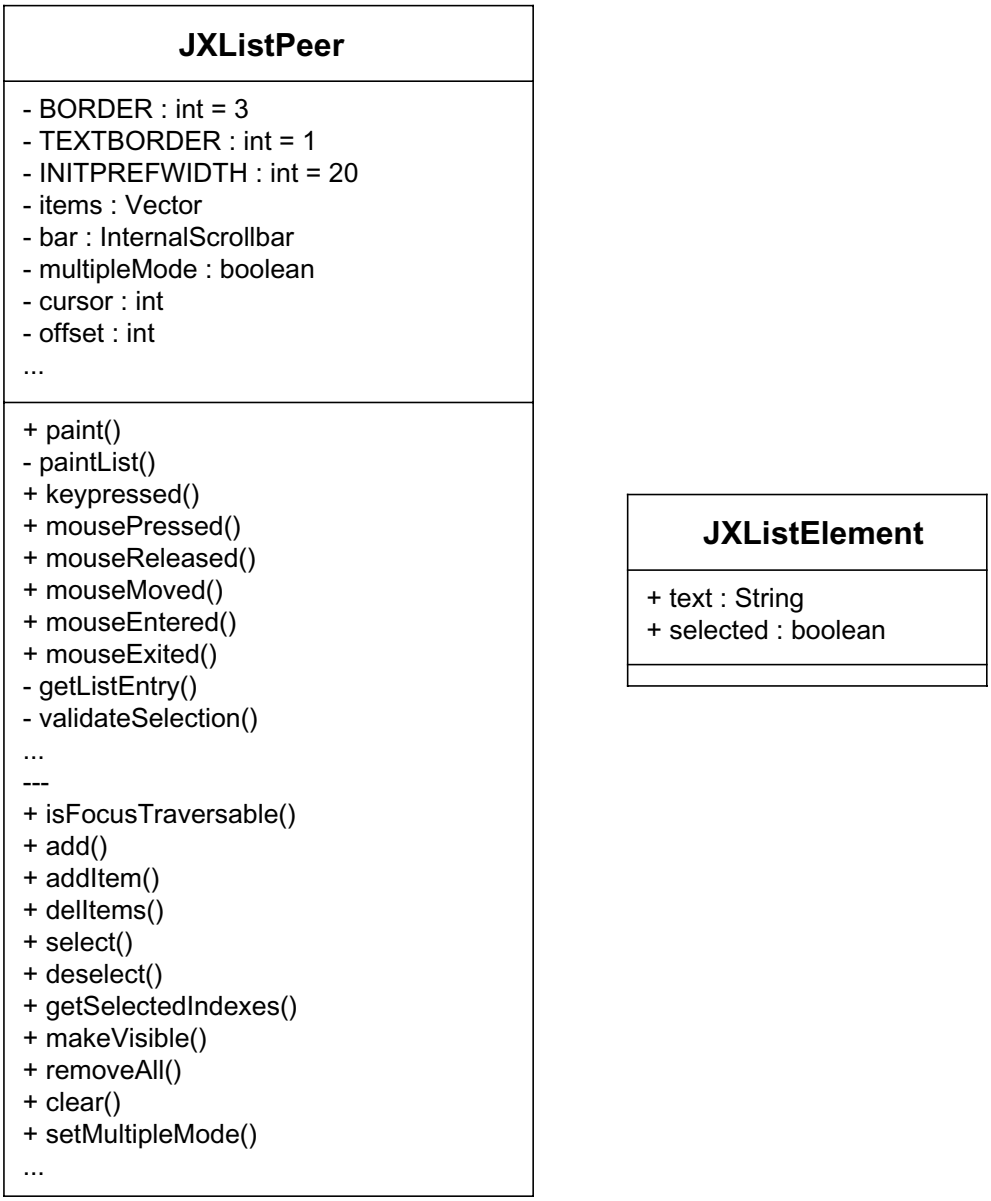


Abb. 4.22 Die Klassen JXListPeer und JXListElement

Sollen im Mehrfachmodus mehrere Einträge, die hintereinander liegen, auf einmal ausgewählt werden, so kann man dazu die <**Shift**>-Taste verwenden: Drückt man sie, so wird die aktuelle Cursorposition intern gespeichert. Wird unter gedrückter <**Shift**>-Taste der Cursor auf eine andere Position bewegt und dort die <**Space**>-Taste gedrückt, so werden alle Einträge zwischen gespeicherter und aktueller Cursorposition ausgewählt.

Der Peer besitzt neben einem sichtbaren Bereich für einen Ausschnitt der Liste auch einen internen Scrollbalken, welcher es ermöglicht, den sichtbaren Listenausschnitt zu wählen. Intern besitzt der Peer dazu ein `InternalScrollbar`-Objekt.

Tritt nun ein (Maus-)Ereignis ein, so wird der jeweilige Handler des Peers aufgerufen. Dieser prüft zuerst, ob die Maus im Bereich des Scrollbalkens liegt. Ist dies der Fall, so wird der entsprechende Handler des `InternalScrollbar`-Objekts aufgerufen, und danach das Aussehen des Peers dem Rückgabewert entsprechend modifiziert. Liegt die Maus dagegen im Bereich des Listenausschnitts, sind nur Ereignisse interessant, welche die Maustasten betreffen: Wurde eine Taste gedrückt, so wird der Cursor auf den Eintrag gesetzt, welche unter der aktuellen Mausposition steht. Eine losgelassene Maustaste wird in etwa wie ein Druck auf die <**Space**>-Taste interpretiert: Im Einfachmodus wird die Auswahl auf die Cursorposition gesetzt, im Mehrfachmodus wird der Eintrag unter dem Cursor invertiert.

Der Peer führt neben der Verarbeitung der Ereignisse auch andere Arbeiten durch, z.B. sorgt er dafür, dass der Cursor immer im sichtbaren Bereich bleibt bzw. der sichtbare Bereich sich der Cursorposition anpasst. Außerdem wird, um zum AWT-Standard kompatibel zu bleiben, bei jeder Modifikation der Listeneinträge ein “Item-Event” an die AWT geschickt.

4.5.5 Die Klasse JXTextAreaPeer

Die Klasse `JXTextAreaPeer` stellt eine umfangreichere und komplexere Version der Klasse `JXTextFieldPeer` dar: Sie implementiert einen mehrzeiligen Textbereich, dessen sichtbarer Teilausschnitt durch interne Scrollbalken verschoben werden kann, sofern diese angezeigt werden.

Durch die Verwandtschaft mit der Klasse `JXTextFieldPeer` laufen auch viele Aktionen weitestgehend gleich ab. So ist z.B. die Bedienung mit der Tastatur bei beiden Peers gleich realisiert, indem bei beiden die geerbte Methode `performTextEditing()` verwendet wird (vgl. Abschnitt 4.4.5).

Auch die Behandlung von Mausereignissen ist an sich gleich implementiert, mit dem Unterschied, dass die Klasse `JXTextAreaPeer` dabei noch berücksichtigen muss, dass sie u.U. noch zwei interne Scrollbalken zu verwalten und darzustellen hat. Erhält die Klasse ein Mausereignis, so wird zuerst geprüft, ob sich die Maus im Bereich eines der sichtbaren Scrollbalken befindet. Ist das der Fall, so wird die entsprechende Handlermethode des jeweiligen `InternalScrollbar`-Objekts aufgerufen, und anschließend, falls notwendig, der sichtbare Ausschnitt dem Rückgabewert gemäß angepasst. Befindet sich die Maus dagegen über dem Textbereich, so wird das Ereignis völlig analog zur Klasse `JXTextFieldPeer` abgearbeitet (vgl. hierzu Abschnitt 4.4.6).

Um mit dem darzustellenden Text besser umgehen zu können, verwaltet der Peer intern einen Vektor aus `String`-Objekten, welche die einzelnen Textzeilen enthalten. Auf diese Weise mus der Text nicht bei jedem Zeichenvorgang des Peers erst in Textzeilen geschnitten, sondern kann direkt aus dem Vektor entnommen werden. Dies erlaubt eine einfache zeilenweise Bearbeitung des Textes und vermeidet das unnötige Erzeugen von temporären Objekten, wie sie sonst beim Schneiden des Textes entstehen würden. Für die Aktualisierung dieses Vektors ist die Methode `handleTextAltered()` zuständig (vgl. Abschnitt 4.4.5). Diese wird bei jeder Änderung

des Textinhaltes aufgerufen, zerlegt in der vorliegenden Implementierung den Text in seine Zeilen und legt diese im Vektor ab. Somit ist die Erzeugung temporärer Objekte auf diese Methode beschränkt.

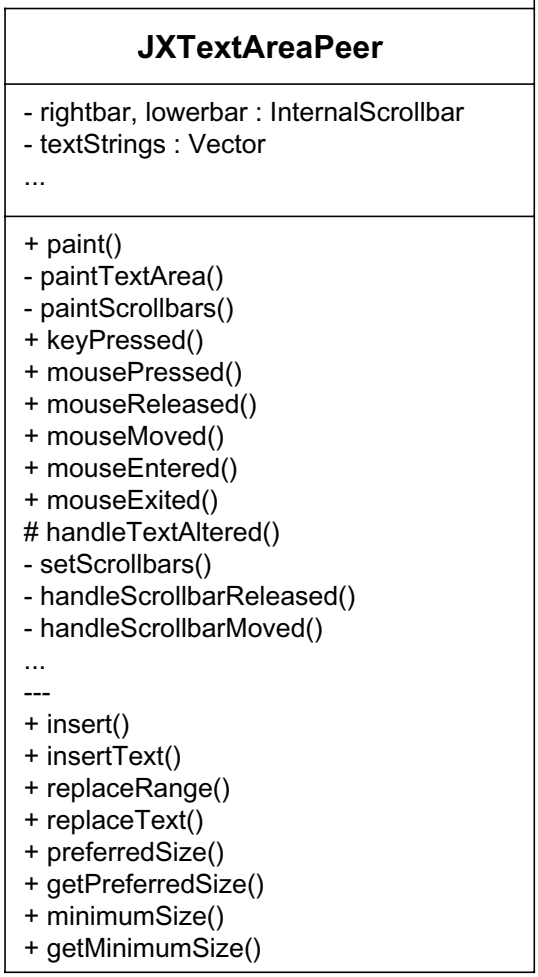


Abb. 4.23 Die Klasse JXTextAreaPeer

Die Methode `handleTextAltered()` wird dabei nur aufgerufen, wenn sich der Inhalt des darzustellenden Textes ändert. Änderungen z.B. bei der sichtbaren Größe der `TextArea`-Komponente oder bei der aktuellen Position des sichtbaren Ausschnitts, führen dagegen nicht zu einem Aufruf der Methode, da hier die Daten im `String`-Vektor nicht modifiziert werden müssen.

4.5.6 Die Klasse JXChoicePeer

Der Peer für die Klasse `Choice` besitzt als einziger eingebetteter Peer neben seinem Bereich im Mutterfenster ein eigenes Fenster, in welches er zeichnen kann (vgl Abschnitt 4.5.2). Um dieses Fenster zu erzeugen, kennt er einen eigenen Konnektor, der in der Klasse `JXChoice-`

`Connector` realisiert wird (siehe Abbildung 4.24). Dieser Konnektor realisiert das Hilfsfenster der Komponente `Choice` und stellt alle Funktionen zur Verfügung, um damit arbeiten zu können.

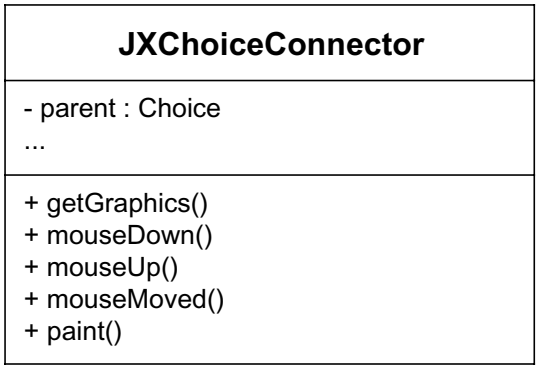


Abb. 4.24 Die Klasse JXChoiceConnector

Ein Problem, welches sich bei der Realisierung eines Hilfsfensters ergibt, ist die Verwaltung des Fensters durch den Windowmanager. Schließlich sollte durch Öffnen des Hilfsfensters das Mutterfenster nicht den Fokus verlieren, d.h. weiterhin als aktiv gekennzeichnet bleiben. Dies ist deshalb notwendig, weil nur das aktive Fenster Tastaturereignisse vom Windowmanager bekommen und darauf reagieren kann. Wäre das Hilfsfenster aktiv, so könnte das Mutterfenster nur durch Schließen des Hilfsfensters wieder mit der Tastatur angesprochen werden. Dieser Zustand ist jedoch nicht sehr intuitiv für den Benutzer und schränkt ihn in seinen Möglichkeiten ein. Davon abgesehen ist es semantisch nicht ganz korrekt, dass das Mutterfenster wegen eines abhängigen Hilfsfensters seinen Fokus verliert.

Aus diesem Grund wurde in den Windowmanager ein neues Fensterflag namens WND_NO_FOCUS eingebaut. Dieses Flag verhindert, dass ein Fenster aktiv werden kann. Die Klasse `JXChoiceConnector` initialisiert sich selbst mit diesem Flag, so dass ihr Fenster niemals den Fokus erhalten kann. Zusätzlich setzt sie noch das Flag WND_NO_BORDER, was bedeutet, dass das Hilfsfenster keinen Rahmen besitzt. Auf diese Weise ist es nun möglich, ein Fenster zu erzeugen, das nichts am Zustand des Mutterfensters ändert, und einfach nur einen zeichenbaren Bereich auf dem Bildschirm darstellt.

Als nächstes stellt sich die Frage nach der Interaktion mit dem Hilfsfenster. Der Windowmanager schickt alle Tastaturereignisse an das aktive Fenster, und Mausereignisse an alle Fenster, die sich unterhalb des Mauszeigers befinden. Das bedeutet, wenn eine `Choice`-Komponente ihr Hilfsfenster geöffnet hat, besitzt das Mutterfenster noch den Tastaturfokus, und alle Tastaturereignisse gehen an den Konnektor des Mutterfensters. Die Mausereignisse gehen, je nachdem wo sich die Maus gerade befindet, an den Konnektor des Mutterfensters, an den Konnektor der `Choice`-Komponente, oder an beide. Das heißt, dass der Peer sowohl die Tastaturereignisse für seinen Bereich im Mutterfenster, als auch für das Hilfsfenster bekommt, und diese dementsprechend behandeln muss.

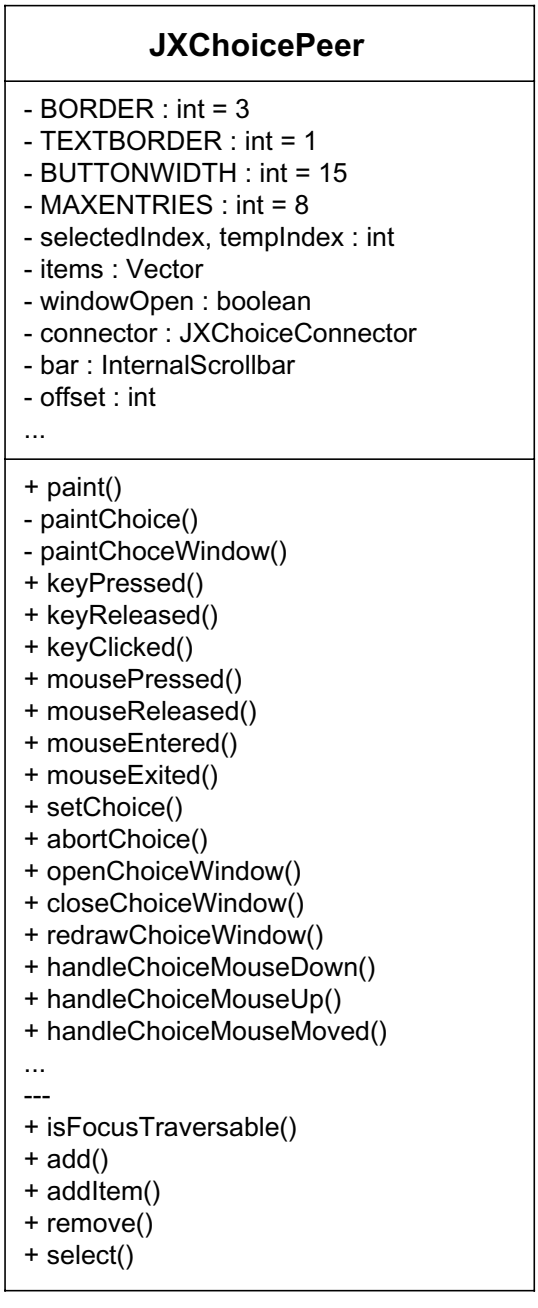


Abb. 4.25 Die Klasse JXChoicePeer

In der vorliegenden Implementierung wurde dieses “Problem” so gelöst, dass der Peer neben den normalen Ereignishandlern, die er von `JXComponentPeer` geerbt hat, noch einige weitere für Mausereignisse im Hilfsfenster kennt. Die Handler werden von der Klasse `JXChoiceConnector` aufgerufen, sobald ein entsprechendes Ereignis im Hilfsfenster aufgetreten ist. Somit leitet der Konnektor alle Ereignisse einfach an der Peer weiter, und dieser verwaltet alle Ereignisse, die ihn bzw. seinen Konnektor betreffen. Abbildung 4.26 skizziert diese Nachrichtenverteilung.

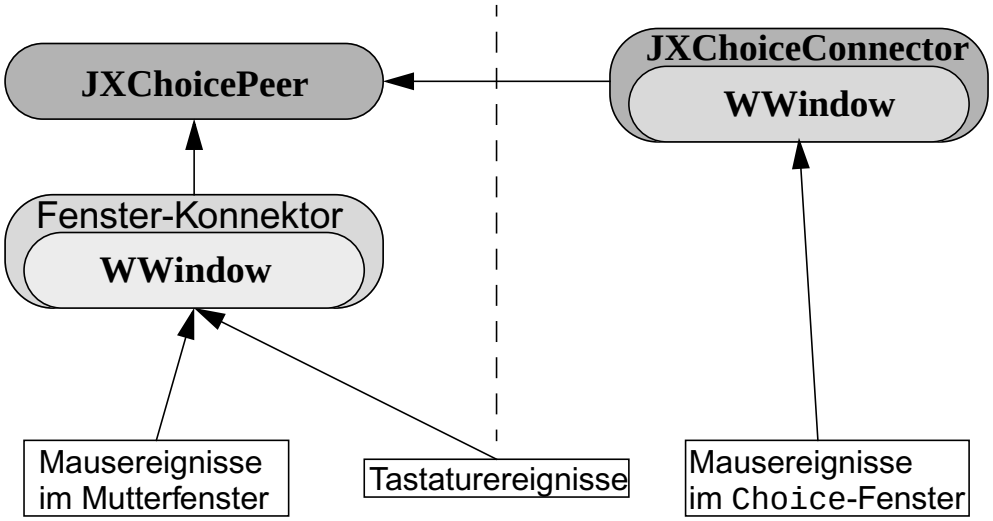


Abb. 4.26 Die Verteilung der Ereignisse bei der Choice-Peer-Implementierung

Der Peer bekommt alle Tastaturevents, die ihn irgendwie betreffen, muss diese auswerten und entscheiden, was getan werden soll, unter Berücksichtigung, ob das Hilfsfenster offen ist oder nicht. Dementsprechend sind die Ereignishandler des Peers folgendermaßen ausgelegt: Ein Druck auf **<Space>** oder **<Return>** öffnet das Hilfsfenster, wenn es noch nicht offen sein sollte, ansonsten wird es geschlossen, und der eben ausgewählte Listeneintrag wird selektiert und als neuer Text angezeigt. Wird bei einem offenen Hilfsfenster **<Esc>** gedrückt, wird die Auswahl abgebrochen und das Fenster ebenfalls geschlossen, aber ohne einen neuen Text anzuzeigen. Die Cursortasten erlauben es, sich durch die Listeneinträge zu bewegen. Sollte das Fenster dabei geöffnet sein, so bewegt sich der Listencursor im Fenster entsprechend den Tasten durch die Liste; Bei geschlossenem Fenster entspricht der angezeigte Text der aktuellen Position des Cursors.

Bei den Maushandlern ist keine solche Unterscheidung nötig, da der Peer ja unterschiedliche Handler für Mutter- und Hilfsfenster besitzt. Wird eine Maustaste im Mutterfenster gedrückt, also die Methoden `mousePressed()` bzw. `mouseReleased()` aufgerufen, so wird einfach, falls noch nicht vorhanden, das Hilfsfenster geöffnet. Alle anderen Maus-Ereignisse im Mutterfenster werden in der vorliegenden Implementierung ignoriert.

Für Ereignisse im Hilfsfenster werden folgende Methoden verwendet:

- `handleChoiceMouseDown()`: Maustaste wurde gedrückt
- `handleChoiceMouseUp()`: Maustaste wurde losgelassen
- `handleChoiceMouseMoved()`: Maus wurde bewegt
- `redrawChoiceWindow()`: Das Hilfsfenster muss neu gezeichnet werden

Diese Methoden werden von der Klasse `JXChoiceConnector` bei Bedarf aufgerufen. Bis auf die letzte Methode handelt es sich hierbei um Handler für Mausereignisse. Diese fallen etwas umfangreicher als die “normalen” Maushandler aus, v.a. deshalb, weil das Hilfsfenster evtl.

einen internen Scrollbalken beinhaltet, falls das Fenster nicht die ganze Liste auf einmal darstellen kann. Deswegen muss auch hier jeder Handler erst einmal überprüfen, ob das Ereignis im Bereich des Scrollbalkens stattgefunden hat, und wenn ja, dann dessen entsprechende Handlermethode aufrufen und den Darstellungsbereich anschließend korrigieren. Sollte sich die Maus dagegen im Listenbereich des Fensters befinden, so wird getestet, ob die Cursorposition angepasst werden soll, denn diese folgt der Maus, während sie sich im Fenster bewegt. Drückt der Benutzer im Listenbereich eine Maustaste, so wird das Fenster geschlossen, und der eben angewählte Listeneintrag übernommen.

Da der Peer Bereiche in zwei verschiedenen Fenstern besitzen kann, kennt er auch zwei verschiedene Zeichenmethoden: `paintChoice()` ist für das klassische Zeichnen im Mutterfensterbereich zuständig, während `paintChoiceWindow()` das Hilfsfenster gestaltet. Letztere Methode wird von `redrawChoiceWindow()` aufgerufen, welche ebenfalls das benötigte `JXGraphics`-Objekt dafür bereitstellt (welches wiederum vom Konnektor des Hilfsfensters kommt!).

Das Öffnen des Hilfsfensters

Zum Öffnen eines neuen Hilfsfensters dient die Methode `openChoiceWindow()`. Diese führt nacheinander folgenden Arbeiten durch:

- Das Fenster wird bei der globalen Instanz der Klasse `SlaveWindowHandler` registriert (vgl. Abschnitt 4.5.2).
- Ein neuer Konnektor wird erzeugt.
- Die Größe des Konnektorfensters wird berechnet: Die Breite des Fensters wird gleich der Breite der `Choice`-Komponente im Mutterfenster, und die Höhe des Fensters gleich der Anzahl der darzustellenden Zeilen gesetzt.

  In der Klasse `JXChoicePeer` ist eine maximale Anzahl gleichzeitig darstellbarer Zeilen definiert. Sollte die Liste länger sein als dieses Maximum, so wird die Höhe des Fensters auf das Maximum begrenzt und dem Fenster eine interne Scrollbar hinzugefügt.
- Die Position des Fensters auf dem Bildschirm wird ermittelt: Normalerweise wird das Fenster unterhalb des Mutterfensterbereichs der `Choice`-Komponente dargestellt. Falls das Fenster dabei nicht mehr ganz auf den Bildschirm gezeichnet werden kann, wird die Position angepasst.
- Der Konnektor wird angewiesen, das Fenster sichtbar darzustellen. Dies führt zu einem Aufruf der Methode `redrawChoiceWindow()` durch den Konnektor, die das Fenster entsprechend zeichnet.

4.5.7 Die Klasse JXScrollPanePeer

Die Klasse `ScrollPane` wurde erst mit Version 1.1 der AWT eingeführt, und zeigt gegenüber anderen Komponenten auch ein etwas anderes Grunddesign. Auffälligster Unterschied zu anderen Komponenten ist die Tatsache, dass die Klasse keine internen Scrollbalken verwendet, son-

dern auf `Scrollbar`-Objekte der AWT zurückgreift. Aus diesem Grund verwendet die vorliegende Peer-Implementierung auch keine `InternalScrollbar`-Objekte, sondern benutzt die Peers der gerade erwähnten `Scrollbar`-Objekte, um die Scrollbalken zu steuern. Auf diese Weise überträgt sich das etwas außergewöhnliche Design der `ScrollPane`-Komponente auch auf deren Peer. Abbildung 4.27 zeigt den Peer im Überblick.

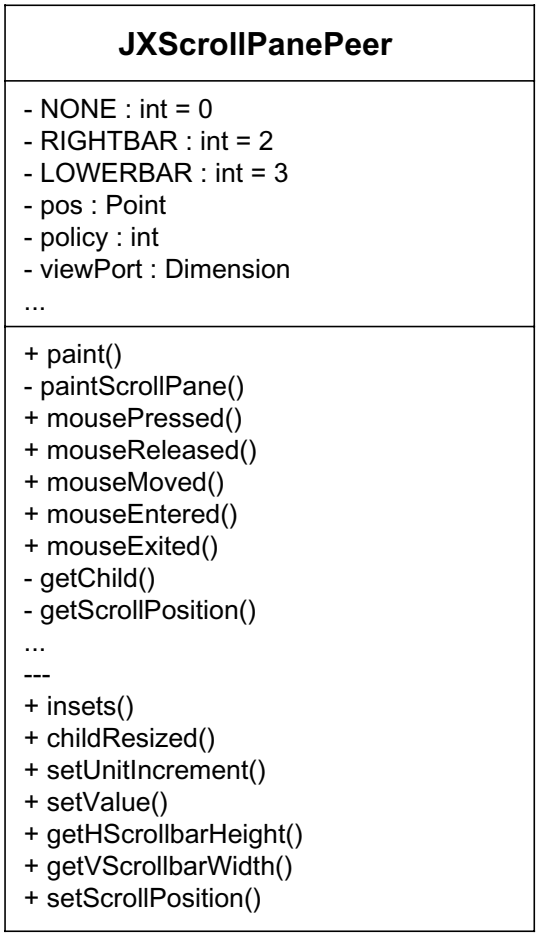


Abb. 4.27 Die Klasse JXScrollPanePeer

Eine `ScrollPane`-Komponente ist ein spezieller Kontainer, der eine einzige Komponente beinhalten kann. Deswegen muss der zugehörige Peer nur sich selbst verwalten und zeichnen, denn die eingebettete Komponente verwaltet sich durch die AWT selbst. Die Verwaltung beschränkt sich folglich auf die sichtbaren Scrollbalken, welche die Komponente momentan besitzt.

Der sichtbare Bereich der Komponente besitzt, je nach Konfiguration[2], bis zu zwei Scrollbalken, welche durch `Scrollbar`-Objekte dargestellt werden. Eine Möglichkeit, diese zu verwalten, wäre gewesen, die Objekte als weitere eingebettete Komponenten in den Kontainer zu

2. Bei der Klasse `ScrollPane` ist es möglich einzustellen, wann die Scrollbalken sichtbar sein sollen: Immer, nie, oder bei Bedarf. Bei Letzterem kann auch nur ein Balken sichtbar sein.

stecken und diese von ihm mitverwalten zu lassen. Dies hätte jedoch einige Umarbeiten der aktuellen Classpath-Implementierung der `ScrollPane`-Klasse zur Folge gehabt, und außerdem müsste dann die Ereignisbehandlung für die Scrollbalken durch AWT-Eventlistener geregelt werden, was kein guter Programmierstil wäre.

Deswegen wurde eine andere Methode gewählt: Beide Objekte werden durch ihre Peers direkt angesprochen und verwaltet, und nur die eingebettete Komponente ist auch wirklich eingebettet. Um die Scrollbalken direkt ansprechen zu können bzw. auf Ereignisse dafür zu reagieren, muss dafür gesorgt werden, dass die Klasse `JXScrollPanePeer` die Mausereignisse bekommt, sobald die Maus sich über einem der Scrollbalken befindet. Dies wird dadurch erreicht, dass der Rahmen des `ScrollPane`-Peers angepasst wird: Die eingebettete Komponente liegt stets im kompletten Komponentenbereich des Kontainers, und die Scrollbalken liegen immer im Rahmenbereich des Kontainers. Auf diese Weise werden alle Mausereignisse an die eingebettete Komponente geschickt, sobald die Maus sich darüber befindet, und an den Peer, sobald sich die Maus über dem Rahmenbereich bzw. den Scrollbalken befindet (vgl. Abschnitt 4.7.2.1).

Abbildung 4.28 zeigt die entstehende Aufteilung der Zeichenfläche, wenn beide Scrollbalken sichtbar sind. Ist ein Balken unsichtbar, so wird der Komponentenbereich um den entsprechenden Rahmenbereich erweitert bzw. die Rahmendicke hier auf null gesetzt.

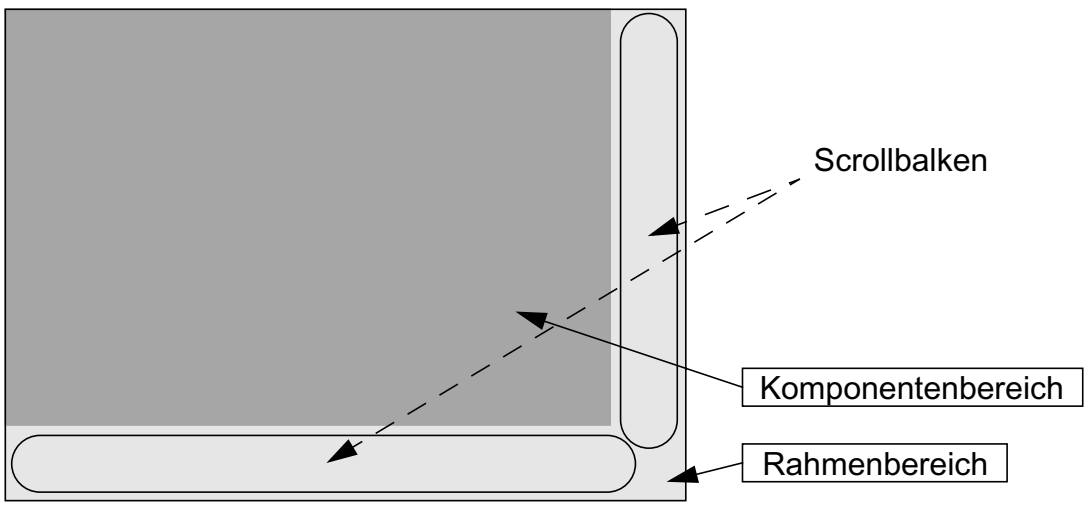


Abb. 4.28 Aufteilung des Zeichenbereichs der Klasse ScrollPane

Die Behandlung der Mausereignisse konzentriert sich demnach auch nur auf die Weiterleitung der Ereignisse an den entsprechenden Scrollbalken, falls sich die Maus gerade über einem befindet. Es werden jedoch, im Gegensatz zu den anderen Peers, keine Methoden der Klasse `InternalScrollbar` verwendet (da der Peer ja kein Objekt dieser Klasse kennt), sondern die entsprechenden Handlermethoden des jeweiligen `Scrollbar`-Peers aufgerufen.

Das Zeichnen des Peers verläuft ein wenig komplizierter: Der Peer muss mit einer variablen Anzahl von Scrollbalken (0 - 2) umgehen können. Um dies zu bewerkstelligen, ruft er vor jedem Zeichenvorgang die private Methode `resetLayout()` auf. Diese Methode gestaltet das kom-

plette Layout des Peers und regelt u.a. auch die Sichtbarkeit der einzelnen Scrollbars, je nach Konfiguration der `ScrollPane`-Komponente. Dabei wird auch berücksichtigt, dass das Erscheinen bzw. Verschwinden eines Scrollbalkens sich auf das Layout auswirken und den anderen Scrollbalken notwendig bzw. überflüssig machen kann. Das Zeichnen an sich besteht dagegen im Grossen und Ganzen darin, die sichtbaren Scrollbalken auf den Rahmen zu zeichnen.

Anpassung der anderen Peer-Klassen

Die Aufgabe der Klasse `ScrollPane` ist es, einen Ausschnitt der eingebetteten Komponente darzustellen, und diesen Ausschnitt mittels Scrollbalken oder Methodenaufrufen zu verschieben. Diese Aufgabe wird mit den bisherigen Betrachtungen des Peers noch nicht gelöst. Tatsächlich bewirkt der Peer in der vorliegenden Implementierung auch nichts Konkretes in dieser Richtung, außer dass er den aktuellen Offset des Ausschnitts speichert.

Im Grunde kann der Peer diese Aufgabe auch nicht alleine lösen, denn die eingebettete Komponente ist, was das Zeichnen angeht, vom Peer vollkommen unabhängig. Deswegen sind die grundlegenden Klassen der JX-Implementierung, welche das Zeichnen von Peers steuern, entsprechend angepasst, um dieses andere Zeichenverfahren unterstützen zu können. Es handelt sich dabei um die Klassen `JXGraphics` und `JXComponentPeer`.

Die Klasse `JXGraphics` besitzt dazu die Methode `setViewClip()`. Diese erlaubt es, einen eigenen Clip-Bereich zu setzen, unabhängig von dem der Methode `setClip()`, und der Komponente, welche das `JXGraphics`-Objekt anfordert, vollkommen unbekannt. Wurde dieser besondere Clip-Bereich gesetzt (im Folgenden als “Super-Clip” bezeichnet), so wird jede weitere Grafikausgabe auf diesen beschränkt, und jeder weitere “normale” Clip-Bereich mit diesem geschnitten. Somit kann ein Bereich definiert werden, in welchem sich die Grafikausgabe eines Peers beschränkt, und der für den Peer selbst vollkommen transparent ist.

In der Klasse `JXComponentPeer` unterstützt die Methode `getJXGraphics()` die Funktionsweise der Klasse `ScrollPane` (vgl. Abschnitt 4.2.2): Kommt die Methode beim Durchsuchen der Peer-Hierarchie nach dem Mutterfenster an einem `ScrollPane`-Objekt vorbei, so wird bei diesem bzw. dessen Peer die aktuelle Größe des Komponentenbereichs und der aktuelle Offset des sichtbaren Ausschnitts abgefragt und gespeichert. Wurde dann ein passendes `JXGraphics`-Objekt gefunden, so wird neben den üblichen Einstellungen daran noch der “Super-Clip” auf den Komponentenbereich der `ScrollPane`-Komponente gesetzt, sowie der Koordinatenursprung dem Offset angepasst. Anschließend wird das Objekt wie gehabt zurückgeliefert.

Durch diese Modifikationen des `JXGraphics`-Objekts durch die Methode `getJXGraphics()` verläuft das Scrollen einer Komponente für diese vollkommen transparent, und auch die Interaktion mit der Komponente wird davon nicht beeinflusst.

4.6 Peers für das Menüsystem

Die vorherigen Kapitel haben alle Peers von AWT-Komponenten beschrieben, welche in einem Fenster (oder einem Applet) plaziert werden können. Neben diesen “normalen” Komponenten gibt es unter der AWT noch eine zweite Gruppe: Die der Menü-Komponenten. Diese sind für die Verwaltung eines Menüsystems gedacht, welches nach dem “Pull-Down”-Prinzip arbeitet: Ein Fenster besitzt eine Menüleiste mit verschiedenen Einträgen, wobei jeder Eintrag bei Auswahl ein eigenes Menü herausklappt, aus welchem weitere Einträge ausgewählt werden können. Diese Menüs können als Einträge auch andere Untermenüs beinhalten, und so eine beliebig tiefe Menühierarchie aufbauen. Durch dieses Prinzip lässt sich die Vielfalt an Interaktionsmöglichkeiten eines Programms effektiv und platzsparend verwalten.

Die Schnittstelle zu dem Komponenten eines Menüsystems ist unter der AWT relativ spartanisch gehalten, ebenso wie die Verwaltung derselben. Ein wichtiger Punkt ist zum Beispiel, dass es für Menü-Komponenten keinen genau bestimmbaren Zeitpunkt gibt, wann deren Peers erstellt werden sollen, zumindest ist in der verwendeten Version der Classpath AWT keine Methode implementiert, welche diese Aufgabe erledigt.

Die konkrete Implementierung eines Menüsystems auf Peer-Ebene bleibt vollständig dem Entwickler überlassen. Somit fällt diese Implementierung der Menü-Komponenten in vielerlei Hinsicht aus dem Rahmen dessen, was bei den vorhergehenden Komponenten üblich war, da sie weniger an Konzepte aus der AWT gebunden ist.

4.6.1 Übersicht und grober Aufbau der Menü-Peers

Die AWT unterscheidet grundsätzlich zwischen “normalen” und Menü-Komponenten. Diese Differenzierung zieht sich hinunter bis in die Peer-Schnittstellen, so dass die vorliegenden Implementierungen der Menüsystem-Peers sich ebenfalls grundlegend von den bisherigen Peers unterscheiden.

Bisher wurde bei der Realisierung der Peers der Ansatz gewählt, für jede vorhandene Komponente auch einen entsprechenden Peer zu erstellen. Diese Vorgehensweise war sinnvoll, da jede der bis jetzt besprochenen Komponenten von den anderen unabhängig erzeugt und verwendet werden kann. Somit braucht jede Komponente auch einen Peer, der unabhängig von anderen Peers funktioniert.

Diese Voraussetzung ist bei einem Menüsystem nicht mehr gegeben: Die Klassen `MenuItem` und `CheckboxMenuItem` der AWT sind immer an ein Objekt der Klasse `Menu` oder `PopupMenu` gebunden, und haben in der AWT letztendlich nur den Zweck, einen Eintrag in einem dieser Menüs zu symbolisieren, wofür eigentlich kein eigener Peer notwendig ist. In der vorliegenden Implementierung kommunizieren die Peers der Klassen `Menu` und `PopupMenu` deshalb nicht mit den Peers ihrer eingebetteten Komponenten, sondern nur mit den Komponenten selbst, da diese bereits alle Daten beinhalten, welche für ein Menü relevant sind. Somit existieren in der vorliegenden Implementierung keine Peers für die Klassen `MenuItem` und `CheckboxMenuItem`.

Damit beschränkt sich die Liste der in der vorliegenden Implementierung realisierten Peers auf die folgenden Klassen:

- `JXMenuComponentPeer`: Die Mutterklasse aller Menü-Komponenten
- `JXMenuBarPeer`: Repräsentiert die Menüleiste in einem Fenster
- `JXMenuPeer`: Stellt ein Menü dar
- `JXPopupMenuPeer`: Stellt ein sogenanntes Popup-Menü dar, das an keine Menüleiste, sondern an eine AWT-Komponente gebunden ist

Abbildung 4.29 zeigt die Ableitungshierarchien der Menü-Komponenten und ihrer Peers im Vergleich.

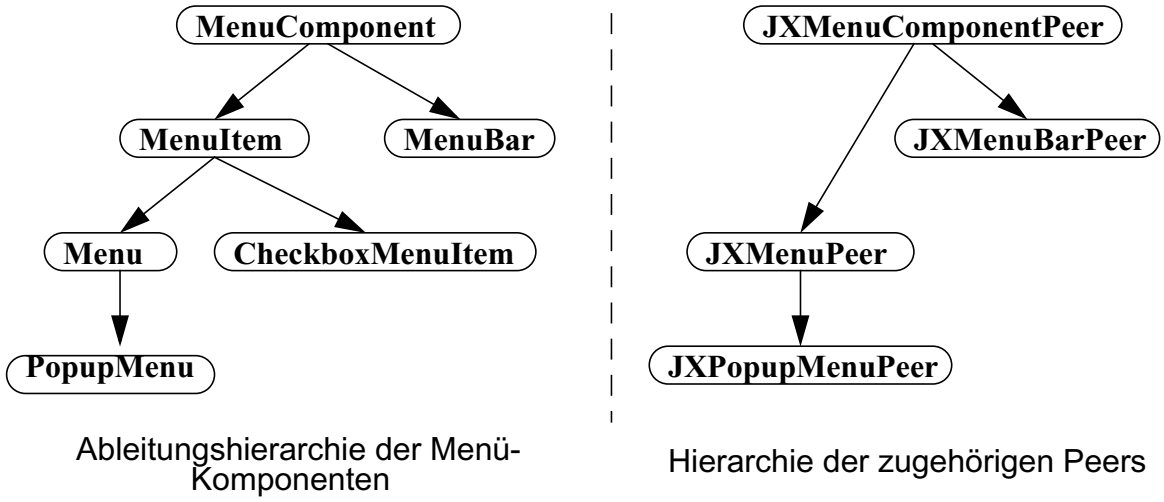


Abb. 4.29 Die Ableitungshierarchien der Menü-Komponenten und deren Peers

4.6.2 Gemeinsame Eigenschaften der Menü-Peers

Wie bereits in Abschnitt 4.3.4 erläutert, besitzt die Klasse `JXMenuComponentPeer` so gut wie keinen Inhalt, da die einzelnen Peers des Menüsystems viel weniger miteinander gemeinsam haben als normale Peers. Außer einem `ready`-Flag und Angaben über die grafischen Dimensionen des Standardzeichensatzes des Windowmanager findet sich deswegen auch nichts weiter in der Klasse (vgl. hierzu Abbildung 4.9).

4.6.3 Unterschiede zwischen Menü und Menüleiste

Die vorliegende Implementierung unterscheidet zwischen einer Menüleiste und einem Menü: Eine Menüleiste ist in ein Fenster eingebunden, welches unter der AWT von einem `Frame`-Objekt repräsentiert wird. Das bedeutet, dass der zugehörige Peer `JXFramePeer` mit der Menüleiste kommuniziert und dafür auch Schnittstellen anbietet. Ein Menü dagegen ist als ein eigenständiges Fenster konzipiert, das gleichberechtigt mit anderen Fenstern des Windowmanagers auf dem Bildschirm zu sehen ist.

Darüber hinaus gibt es noch die sogenannten “Popup-Menüs”, also Menüs, die an keine Menüleiste, sondern an eine AWT-Komponente gebunden sind. Auch diese Tatsache macht es sinnvoll, Menü und Menüleiste voneinander zu trennen, so dass “normale” und Popup-Menüs auf gleiche Weise gehandhabt werden, und die Menüleiste nur im entsprechenden Fall verwaltet wird.

4.6.4 Allgemeine Implementierung der Menüs

Menüs werden unter JX als rahmenlose Fenster realisiert, welche keinen Fokus bekommen können, ganz analog zur Realisierung der Hilfsfenster für `Choice`-Peers (vgl. Abbildung 4.5.6). Die Realisierung als eigenständiges Fenster ist notwendig, da ein Menü meist aus seinem Mutterfenster herausragen kann, und dort ebenfalls richtig gezeichnet werden muss. Außerdem ist es auch hier nicht wünschenswert, dass das “Mutterfenster” den Fokus verliert, denn es sollte immer möglich sein, per Tastatur das Menü zu verlassen und direkt im Mutterfenster weiterzuarbeiten.

Ebenfalls analog zu Abschnitt 4.5.6 ist auch hier das Problem der Tastaturverarbeitung gegeben: Wenn das Mutterfenster aktiv bleibt, bekommt nur dessen Konnektor Tastaturevents zugeschickt. Dieser muss dann folglich die Events auch an ein Menü weiterleiten, falls eines offen ist. Besitzt dieses Menü ein Untermenü, so muss das Ereignis dorthin weitergeleitet werden, usw. Auf diese Weise durchläuft ein Tastaturevent meist den ganzen Menüzweig, ehe es im Blatt bzw. im untersten Menü verarbeitet wird. Abbildung 4.31 zeigt den gesamten Weg, den ein solches Event maximal gehen kann.

Jedes Fenster kennt grundsätzlich zwei Darstellungsformen: Passt ein Fenster komplett auf den Bildschirm, so wird es “normal” gezeichnet. Ist die Liste der Einträge dagegen so groß, dass das komplette Fenster nicht auf den Bildschirm passen würde, so wird das Fenster auf die maximale Bildschirmgröße gekürzt, und erhält zusätzlich je einen “Scroll-Knopf” oben und unten. Durch einen Mausklick auf diese Knöpfe kann der Benutzer durch die Eintragsliste scrollen und den gewünschten Eintrag sichtbar machen.

Ein kleines Flag, das mit eingebaut wurde, ermöglicht es anzugeben, wie das Menüsystem auf Mausbewegungen über einen Untermenüeintrag reagieren soll: Ist dieses `sensitive`-Flag gesetzt, so wird das Untermenü sofort geöffnet oder geschlossen, je nachdem ob der Mauscursor den entsprechenden Eintrag berührt oder nicht. Ist es dagegen gelöscht, so wird ein Untermenü erst durch Anklicken geöffnet.

4.6.4.1 Die Klasse MenuHandler

Eine Möglichkeit, ein Menüsystem zu verwalten, bestünde darin, jedes Fenster mit der Verwaltung seines eigenen Menüsystems zu betreuen. Dies würde jedoch zu einem relativ grossen Overhead bei den entsprechenden Klassen führen: Nicht jedes Fenster benötigt Menüs, und normalerweise ist es auch so, dass nur maximal ein Menüzweig auf dem Schirm sichtbar ist.

Aus diesen Gründen, und um den Zugriff auf ein Menüsystem so einfach wie möglich zu gestalten, wurde die Klasse `MenuHandler` eingeführt. Diese realisiert sozusagen den “Kopf” eines Menüzweigs: So wie jedes Menü sein Untermenü verwaltet, kontrolliert diese Klasse das oberste Menü der Hierarchie, und kann angesprochen werden, wenn es um die Verwaltung der Menühierarchie als Ganzes geht. Auf diese Weise vereinfacht sich der Vorgang, einen kompletten Menüzweig vom Bildschirm zu entfernen und einen neuen aufzubauen. Abbildung 4.30 zeigt die Klasse `MenuHandler` im Überblick.

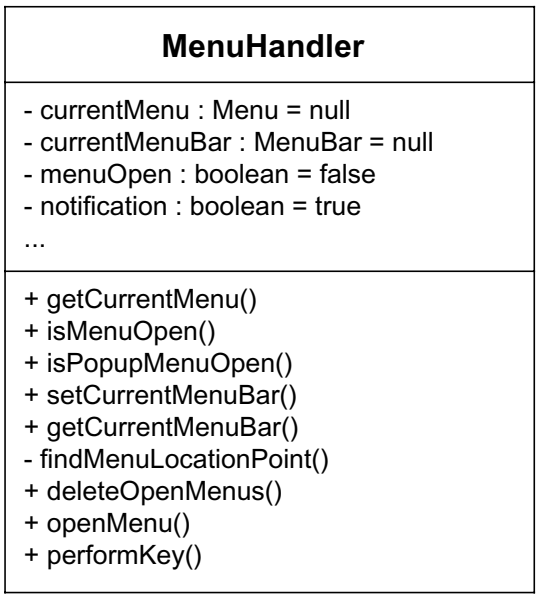


Abb. 4.30 Die Klasse MenuHandler

Das Öffnen und Schließen eines zu verwaltenden Menüs wird durch die Methoden `openMenu()` bzw. `deleteOpenMenus()` erledigt. Beide Methoden arbeiten nach dem gleichen Prinzip, welches noch in Abschnitt 4.6.5.1 beschrieben werden wird, und sollen deswegen hier nicht weiter erläutert werden.

Die Klasse `MenuHandler` verwaltet nur den aktuell sichtbaren Menüzweig, nicht aber eine evtl. vorhandene Menüleiste. Somit unterstützt sie die Trennung zwischen Menüleiste und Menü, und erlaubt es, für normale und Popup-Menüs gleichermaßen eingesetzt zu werden.

Von der Klasse `MenuHandler` gibt es zur Laufzeit nur ein einziges Objekt, welches über die Schnittstelle der globalen `JXToolkit`-Instanz angesprochen werden kann. Die Realisierung als globales Objekt vereinfacht die Verwaltung und geht Hand in Hand mit der Forderung, dass stets nur maximal ein Menüzweig sichtbar sein soll.

Die Klasse ist ebenfalls in den Prozess der Weiterleitung von Tastaturevents eingebunden: Erhält der Konnektor des Mutterfensters ein Tastaturevent, und ein Menü ist offen, so muss das Event an das Menü weitergeleitet werden. Der erste Ansprechpartner dafür stellt dann wiederum die globale Instanz der Klasse `MenuHandler` dar. Der maximale Pfad eines Tastaturevents zu seinem Menü oder der Menüleiste ergibt sich, wie in Abbildung 4.31 skizziert.

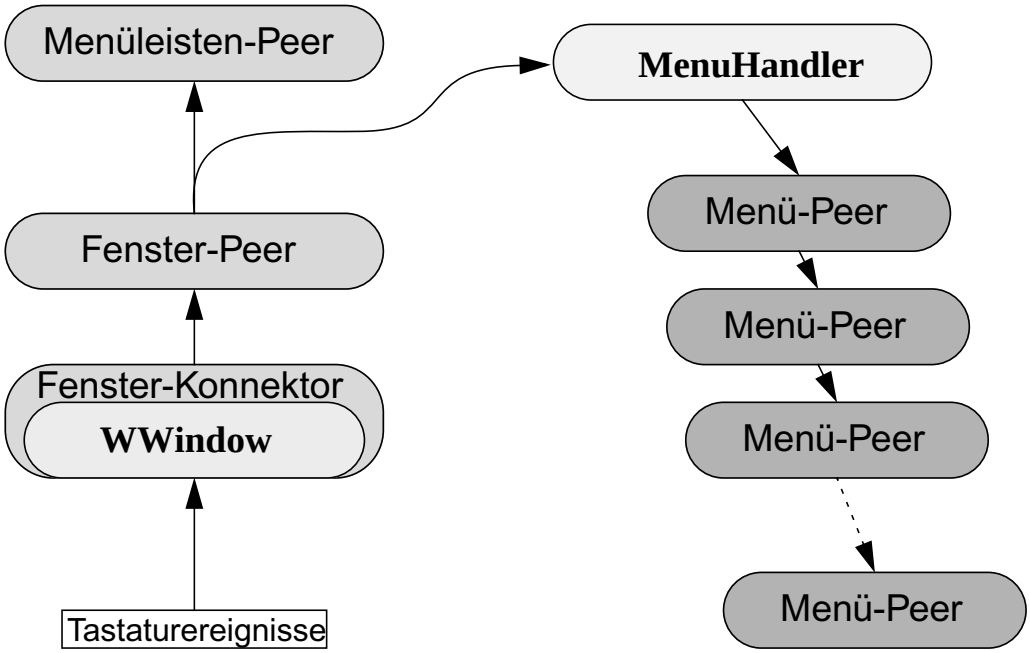


Abb. 4.31 Weg eines Tastaturevents in der Menühierarchie

Für die Weiterleitung der Tastaturevents sorgt die Methode `performKey()`. Diese wird von der Klasse `JXFramePeer` bei Erhalt eines Tastaturevents aufgerufen. Besitzt die Klasse `MenuHandler` ein "Enkel-Menü", d.h. gibt es mindestens zwei Menüs in der sichtbaren Menüstruktur, so wird der Tastencode direkt an die `performKey()`-Methode des hierarchisch obersten Menüs weitergeleitet, damit diese den Code entsprechend verarbeiten oder weiterleiten kann. Gibt es nur ein sichtbares Menüfenster, so verarbeitet die Methode den übergebenen Tastencode folgendermaßen:

Mit der Cursortasten **<up>** oder **<down>** wird der Auswahlbalken des Menüs nach oben oder unten bewegt, genauer gesagt auf das nächste auswählbare Element (Einträge können auch deaktiviert sein). Bei Druck auf **<Return>** wird eine dem gerade ausgewählten Menüeintrag entsprechende Aktion ausgeführt: Handelt es sich bei dem Eintrag um ein Untermenü, so wird dieses geöffnet, und dessen Auswahlbalken auf das erste auswählbare Element gesetzt. Bei einem anderen Eintragstyp wird dessen entsprechende Aktion ausgeführt (vgl. hierzu Abschnitt 4.6.9). Die Cursortasten **<left>** und **<right>** werden ebenfalls an die `performKey()`-Methode des obersten Menüs weitergeleitet, und mit **<Esc>** wird das sichtbare Menü geschlossen.

4.6.5 Konkrete Implementierung der Menüs

Die Klasse `JXMenuPeer` repräsentiert ein komplettes Menü und enthält alles, was dafür notwendig ist: Einfache Hilfsmethoden, Ereignishandler, Methoden zum Layouten und Zeichnen, usw. Abbildung 4.32 zeigt die Klasse im Überblick.

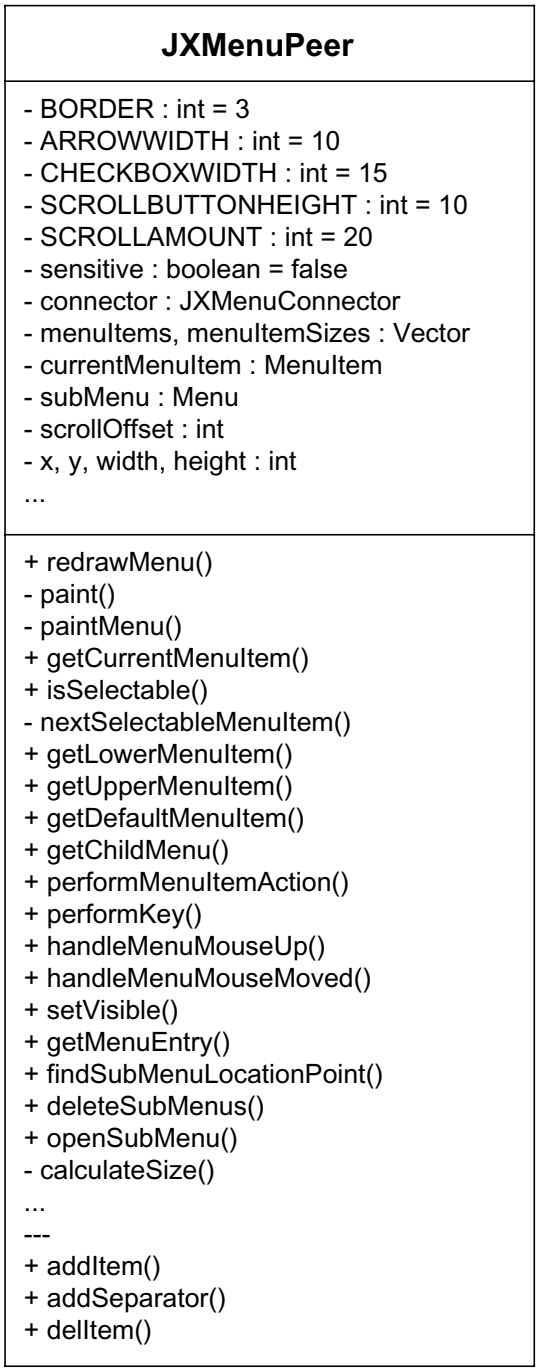


Abb. 4.32 Die Klasse JXMenuPeer

Da ein Menü in einem eigenen Fenster gezeichnet wird, besitzt die Klasse einen Konnektor namens `JXMenuConnector`, welcher das Menüfenster darstellt. Dieser leitet alle Ereignisse, welche für den Peer interessant sind, einfach an diesen weiter bzw. ruft entsprechende Methoden des Peers auf (vgl. Abschnitt 4.5.6). Daneben stellt er die Methode `getGraphics()` zur Verfügung, welche ein `JXGraphics`-Objekt zum Zeichnen in das Fenster zurückgibt. Abbildung 4.33 zeigt den Konnektor im Überblick.

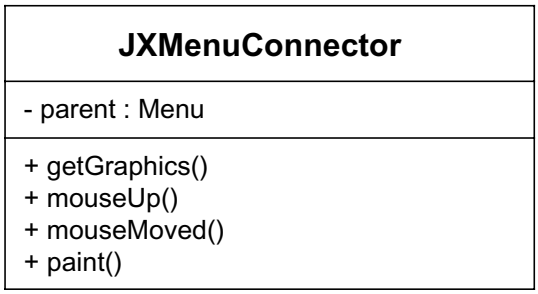


Abb. 4.33 Die Klasse JXMenuConnector

Die Klasse `JXMenuPeer` besitzt intern einige Variablen, allen voran zwei wichtige Vektoren, welche zum Zeichnen und Abfragen der aktuellen Mausposition verwendet werden: Den Vektor `menuItems`, welcher alle `MenuItem`-Einträge beinhaltet, die im Menü dargestellt werden müssen, sowie den Vektor `menuItemSizes`, der die Größe und Position jedes Eintrags im Menü speichert. Daneben kennt sie die Variable `scrollOffset`, welche den Offset angibt, um den die Eintragsliste momentan gescrollt wurde. Außerdem besitzt sie eine Referenz auf ihr momentan offenes Untermenü, falls vorhanden.

Die Klasse kennt eine ganze Reihe von Methoden: Methoden, um den nächsten gültigen Eintrag in der Liste zu finden, um den Scroll-Offset der Auswahlposition anzupassen, usw., um nur einige zu nennen. Die Wichtigsten dabei sind die folgenden Methoden:

- Der Konnektor ruft die Methode `redrawMenu()` auf, sobald er ein Ereignis zum Neuzeichnen empfängt. Diese Methode lässt sich ein neues `JXGraphics`-Objekt geben, und zeichnet das komplette Menü neu.
- Die Methode `handleMouseMoved()` wird vom Konnektor aufgerufen, sobald sich die Maus im Menüfenster bewegt. Sie berücksichtigt bei der Bearbeitung des Ereignisses das `sensitive`-Flag: Ist es gesetzt, so wird ein Untermenü geöffnet oder geschlossen, je nachdem, ob die Maus einen entsprechenden Eintrag betreten oder verlassen hat. In jedem Fall dabei wird der Auswahlbalken auf den neuen Eintrag unter der Maus gesetzt, falls dieser auswählbar bzw. aktiviert sein sollte.
- Die Methode `handleMouseUp()` ruft der Konnektor auf, sobald eine Maustaste innerhalb des Menüfensters losgelassen wurde. Handelt es sich bei dem angeklickten Bereich um einen der Scroll-Knöpfe, so wird der sichtbare Ausschnitt der Liste entsprechend verschoben. Wurde dagegen ein Menüeintrag ausgewählt, so wird eine dem Eintrag entsprechende Aktion ausgeführt (vgl. hierzu Abschnitt 4.6.9).
- Für die Verarbeitung von Tastaturereignissen ist die Methode `performKey()` verantwortlich, deren Ablauf in Abbildung 4.34 dargestellt ist. Diese wird von der globalen Instanz der Klasse `MenuHandler` aufgerufen, falls ein Tastencode irgendwo in einem der Menüs verarbeitet werden soll (vgl. vorherigen Abschnitt). Im Gegensatz zu den Hand-

lern für Mausereignisse verarbeitet diese Methode die Ereignisse nicht ausschließlich für ihr Fenster, sondern auch für das momentan offene Untermenü. Der grobe Algorithmus funktioniert dabei folgendermaßen:

– Besitzt das Menü ein offenes Untermenü, so wird folgende Bedingung geprüft: Wurde die Taste **<Esc>** oder **<left>** gedrückt, und das Untermenü ist das Blatt im Menüzweig (besitzt selbst also kein Untermenü mehr), so wird das Untermenü geschlossen. Ist die Bedingung dagegen nicht erfüllt, so wird der Tastencode an das Untermenü weitergeleitet.

– Besitzt das Menü kein offenes Untermenü, sondern ist selbst das Blatt des Menüzweiges, so kann es den empfangenen Tastencode vollständig auswerten: Wurde die **<Return>**-Taste gedrückt, so wird der aktuell ausgewählte Eintrag entsprechend seines Typs verarbeitet (vgl. vorherigen Abschnitt). Mit den Cursortasten **<up>** und **<down>** wird der Auswahlbalken nach oben bzw. unten bewegt, bei der Cursortaste **<right>** wird, falls der aktuell ausgewählte Eintrag ein Untermenü bezeichnet, dieses geöffnet, ansonsten wird die Methode mit `false` beendet. Die Taste **<left>** beendet die Methode grundsätzlich mit `false`.

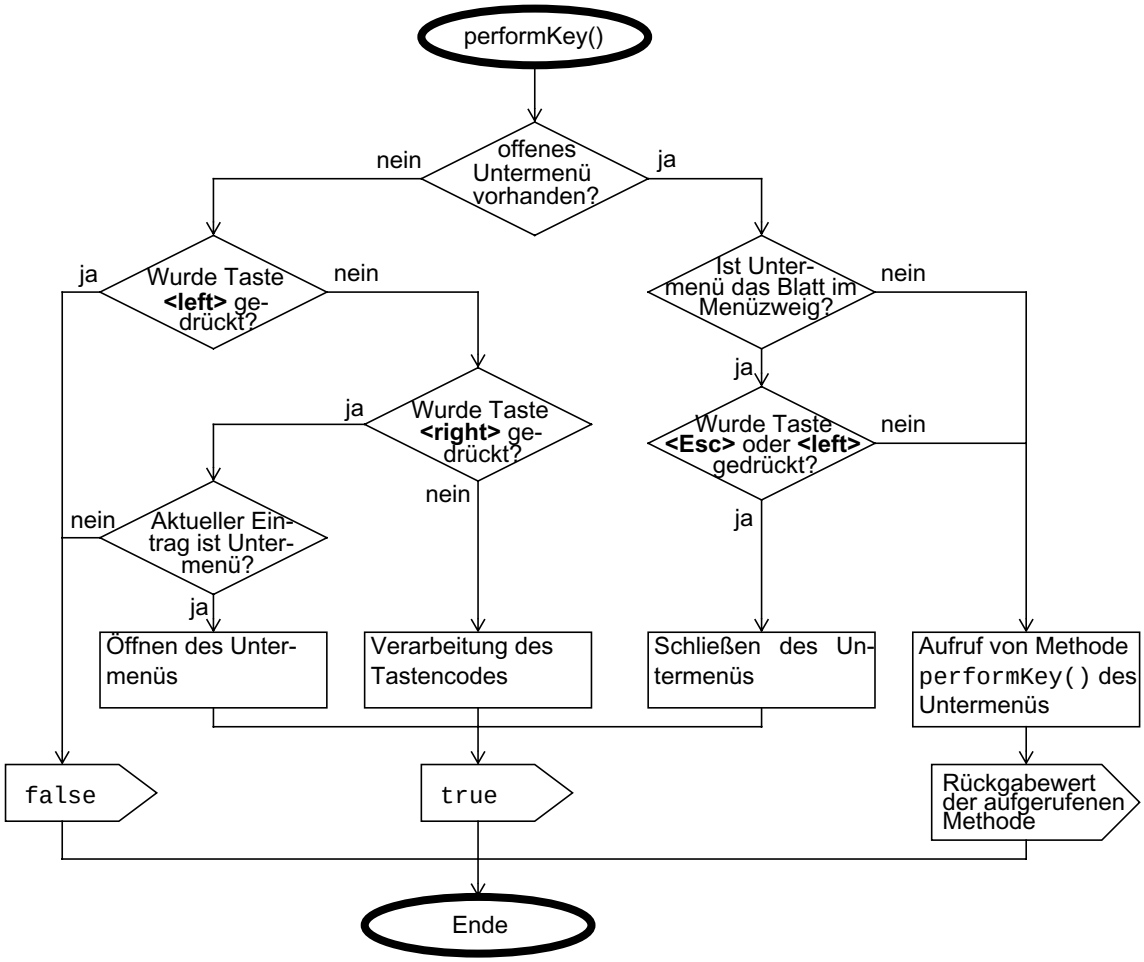


Abb. 4.34 Die Methode performKey()

Die Methode `performKey()` liefert als Rückgabe einen boolschen Wert zurück, der angibt, ob der übergebene Tastencode von der Methode verarbeitet werden konnte oder nicht. Dieses Feedback gibt dem aufrufenden Code die Möglichkeit, selbst den Tastencode auszuwerten, falls die Methode dies nicht bewerkstelligen konnte. Dies ist in bestimmten Situationen sinnvoll, wie noch in Abschnitt 4.7.3.1 gezeigt werden wird.

4.6.5.1 Öffnen und Schließen eines Untermenüs

Für das Öffnen eines Untermenüs ist die Methode `openSubMenu()` verantwortlich. Diese bekommt das zu öffnende `Menu`-Objekt als Parameter übergeben, und arbeitet nach folgendem einfachen Schema:

- Entspricht das zu öffnende Menü dem bereits geöffneten Untermenü, so bricht die Methode ab.
- Ein bereits vorhandenes Untermenü sowie dessen Unterzweig wird geschlossen.
- Ist das übergebene `Menu`-Objekt eine Null-Referenz, so bricht die Methode ab.
- Der Peer des Untermenüs wird mittels `addNotify()` erzeugt.
- Mit Hilfe der Methode `findSubMenuLocationPoint()` wird die optimale Position des Untermenüs auf dem Schirm ermittelt.
- Das Untermenü wird angezeigt.

Der Konstruktor des neu erzeugten Peers führt zuerst die Methode `addItem()` aus, um alle Einträge der zugehörigen Komponente zu übernehmen. Diese füllt die Vektoren `menuItems` und `menuItemSizes` mit entsprechenden Werten. Danach wird die private Methode `calculateSize()` aufgerufen. Diese ist für das gesamte Layout des neuen Menüfensters verantwortlich: Alle Einträge bekommen ihre entgültige Größe und Position zugewiesen, das Fenster wird auf die richtige Größe gesetzt und bekommt, falls notwendig, zwei Scroll-Knöpfe hinzugefügt.

Die Methode `findSubMenuLocationPoint()` liefert die Koordinaten der linken oberen Ecke des Untermenüs wieder. Standardmäßig wird das Untermenü rechts neben dem zugehörigen Eintrag plaziert, sollte es dabei jedoch nicht mehr ganz auf dem Bildschirm zu sehen sein, wird die Position entsprechend angepasst.

Zum Schließen eines Untermenüs wird die Methode `deleteSubMenus()` verwendet. Damit es möglich ist, auch einen ganzen Zweig damit zu schließen, arbeitet die Methode rekursiv:

- Ist ein Untermenü vorhanden, so wird als Erstes dessen Methode `deleteSubMenus()` aufgerufen.
- Danach wird der Peer des Untermenüs mit `removeNotify()` entfernt.

Durch diese Rekursion werden alle Untermenüs, vom Blatt des Menüzweiges angefangen, entfernt.

4.6.6 Implementierung von Popup-Menüs

Die AWT behandelt “normale” und Popup-Menüs unterschiedlich: Während ein normales Menü dem Benutzer nur durch eine Menüleiste zugänglich ist und sonst keinerlei Schnittstellen für den Programmierer kennt, bietet die Komponente `PopupMenu` ein konkrete Methode `show()`, welche das Menü veranlasst, an einer bestimmten Stelle zu erscheinen.

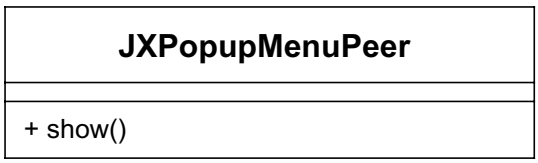


Abb. 4.35 Die Klasse JXPopupMenuPeer

In der vorliegenden Implementierung wird ein Popup-Menü durch die Klasse `JXPopupMenuPeer` repräsentiert. Diese ist von `JXMenuPeer` abgeleitet, und bietet deswegen dieselbe Funktionalität wie ein “normales” Menü. Darüber hinaus implementiert sie noch die vom AWT-Interface `PopupMenuPeer` geforderte Methode `show()`. Diese Methode bekommt eine Komponente und Koordinaten innerhalb der Komponente übergeben, und ist so implementiert worden, dass sie die globale Instanz der Klasse `MenuHandler` anweist, das Menü, welches durch den Peer repräsentiert wird, an der angegebenen Position innerhalb der Komponente zu öffnen.

4.6.7 Erzeugung von Peers der Klassen Menu und PopupMenu

Wie zu Beginn des Kapitels erwähnt, ist es Aufgabe der konkreten Implementierung, zu entscheiden, wann die Peers der jeweiligen Menü-Komponenten erzeugt werden sollen. Die Erzeugung von Peers der Komponente `Menu` überträgt die vorliegende Implementierung dabei den Klassen `MenuHandler` und `JXMenuPeer`: Die Klasse `MenuHandler` erzeugt den Peer für das oberste Menü, und jeder Menü-Peer wiederum erzeugt den Peer für das nächste Untermenü, und das jeweils erst dann, wenn das entsprechende Menü auf dem Bildschirm dargestellt werden soll. Beim Entfernen von Peers verläuft das Ganze analog.

Dies funktioniert bei `Menu`-Objekten ohne Weiteres richtig, da der Aufhängepunkt des Ganzen, also die Klasse `MenuHandler`, bereits beim Start der JX-Implementierung instanziiert wird und somit immer angesprochen werden kann.

Diese Vorgehensweise lässt sich jedoch nicht ohne Weiteres auf Komponenten des Typs `PopupMenu` übertragen. Ein Objekt der Klasse `PopupMenu` wird in der AWT durch die Methode `add()`, die jede “normale” AWT-Komponente besitzt, an diese gebunden. Danach kann das Popup-Menü mittels dessen Methode `show()` dargestellt werden.

Das Problem dabei war die konkrete Implementierung der Klasse `PopupMenu` in der vorliegenden Version der Classpath AWT: Deren Implementierung der Methode `show()` geht davon aus, dass bei Aufruf bereits ein zugehöriger Peer besteht, dessen entsprechende `show()`-Me-

thode aufgerufen werden kann. Das bedeutet, dass der Peer irgendwann vor dem Aufruf der Methode `show()` erstellt werden muss. Um diese Bedingung zu erfüllen, gibt es zwei verschiedene Möglichkeiten:

(1) Um zu der gerade erwähnten Vorgehensweise bei der Erzeugung von Menü-Peers kompatibel zu bleiben, müsste die Methode `show()` der AWT-Komponente `PopupMenu` die JX-Klasse `MenuHandler` anweisen, ein neues Menü zu erstellen. Dies würde es erlauben, den Peer des Menüs erst dann zu erstellen, wenn er auch benötigt wird, ist jedoch ein inakzeptables Vorgehen, da hier eine Spezialisierung der Classpath AWT auf eine bestimmte, systemabhängige Implementierung vorgenommen wird.

(2) Der Peer der Klasse `PopupMenu` muss schon vorher erstellt werden. Auf diese Weise bleibt die Methode `show()` der Klasse unmodifiziert, jedoch muss dann die Klasse `MenuHandler` angepasst werden.

Sinnvollerweise wurde in der vorliegenden Implementierung der zweite Weg gewählt. Dazu wurde die Methode `addNotify()` der AWT-Klasse `Component` dahingehend modifiziert, dass bei Erstellung eines Peers auch gleich die Peers der damit verbundenen Popup-Menüs erzeugt werden. Die Methode `removeNotify()` wurde analog verändert. Auf diese Weise existieren auf Peer-Ebene alle Popup-Menüs so lange, wie ihre zugehörigen Komponenten existieren.

Die Klasse `MenuHandler` wurde nun dahingehend modifiziert, dass ihr durch ein Flag mitgeteilt wird, ob der zu verwaltende Menüzweig zu einem normalen oder einem Popup-Menü gehört. Bei ersterem wird der Peer des obersten Menüs erzeugt oder gelöscht, sobald der Menüzweig dargestellt bzw. entfernt werden soll. Im zweiten Fall dagegen wird der Peer nicht manipuliert.

Die Klasse `JXMenuPeer` musste nicht modifiziert werden, da das Problem nur den Peer des obersten Menüs betrifft. Alle weiteren Untermenüs werden wie gehabt durch die Klasse `JXMenuPeer` erzeugt.

4.6.8 Implementierung der Menüleiste

Die vorliegende Realisierung der Menüleiste besteht aus zwei Klassen: Während die Klasse `JXMenuBarPeer` alle Methoden für das grafische Layout und Zeichnen, sowie die Verwaltung der Leiste enthält, läuft die Interaktion mit derselben und den davon abhängigen Menüs ausschließlich über die Klasse `JXFramePeer` und deren Konnektor. Diese Zweiteilung ist notwendig, da z.B. Popup-Menüs keine zugehörige Menüleiste besitzen, aber genauso angesteuert werden sollen wie gewöhnliche Menüs. In diesem Abschnitt wird nur die Klasse `JXMenuBarPeer` erläutert, die Beschreibung für die Klasse `JXFramePeer` findet sich in Abschnitt 4.7.3.

Um den Peer der Menüleiste zum richtigen Zeitpunkt erstellen zu können, musste eine Modifikation an der AWT-Klasse `Frame` vorgenommen werden: Analog zur Klasse `Component` wurde deren `addNotify()`-Methode so modifiziert, dass sie neben dem Peer der Klasse `Frame` auch gleich den Peer der zugehörigen Menüleiste erzeugt, falls notwendig (vgl. vorherigen Abschnitt).

Die Klasse `JXMenuBarPeer` ist für die korrekte Darstellung der Menüleiste im Mutterfenster verantwortlich (siehe Abbildung 4.36). Intern verwaltet sie dazu, analog zur Klasse `JXMenuPeer`, zwei wichtige Strukturen: Einen Vektor `menus`, der die `Menu`-Objekte selbst beinhaltet, welche in der Leiste dargestellt werden sollen, sowie einen Vektor namens `menuSizes`, der die grafische Größe und Position jedes Eintrags in der Leiste beinhaltet. Diese beiden Vektoren werden von den Methoden der Klasse verwendet, um die Leiste einfacher zeichnen zu können, und um leichter herauszufinden, über welchem Eintrag sich die Maus gerade befindet.

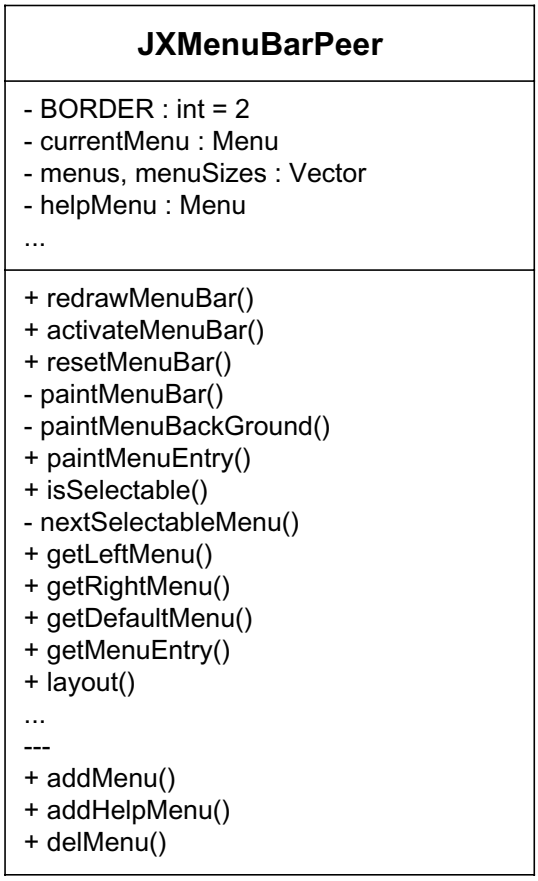


Abb. 4.36 Die Klasse JXMenuBarPeer

Auch von den Methoden her hat die Klasse Einiges mit der Klasse `JXMenuPeer` gemeinsam: So besitzt auch sie Methoden, um den nächsten bzw. vorherigen auswählbaren Eintrag in der Menüleiste zu finden, oder um die Auswahlposition zu ändern. Ebenso kennt sie eine Methode namens `layout()`, die, wie der Name schon sagt, für das Layout der Menüleiste zuständig ist. Die Methode berücksichtigt dabei u.a. die Größe des Mutterfensters und passt die Leiste entsprechend an.

Im Gegensatz zur Klasse `JXMenuPeer` befinden sich hier jedoch keine Handlermethoden. Diese sind hier auch nicht notwendig, da die Klasse `JXFramePeer` und deren Konnektor die notwendige Interaktion mit der Menüleiste steuern.

4.6.9 Ausführung von Menübefehlen: Die Klasse JXMenuThread

Ein Menüsystem dient letztendlich dazu, dem Benutzer eine einfache Möglichkeit in die Hand zu geben, um aus einer Vielzahl zur Verfügung stehender Aktionen die Gewünschte auszuwählen und auszuführen. In diesem Abschnitt soll erläutert werden, wie die Ausführung eines Menüeintrags konkret realisiert wird.

Im Prinzip sollte nach der endgültigen Auswahl eines Menüeintrags Folgendes passieren: Alle offenen Menüfenster sollten geschlossen werden, und eine evtl. benutzte Menüleiste wieder in ihren Grundzustand versetzt werden. Außerdem sollte eine dem Eintrag entsprechende Aktion ausgeführt werden. Da die AWT vom Menüsystem Nachrichten in Form von Ereignissen erwartet, bestünde ein Aktion in diesem Fall aus dem Senden des entsprechenden Ereignisses an die AWT, die dieses an die jeweiligen "Eventlistener" weiterleiten kann.

Dennoch bleibt die Frage, wer diese Arbeiten erledigen sollte. Die Fenster des Windowmanagers von JX sind so implementiert, dass jedes Fenster einen eigenen Thread besitzt, der ständig interne Nachrichten auswertet und die entsprechenden Handler des Fensters aufruft. Theoretisch wäre es also möglich, ein Fenster des Windowmanagers mit den "Aufräumarbeiten" zu betrauen. Das Menüfenster, das den auslösenden Eintrag beinhaltet, wäre eine erste Wahl dafür. Dieses würde jedoch schon am Schließen der Fenster scheitern, da sein Thread der Erste wäre, der dabei beendet würde. Als Zweites käme vielleicht das Mutterfenster des Menüsystems in Frage: Es könnte alle Menüs schließen und seine Menüleiste zurücksetzen, ohne dass dabei Schwierigkeiten entstehen würden. Dazu muss ihn aber erst einmal gesagt werden, dass eine solche Aktion ausgeführt werden soll, d.h. das jeweilige Menüfenster muss ihm erst einmal eine interne Nachricht zukommen lassen, dass etwas passieren soll (was einige Modifikationen im Inneren des Windowmanagers notwendig macht), und außerdem sollte der auslösende Menüeintrag irgendwo gespeichert werden, damit das Mutterfenster auch weiß, was es genau zu tun hat. Das nächste Problem ist, dass die Klasse `MenuHandler` nirgends speichert, welches AWT-Fenster gerade das Menüsystem benutzt, d.h. das Menüfenster weiß gar nicht, an welches Mutterfenster es sich wenden soll.

Diese Probleme können zwar alle mit entsprechendem Aufwand gelöst werden, um die Implementierung jedoch so überschaubar wie möglich zu halten, wurde eine andere Lösung gewählt: Die Aufräumarbeiten werden an einen speziell dafür vorgesehenen Thread übertragen, der dann auch die notwendigen Ereignisse an die Ereigniswarteschlange der AWT schickt. Dieser Thread, der in der Klasse `JXMenuThread` realisiert ist, wird gleich beim Start der JX-Implementierung gestartet und blockiert sich anschließend, bis eine Menüaktion ausgeführt werden soll. Diese wird vom Thread ausgeführt, und dieser legt sich danach gleich wieder schlafen, solange bis die nächste Aktion bearbeitet werden soll. Abbildung 4.37 zeigt die Klasse `JXMenuThread` im Überblick.

Dieser Thread ist sinnvollerweise eines der globalen Objekte der Implementierung, von denen es nur eine globale Instanz gibt. Auf diese Weise ist er einfach und eindeutig durch die `JXToolkit`-Schnittstelle ansprechbar.

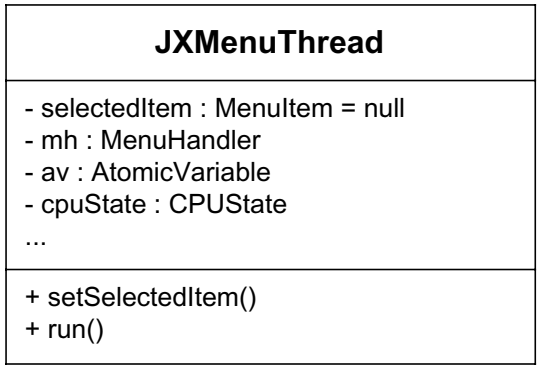


Abb. 4.37 Die Klasse JXMenuThread

Intern verwendet der Thread ein JX-spezifisches Konstrukt, ein Objekt der Klasse `AtomicVariable`. Unter JX gab es zum Zeitpunkt dieser Studienarbeit noch keine Unterstützung für die Methoden `wait()` und `notify()` der Klasse `Object`, die für das Schlafenlegen und Aufwecken eines Threads verantwortlich sind. Die Klasse `AtomicVariable` von JX löst dieses Problem auf eigene Weise: Die Methode `set()` setzt eine interne Referenz auf das als Parameter übergebene Objekt. Mit `blockIfEqual()` wird der aufrufende Thread schlafengelegt, falls die interne Referenz auf das übergebene Objekt zeigt. Die Methode `atomicUpdateUnblock()` schließlich setzt die interne Referenz der Klasse neu und weckt dabei gleichzeitig einen schlafenden Thread wieder auf. Damit der aufzuweckende Thread identifiziert werden kann, wird der Methode noch ein JX-eigenes `CPUState`-Objekt übergeben, welches auf den schlafenden Thread zeigt.

Ablauf des Threads

Der Thread wird gleich zu Beginn geladen und gestartet. Dadurch springt er in seine `run()`-Methode, und blockiert sich gleich zu Anfang seiner internen Endlosschleife an der Methode `blockIfEqual()`.

Wird nun ein aktionsauslösender Eintrag eines Menüs ausgewählt, so ruft dieses seine eigene Methode `performMenuItemAction()` mit dem auslösenden `MenuItem`-Objekt als Parameter auf. Diese wiederum ruft die Methode `setSelectedItem()` der Klasse `JXMenuThread` auf, welche das übergebene `MenuItem`-Objekt intern registriert und den Thread mittels `atomicUpdateUnblock()` aufweckt. Dieser führt nun folgende Schritte durch:

– Das `AtomicVariable`-Objekt wird zurückgesetzt, damit der Thread beim nächsten Schleifendurchlauf wieder blockiert.

– Die globale Instanz der Klasse `MenuHandler` wird angewiesen, den kompletten sichtbaren Menüzweig zu entfernen.

– Falls es sich bei dem gerade geschlossenen Menü um kein Popup-Menü handelte, so wird die zugehörige Menüleiste wieder zurückgesetzt. Eine Menüleiste kann sich bei der Klasse `MenuHandler` registrieren, und diese Registrierung wird hier verwendet, um die Menüleiste ansprechen zu können.

– Dann wird das dem Eintrag entsprechende Ereignis verschickt: Ein registriertes `CheckBoxMenuItem`-Objekt hat ein "Item-Event" zur Folge, während ein `MenuItem`-Objekt zu einem "Action-Event" führt.

– Zuletzt blockiert sich der Thread wieder und wartet auf die nächste auszuführende Aktion.

4.7 Peers für das Fenstersystem

Die letzten Abschnitte haben sich mit den Peers der beiden großen Gruppen befasst, d.h. den "normalen" sowie den Menü-Peers. Obwohl diese Gruppen sehr verschieden voneinander sind, haben sie doch die Gemeinsamkeit, dass sie alle in irgendeiner Weise an ein Fenster gebunden sind: Normale Peers werden in einem Fenster dargestellt, und auch Menü-Peers sind durch die Menüleiste oder die AWT-Komponenten von einem Fenster abhängig.

Die in diesem Kapitel zu besprechenden Peers `JXWindowPeer` und `JXFramePeer` gehören zwar von der Einteilung durch die AWT her den "normalen" Peers an, nehmen aber trotzdem eine Sonderstellung ein, da sie im Gegensatz zu den anderen Komponenten kein Teil eines Fensters sind, sondern das Fenster selbst als Komponente zur Verfügung stellen.

Aus der Sicht der AWT stellt die abstrakte Klasse `Window` ein Grundgerüst zur Verfügung, von dem sich Klassen wie `Frame` oder `Dialog` ableiten lassen, welche dann die wirklichen Fenster auf dem Bildschirm darstellen. Auf Peer-Ebene wurde das Zusammenspiel der einzelnen Klassen analog realisiert: Die Klasse `JXWindowPeer` stellt alle Methoden zur Verfügung, die für ein Fenster benötigt werden, ist aber selbst eine abstrakte Klasse. Um dennoch ein Fenster zu erzeugen, gibt es die Klasse `JXFramePeer`. Dieser Peer mit der zugehörigen Komponente `Frame` stellt unter JX ein Fenster dar, enthält selbst aber so gut wie keine zusätzlichen Methoden für die Fenstersteuerung.

Wie jeder andere Peer, der ein Fenster verwendet, kennt auch die Klasse `JXWindowPeer` einen zugehörigen Konnektor, welcher in diesem Fall in der Klasse `JXWindowConnector` realisiert ist. Dieser Konnektor stellt die Schaltzentrale eines Fensters der JX-Implementierung dar, denn ihm fallen Aufgaben zu wie z.B. Verwaltung von Komponenten- und Menübereich, Weiterleitung von Ereignissen an die jeweiligen Komponenten, und andere Dinge, die für das Funktionieren der Kommunikation zwischen Benutzer und Komponenten unverzichtbar sind.

4.7.1 Die Klasse JXWindowPeer

Dieser Peer repräsentiert ein allgemeines Fenster, wie es von abgeleiteten Klassen benutzt wird. Die Klasse definiert einige allgemeine Methoden für die Fenstersteuerung, und besitzt daneben eine Referenz auf ihren zugehörigen Konnektor, der auch von ihren abgeleiteten Klassen be-

nutzt werden kann. Alle Methoden stellen allgemeine Hilfsmethoden dar, welche benutzt werden können, um einige Aspekte des Fensters, wie Größe oder Position, steuern zu können. Abbildung 4.38 zeigt die Klasse im Überblick.

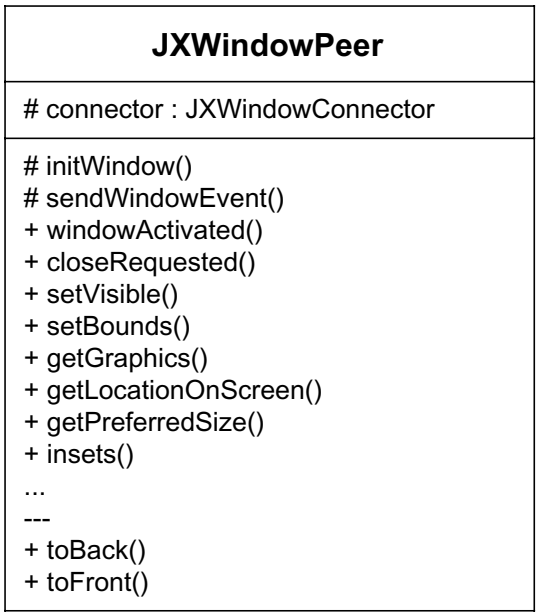


Abb. 4.38 Die Klasse JXWindowPeer

Eine Methode, die etwas unter den anderen hervorragt, ist die Methode `setVisible()`. Diese wird, wie bei anderen Komponentenpeers auch, dazu verwendet, um die entsprechende Komponente zu zeigen oder zu verstecken. Im Gegensatz zu den anderen Peers, bei denen diese Methode so implementiert ist, dass einfach ein Flag gesetzt und der Peer entsprechend neu gezeichnet wird, fällt die Implementierung hier etwas anders aus: Versucht man hier, ein Fenster sichtbar zu machen, wird im Normalfall ein neuer Konnektor erzeugt, initialisiert und auf dem Bildschirm dargestellt. Soll das Fenster dagegen entfernt werden, so wird der Konnektor beendet und seine Referenz gelöscht. Somit existiert der zugehörige Konnektor im Normalfall[3] nur solange, wie er auf dem Bildschirm zu sehen ist.

Diese Realisierung, bei der eine Vielzahl von Konnektor-Objekten erzeugt und verworfen wird, wenn man das Fenster öfter anzeigen und verstecken will, ist zwar umständlich und mit viel Overhead verbunden, aber nicht zu vermeiden. Dies liegt daran, dass der Windowmanager keine Möglichkeit kennt, ein Fenster vorübergehend zu verstecken. Deswegen ist es notwendig, das Verstecken eines Fensters durch diese Implementierung zu “simulieren”.

3. “im Normalfall” heißt, dass es auch einen Fall gibt, in dem der Konnektor schon vor seiner Darstellung existiert: Wird der Peer selbst erzeugt, so erzeugt er sich gleich auch einen Konnektor. Diese Implementierung wurde gewählt, um der Forderung der AWT nachzukommen, dass ein Fenster, wenn es zum ersten Mal sichtbar wird, ein entsprechendes Ereignis aussenden soll.

4.7.2 Die Klasse JXWindowConnector

Dieser Konnektor ist das Kernstück der JX-Implementierung: Er kontrolliert die Umwandlung von einfachen Maus- und Tastaturereignissen, welche in seinem Fenster auftreten, in die entsprechenden Methodenaufrufe der zugehörigen Peers, damit diese auf die Ereignisse reagieren können. Abbildung 4.39 zeigt den Konnektor im Überblick.

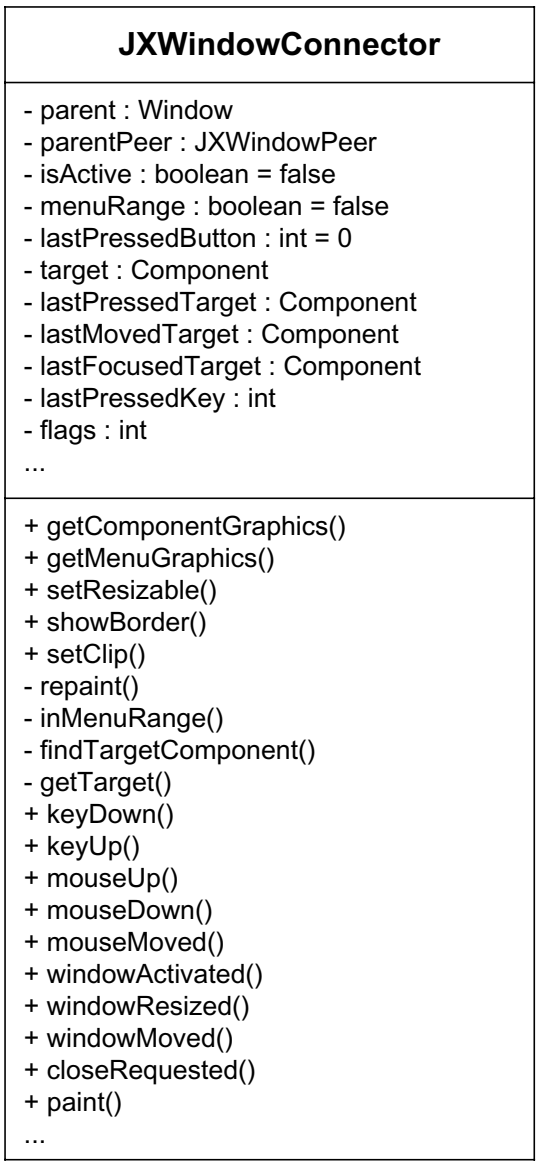


Abb. 4.39 Die Klasse JXWindowConnector

Die Klasse ist grob in zwei Teile getrennt: Der eine Teil besteht aus Hilfsmethoden, sowohl für den zugehörigen Peer, als auch für die Handlermethoden des Fensters gedacht. Den zweiten Teil stellen eben diese Handlermethoden dar, welche vom Windowmanager aufgerufen werden, und in der Klasse entsprechend implementiert wurden.

Die Hilfsmethoden bewerkstelligen die unterschiedlichsten kleineren Aufgaben: Sie können einstellen, ob das Fenster in seiner Größe veränderbar ist, oder ob es einen Rahmen besitzen soll; Sie erlauben es, die Komponente des Fensters von einer Größenänderung desselben durch den Windowmanager informieren, einen Clipping-Bereich im Fenster setzen, oder ein `JXGraphics`-Objekt zurückgeben, welches das Zeichnen in das Fenster erlaubt.

Unter anderem besitzt der Konnektor auch eine private Methode namens `doRepaint()`. Diese wird aufgerufen, wenn es darum geht, das Fenster komplett neu zu zeichnen. Wenn das übergebene Flag es anzeigt, werden zuerst die Layoutmanager des Fensters und seiner eingebetteten Komponenten angewiesen, das Layout neu zu bestimmen. Anschließend wird das Neuzeichnen gestartet, indem die `redraw()`-Methode des Fenster-Peers aufgerufen wird. Wie bereits in Abschnitt 4.2.2 erläutert, führt dies durch einen rekursiven Algorithmus zum Neuzeichnen des kompletten Fensters.

4.7.2.1 Finden von eingebetteten Komponenten

Eine Aufgabe, die sich bei der Implementierung ergibt, ist die Suche nach der Komponente, für die das jeweilige Ereignis des Windowmanagers bestimmt ist. Bei Tastaturereignissen ist dies einfach, denn diese werden an diejenige Komponente geschickt, welche gerade den Tastaturfokus besitzt. Welche Komponente das ist, kann durch Anfrage bei der globalen Instanz der Klasse `FocusHandler` herausgefunden werden. Bei Mausereignissen kennt man dagegen nur die Koordinaten des Punktes im Fenster, bei dem das Ereignis aufgetreten ist.

Die AWT-Klasse `Container` kennt die Methode `findComponentAt()`, die es erlaubt, einen Kontainer rekursiv zu durchsuchen, und die Komponente zu finden, welche sich direkt unter den angegebenen Koordinaten befindet. Die Methode hat jedoch den Nachteil, dass sie dabei einige Annahmen über Kontainer macht, welche bei der Realisierung der Klasse `JXScrollPanePeer` nicht zutreffen (vgl. Abschnitt 4.5.7).

Die Problematik soll anhand der Abbildung 4.40 kurz verdeutlicht werden: Eine Komponente des Typs `ScrollPane` kann eine eingebettete Komponente besitzen, deren Dimensionen größer sein können als der sichtbare Bereich des `ScrollPane`-Kontainers. In diesem Fall besitzt der Kontainer normalerweise ein oder zwei Scrollbalken, mit denen durch die Komponente gescrollt werden kann. Nun kann es vorkommen, dass sich dabei der Zeichenbereich der eingebetteten Komponente mit dem Rahmenbereich des `ScrollPane`-Kontainers überlappt (vgl. Skizze). Diese Überlappung der Bereiche ist durchaus beabsichtigt, schließlich soll sich die Komponente nicht wegen ihrer Einbettung in einen `ScrollPane`-Kontainer in der Größe ändern, sondern ihre Originalgröße beibehalten. Außerdem ist die Überlappung nicht sichtbar, da die Ausgabe der eingebetteten Komponente auf den sichtbaren Bereich des `ScrollPane`-Kontainers beschränkt ist.

Probleme mit dieser Überlappung treten erst dann auf, wenn die Methode `findComponentAt()` arbeitet: Diese verarbeitet den eingestellten Rahmenbereich eines Kontainers nicht in der gewünschten Art und Weise, sondern gibt einer (darunterliegenden!) eingebetteten Kompo-

nente den Vorzug, d.h. liefert diese als Ergebnis zurück (vgl. Skizze). Bei anderen Kontainern tritt dieses Problem nicht auf, da deren eingebettete Komponenten nie im Rahmenbereich liegen.

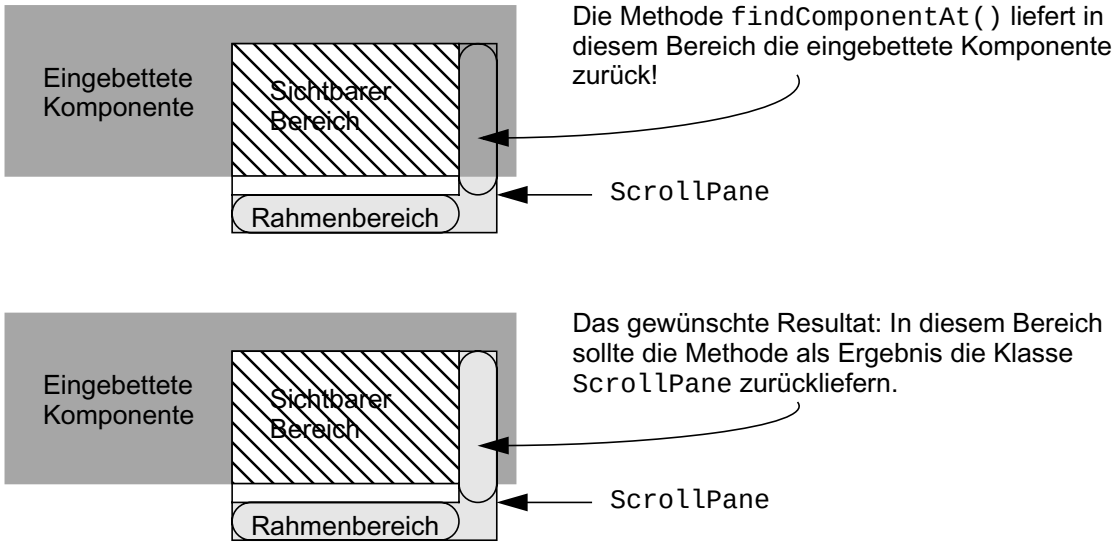


Abb. 4.40 Die Problematik der Methode findComponentAt()

Aus diesem Grund wurden die Methoden `findTargetComponent()` und `getTarget()` in die Klasse `JXWindowConnector` implementiert. Die Methode `findTargetComponent()` realisiert einen angepassten Algorithmus der Methode `findComponentAt()`, während die Methode `getTarget()` diese kapselt, um bestimmte Situationen besser zu handhaben (siehe weiter unten). Der Algorithmus der Methode `findTargetComponent()` läuft dabei wie folgt ab:

- Zuerst wird getestet, ob die übergebenen Koordinaten überhaupt im Bereich des übergebenen Kontainers liegen. Falls nicht, ist keine weitere Bearbeitung nötig, und die Methode bricht ab.
- Nun wird jede eingebettete Komponente des Kontainers untersucht:
  - Ist sie unsichtbar, wird sie ignoriert.
  - Die Koordinaten werden an die Komponente angepasst, da jede Komponente ihre Position relativ zum Mutterkontainer speichert.
  - Ist die Komponente wieder ein Kontainer, so wird dessen Rahmen überprüft: Liegen die Koordinaten genau im Rahmenbereich des Kontainers, so wird dieser Kontainer als Ziel zurückgegeben. Liegen die Koordinaten dagegen im Komponentenbereich des Kontainers, so wird für diesen die Methode `findTargetComponent()` ausgeführt.

  - Ist die Komponente kein Kontainer, so wird getestet, ob die Koordinaten in ihrem Bereich liegen. Ist das der Fall, so wird diese Komponente als Ziel zurückgegeben.
- Wurde keine eingebettete Komponente gefunden, welche die Koordinaten beinhaltet, so wird der übergebene Kontainer als Ziel zurückgeliefert.

Durch diesen Algorithmus wird die Komponente zu den übergebenen Koordinaten immer gefunden, und auch der Peer der Klasse `ScrollPane` funktioniert so korrekt.

Die Methode `getTarget()` überprüft den Rückgabewert der Methode `findTargetComponent()` und korrigiert ihn gegebenfalls: Sollte diese eine Null-Referenz zurückliefern, so wird stattdessen eine Referenz auf die `Frame`-Komponente des Mutterfensters zurückgeliefert. Diese Korrektur ist notwendig, wenn die angegebenen Maus-Koordinaten z.B. im Rahmen des Fensters liegen.

4.7.2.2 Implementierung der Handlermethoden

Wie jeder Konnektor, besitzt auch die Klasse `JXWindowConnector` die Handlermethoden, welche vom Windowmanager aufgerufen werden, falls bestimmte Ereignisse eintreten. Diese Methoden sind in dieser Klasse sehr umfangreich implementiert, da sie die eingangs erwähnten Aufgaben zu erfüllen haben.

Anmerkung: Die folgende Liste enthält nur Angaben über die Implementierung derjenigen Teile des Codes, welche für das grundsätzliche Verwalten von eingebetteten Komponenten innerhalb des Fensters notwendig sind. Wie im nächsten Abschnitt noch gezeigt wird, ist diese Implementierung damit noch nicht vollständig, da die Verwaltungsstrukturen und -algorithmen für das Menüsystem eines Fensters fehlen. Diese werden in Abschnitt 4.7.3.2 beschrieben.

Die Implementierung der einzelnen Methoden sieht nun im Wesentlichen wie folgt aus:

- Die Methode `keyDown()` ermittelt zuerst diejenige Komponente, welche gerade den Tastaturfokus besitzt. Dann wird der übergebene Tastencode ausgewertet: Handelt es sich um die <**Tab**>-Taste, so werden zuerst alle Hilfsfenster wie die der Klasse `Choice` geschlossen. Anschließend wird die Methode `transferFocus()` der fokussierten Komponente aufgerufen, damit diese den Tastaturfokus weiterleitet. Wurde dagegen eine andere Taste gedrückt, so wird diese an den Peer der fokussierten Komponente weitergeleitet, indem dessen Methode `keyPressed()` aufgerufen wird.
- Die Methode `keyUp()` arbeitet analog: Es wird zuerst die aktuell fokussierte Komponente ermittelt. Falls der Tastencode dem entspricht, der zuvor von der Methode `keyDown()` verarbeitet wurde, wird die Methode `keyClicked()` des Peers der Komponente aufgerufen. Um den Peer auf jeden Fall davon zu informieren, dass eine Taste losgelassen wurde, wird abschließend dessen Methode `keyReleased()` aufgerufen.
- Die Methode `mouseDown()` überprüft zuerst, ob die gedrückte Maustaste ist Einzige ist, oder ob schon eine andere ebenfalls gedrückt ist. In letzterem Fall ist zuerst die Methode `mouseUp()` aufgerufen, damit dieses Ereignis richtig verarbeitet werden kann. Anschließend wird mittels `getTarget()` die Ziel-Komponente bestimmt, die unter den

angegebenen Koordinaten liegt. Diese wird mittels `requestFocus()` angewiesen, den Tastaturfokus für sich zu beanspruchen, anschließend wird die Methode `mousePressed()` des zugehörigen Peers aufgerufen.

- Auch die Methode `mouseUp()` ermittelt zuerst die Ziel-Komponente zu den übergebenen Koordinaten. Entspricht diese Komponente derjenigen, die schon beim letzten Aufruf der Methode `mouseDown()` ermittelt wurde, so wird an dessen Peer die Methode `mouseClicked()` aufgerufen. Abschließend wird am Peer denjenigen Komponente, welche beim letzten Aufruf von `mouseDown()` ermittelt wurde, die Methode `mouseReleased()` aufgerufen. Diese spezielle Implementierung unterstützt die Verwendung von Drag&Drop: Ein Peer wird gedrückt gezeichnet, sobald eine Maustaste darüber gedrückt wird, und erst wieder normal dargestellt, wenn die entsprechende Taste wieder losgelassen wird, ganz egal, wo die Maus sich dann befindet.
- Die Methode `mouseMoved()` erledigt die meisten anfallenden Aufgaben bei Mausereignissen. Die Ermittlung der Ziel-Komponente geschieht hier ein wenig anders: Ist eine Maustaste gedrückt, so wird als Ziel-Komponente diejenige hergenommen, über welcher diese Taste gedrückt wurde. Ansonsten wird die Zielkomponente ganz normal durch `getTarget()` bestimmt. Auch diese Vorgehensweise unterstützt das Drag&Drop-Prinzip, und ist besonders nützlich bei Verwendung von Komponenten der Klasse `Scrollbar`.

  Wurde die Ziel-Komponente ermittelt, finden eine ganze Reihe von Überprüfungen statt: Unterscheidet sich die neue Komponente von der zuletzt durch `mouseMoved()` Ermittelten, so wird dies als das Betreten einer neuen Komponente durch die Maus betrachtet, und die Methoden `mouseExited()` und `mouseEntered()` der jeweils zugehörigen Peers werden aufgerufen. Hat sich die Komponente dagegen nicht geändert, so wird einfach die `mouseMoved()`-Methode des zugehörigen Peers aufgerufen.

  Handelt es sich bei der ermittelten Komponente um die Komponente `Frame` des Mutterfensters, so sind einige spezielle Überprüfungen notwendig, da sich die Maus dann im Rahmen des Fensters befindet: Abhängig vom Parameter des Windowmanagers, der angibt, wo sich die Maus momentan im Rahmen bewegt, werden die zugehörigen Methoden des `Frame`-Peers aufgerufen.

  Ist bei Aufruf der Methode `mouseMoved()` eine Maustaste gedrückt (wurde also die Maus mit gedrückter Taste bewegt), so wird die `mouseDragged()`-Methode des aktuellen Ziel-Peers aufgerufen.
- Die Methode `windowActivated()` wird immer dann aufgerufen, wenn das Fenster aktiviert bzw. deaktiviert wird, oder wenn mit der Maus auf einen Bereich des Fensters geklickt wird. Sie wurde so implementiert, dass zuerst alle Hilfsfenster, wie die der Klasse `Choice`, entfernt werden. Falls das Fenster bereits aktiviert ist und nochmals aktiviert werden soll, beendet sich die Methode. Ansonsten wird unterschieden, welchen neuen Status das Fenster bekommt: Soll es deaktiviert werden, so speichert der Konnektor die aktuell fokussierte Komponente seines Fensters in einer eigenen internen Referenz. Wird

es dagegen wieder aktiviert, so fordert die gespeicherte Komponente oder, falls nicht vorhanden, das `Frame`-Objekt des Fensters den Fokus an. Auf diese Weise zeigt ein aktiviertes Fenster immer seine aktuell fokussierte Komponente an. Abschließend wird noch die Methode `windowActivated()` des `Window`-Peers aufgerufen, damit dieser auf das Ereignis reagieren kann.

- Die Methoden `windowResized()` und `windowMoved()` sind beide so realisiert, dass sie auf die Methode `doWindowChange()` des Konnektor zurückgreifen: Diese aktualisiert die internen Daten der zugehörigen `Window`-Komponente, was die aktuelle Größe und Position des Fensters betrifft, und zeichnet im Falle von `windowResized()` außerdem das Fenster mittels `doRepaint()` neu.
- Die Methode `closeRequested()` wird aufgerufen, sobald der "Close-Button" des Fensters gedrückt wurde. In der vorliegenden Implementierung ruft er einfach die Methode `closeRequested()` des `Window`-Peers auf, so dass dieser darauf reagieren kann.
- Die `paint()`-Methode, die immer dann aufgerufen wird, wenn das Fenster neu gezeichnet werden soll, ruft ihrerseits einfach die interne `doRepaint()`-Methode des Konnektors auf.

4.7.3 Die Klasse JXFramePeer

Die Klasse `JXFramePeer` stellt den Peer für die `Frame`-Komponente dar. Sie ergänzt die Klasse `JXWindowPeer` um die Fähigkeit zur Menüverwaltung, und führt somit die beiden grossen Gruppen der "normalen" und Menü-Komponenten zusammen. Da unter der AWT nur die Klasse `Frame` ein eigenes Menü besitzen kann, wird die Verwaltung dafür auch erst im zugehörigen Peer realisiert. Abbildung 4.41 zeigt den Peer im Überblick.

Die Klasse erweitert die Methode `redrawComponent()` um die Möglichkeit, eine Menüleiste in das Fenster zu zeichnen, falls notwendig. Außerdem stellt sie der Klasse `JXMenuBarPeer` die Methode `getMenuGraphics()` zur Verfügung, damit diese die Menüleiste in das Fenster zeichnen kann. Die Klasse implementiert daneben eine Reihe von Methoden, welche es dem Konnektor erlauben, direkt mit der Menüleiste zu kommunizieren (siehe nächsten Abschnitt), und einfach an das zugehörige `JXMenuBarPeer`-Objekt weitergeleitet werden.

Der Peer besitzt zum Öffnen bzw. Umschalten von Menüs zwei Hilfsmethoden namens `openMenu()` und `switchToMenu()`. Die Methode `openMenu()` ist für das Öffnen eines Menüs zuständig: Sie stellt zuerst den Auswahlbalken der Menüleiste auf das neue Menü ein, und zeichnet diese neu. Anschließend wird die Position des zugehörigen Eintrags der Menüleiste auf dem Bildschirm ermittelt, und dann mit diesen Informationen die Methode `openMenu()` der Klasse `MenuHandler` aufgerufen, die sich um alles Weitere kümmert.

Die Methode `switchToMenu()` ist für das "Umschalten" von einem Eintrag der Menüleiste zum anderen verantwortlich. Hat der vorhergehende Eintrag sein zugehöriges Menü geöffnet, so wird dieses geschlossen, und das Menü des neuen Eintrags mittels `openMenu()` geöffnet. Ansonsten wird nur der Auswahlbalken der Menüleiste auf den neuen Eintrag gesetzt.

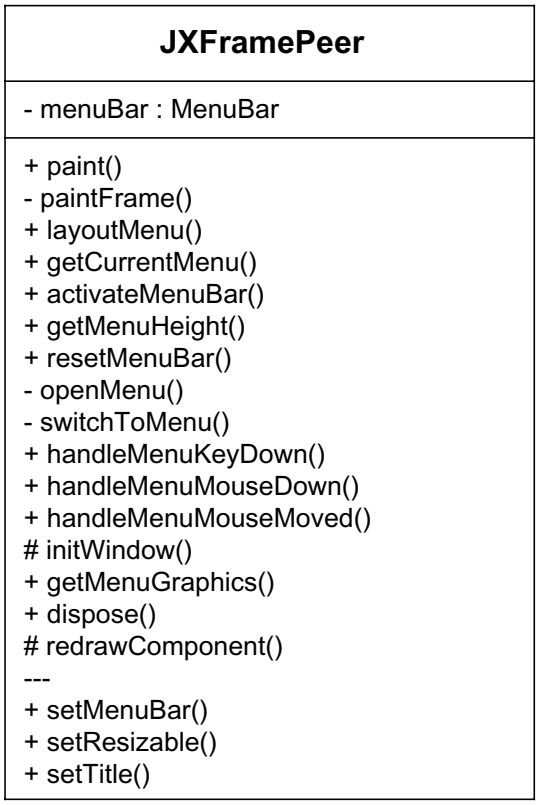


Abb. 4.41 Die Klasse JXFramePeer

Als wichtigste Bestandteile besitzt der Peer eigene Handler für Maus- und Tastaturereignisse, welche die Menüleiste betreffen. Die entsprechenden Methoden sind dabei wie folgt implementiert:

- Die Methode `handleMenuMouseDown()` wird aufgerufen, wenn eine Maustaste im Bereich der Menüleiste gedrückt wird. Sie ermittelt zuerst das Menü, dessen Name ausgewählt wurde, und schließt ein vorhandenes offenes Menü. Wurde der Name eines Menüs angeklickt, so wird dieses nun geöffnet, und die Menüleiste aktualisiert.
- Die Methode `handleMenuMouseMoved()` wird aufgerufen, wenn sich die Maus im Bereich der Menüleiste bewegt. Sie ist dafür zuständig, die Menüleiste und das gerade offene Menü immer auf dem neuesten Stand zu halten. Wird die Maus auf einen neuen Eintrag der Menüleiste bewegt, so wird die Menüleiste durch Verwendung von `switchToMenu()` aktualisiert. Sollte ein Menü geöffnet gewesen sein, so wird dieses dabei geschlossen und dasjenige des neuen Eintrags in der Leiste geöffnet.
- Die Methode `handleMenuKeyDown()` ist die Umfangreichste der Klasse. Sie behandelt alle Tastendrücke, die auftreten, während die Menüleiste oder ein Menü aktiv sind. Im Gegensatz zu den beiden vorhergehenden Methoden wird diese auch aufgerufen, wenn ein Popup-Menü offen ist, also die Menüleiste selbst nicht davon betroffen ist. Ihre Implementierung wird gleich noch genauer besprochen werden.

Die restlichen Handler, welche in der Klasse `JXWindowPeer` definiert sind, werden in der vorliegenden Implementierung nicht verwendet.

4.7.3.1 Verarbeitung von Tastaturereignissen

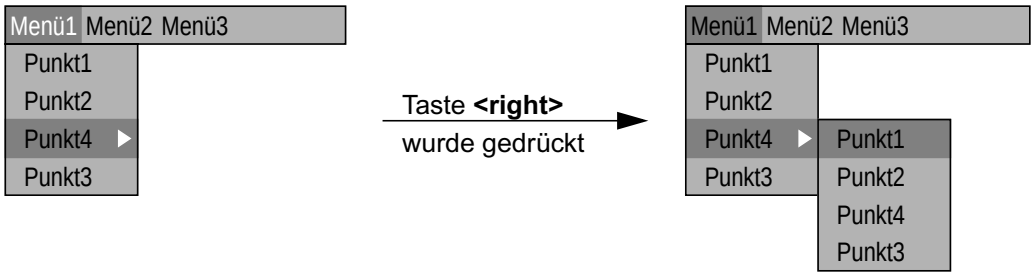
Die Methode `handleMenuKeyDown()` regelt alle Tastaturereignisse für die Menüleiste und evtl. offene Menüs. Dabei wertet sie den Tastencode aus, und führt die entsprechenden Aktionen durch:

- Die Tasten <**Return**>, <**up**> oder <**down**> werden gleich verarbeitet: Sollte ein Menü offen sein, so werden sie einfach dahin weitergeleitet, indem die Methode `performKey()` der Klasse `MenuHandler` aufgerufen wird (vgl. Abschnitt 4.6.4.1). Ist dagegen kein Menü offen, so wird das Menü geöffnet, dessen Eintrag gerade unter dem Auswahlbalken der Menüleiste steht. Wurde die Taste <**up**> gedrückt, so wird der Auswahlbalken des neuen Menüs auf den untersten auswählbaren Eintrag gesetzt, ansonsten auf den obersten auswählbaren Eintrag.
- Auch die Taste <**Esc**> wird an die Klasse `MenuHandler` weitergeleitet, falls ein Menü geöffnet sein sollte. Anderenfalls wird die Menüleiste zurückgesetzt und damit verlassen.
- Die Behandlung der Cursortasten <**left**> und <**right**> verläuft weitestgehend identisch: Zuerst wird der linke bzw. rechte Nachbar des momentan ausgewählten Eintrags in der Menüleiste gesucht. Ist kein Menü offen, so wird der Auswahlbalken der Menüleiste einfach auf den neuen Eintrag gesetzt, und diese neu gezeichnet. Ist dagegen ein Menü offen, so wird ein wenig anders verfahren: Zuerst wird die Taste an die Klasse `MenuHandler` mittels `performKey()` weitergeleitet. Sollte dieser Aufruf der Methode den Wert `false` zurückliefern (d.h. die Taste nicht verarbeitet worden sein, siehe nächsten Absatz), und es sich bei dem offenen Menü um kein Popup-Menü handeln, dann wird das linke bzw. rechte Nachbarmenü mittels `switchToMenu()` aktiviert.

Die Methoden `performKey()` der Klassen `MenuHandler` und `JXMenuPeer` liefern einen boolschen Wert zurück, der angibt, ob der übergebene Tastencode verarbeitet wurde oder nicht. Wie gerade gesagt, ist diese Rückmeldung wichtig für die richtige Verarbeitung der Cursortasten <**left**> und <**right**>: Um mittels dieser Cursortasten ein benachbartes Menü zu öffnen, muss erst sichergestellt werden, dass diese Taste nicht irgendwo im bestehenden Menüzweig verarbeitet wird. Nur, wenn dies nicht der Fall ist, kann die Klasse `JXFramePeer` das Nachbarmenü öffnen.

Abbildung 4.42 verdeutlicht diesen Zusammenhang anhand eines konkreten Beispiels: Befindet sich der Menücursor auf einem Untermenü-Eintrag, und die Taste <**right**> wird gedrückt, so wird die Methode `performKey()` des Menüs den Wert `true` zurückliefern, da es die Taste auswerten konnte, indem es das zugehörige Untermenü geöffnet hat. In diesem Fall unternimmt die Klasse `JXFramePeer` nichts weiter. Befindet sich der Cursor dagegen auf einem “normalen” Eintrag, und die Taste wird gedrückt, so verarbeitet die Methode `performKey()` das Ereignis nicht, und liefert `false` zurück. In diesem Fall kann die Klasse `JXFramePeer` das rechte Nachbarmenü öffnen.

Situation1: Aufruf von `performKey()` liefert `true` zurück



Situation2: Aufruf von `performKey()` liefert `false` zurück

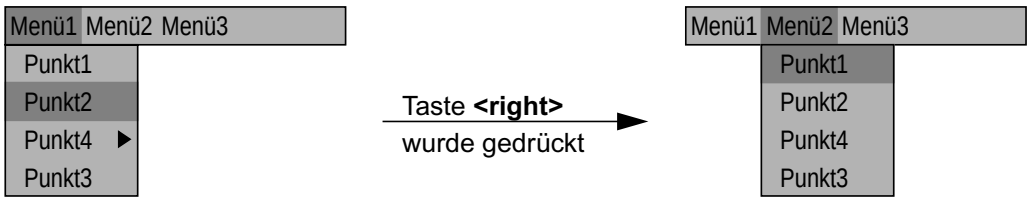


Abb. 4.42 Bedeutung des Rückgabewertes der Methoden performKey()

4.7.3.2 Erweiterungen des zugehörigen Konnektors

Für die Menüverwaltung sind eine Reihe von Erweiterungen und Ergänzungen an der Klasse `JXWindowConnector` notwendig, wie sie in Abschnitt 4.7.2 beschrieben wurde: Da unter der AWT, sowie in der vorliegenden Implementierung das Menüsystem und die normalen Komponenten voneinander getrennt behandelt werden, ist schon auf Konnektor-Ebene eine Unterscheidung derselben notwendig. Deswegen wurden hier folgende Variablen und Methoden modifiziert oder neu hinzugefügt:

- Die Klasse kennt durch ihre Mutterklasse `GeneralConnector` zwei Variablen `componentOffsetX` und `componentOffsetY`. Diese Variablen geben die linke obere Ecke des Bereiches an, in dem die normalen AWT-Komponenten sich befinden dürfen. Die restliche Fläche (d.h. falls diese Variablen größer als null sind), die sich oberhalb bzw. links des Komponentenoffsets befindet, kann für das Menüsystem verwendet werden.
- Die Klasse kennt nun zwei Methoden zur Rückgabe von `JXGraphics`-Objekten: Die Methode `getComponentGraphics()` liefert ein Objekt zurück, das bereits an den Komponentenoffset angepasst wurde, so dass die AWT-Komponenten immer relativ zum Offset gezeichnet werden. Die Methode `getMenuGraphics()` liefert ein unmodifiziertes Objekt zurück, welches zum Zeichnen der Menüleiste verwendet wird.
- Die Methode `inMenuRange()` wurde eingeführt, welche angibt, ob sich die übergebenen Koordinaten im Bereich der Menüleiste befinden oder nicht.

- Die private Methode `doRepaint()` wurde um das Layouten der Menüleiste erweitert.
- Die Methoden zur Behandlung von Mausereignissen wurden so erweitert, dass jede Methode zu Beginn erst einmal überprüft, ob sich die Maus im Bereich der Menüleiste befindet. Ist das der Fall, so wird die entsprechende Handlermethode der Klasse `JXFramePeer` aufgerufen (vgl. vorherigen Abschnitt), andernfalls wird als Nächstes getestet, ob schon ein Menü geöffnet ist. Falls ja, so wird der Handler beendet, damit das Mutterfenster nicht auf ein Ereignis reagiert, welches für das offene Menü bestimmt ist. Ist kein Menü offen, so werden die Koordinaten des Mausereignisses um den Komponentenoffset korrigiert, und dann wie zuvor vom Konnektor verarbeitet. Durch die Korrektur der Koordinaten ist für eine Komponente nicht erkennbar, ob ihr Mutterfenster ein Menüsystem besitzt oder nicht.
- Die Methoden zur Behandlung von Tastaturereignissen wurden ähnlich erweitert: Ist bei Aufruf eines Handler ein Menü offen, so wird das Tastaturereignis an die Klasse `JXFramePeer` weitergeleitet, ansonsten wird das Ereignis wie zuvor an diejenige Komponente weitergeleitet, welche den Tastaturfokus besitzt. Außerdem wurde eine “Menütaste” realisiert: Durch Drücken von **<Alt>** kann der Benutzer den Fokus zwischen Menü- und Komponentenbereich eines Fensters hin- und herschalten.
- Die Handlermethode `windowActivated()` wurde dahingehend erweitert, dass zu Beginn ein aktuell sichtbarer Menüzweig entfernt wird. Auf diese Weise wird ein Menü automatisch entfernt, sobald das Mutterfenster inaktiv, oder mit der Maus auf einen neuen Bereich des Mutterfensters geklickt wird.

4.8 Sonstige peer-spezifische Klassen

Der folgende Abschnitt enthält eine Beschreibung der Klassen, die keine weiteren Peers oder Konnektoren realisieren, sondern andere Zwecke in der JX-Implementierung der AWT erfüllen.

4.8.1 Die Klasse KeyMap

Der Windowmanager verwendet für die Weitergabe und interne Verarbeitung von Tastaturereignissen seine eigene Klasse `Keycode`, welche alle wichtigen Daten über das Ereignis speichert. Die AWT hingegen besitzt die Klasse `KeyEvent`, um Tastaturereignisse zu verarbeiten. Beide Systeme bieten in ihren Klassen u.a. eine ganze Reihe von Konstanten an, welche die jeweilige Taste identifizieren. Um diese Konstanten ineinander zu konvertieren, wurde die Klasse `KeyMap` geschaffen.

Die Klasse besitzt die statische Methode `translate()`, welche die Umwandlungsarbeit leistet. Dabei bekommt diese als Parameter den Tastencode der Klasse `Keycode` übergeben und liefert als Ergebnis den entsprechenden Code von `KeyEvent` zurück. Daneben sind noch einige Methoden vorhanden, welche erkennen können, ob die Modifikationstasten (Ctrl, Alt, Shift) zum Zeitpunkt des aktuellen Tastaturereignisses gedrückt waren oder nicht.

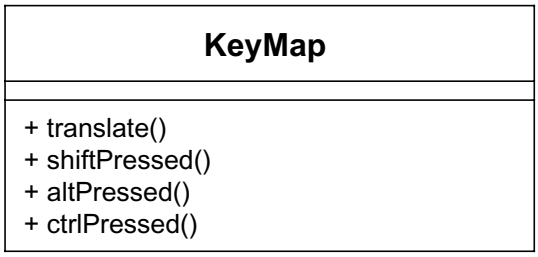


Abb. 4.43 Die Klasse KeyMap

4.8.2 Die Klasse JXColors

Diese Klasse stellt ein Farbschema für die Peers zur Verfügung, welches diese zum Selbstzeichnen verwenden. Sie besteht aus zwei Teilen: Einer Reihe von statischen `Color`-Objekten, sowie der Methode `loadColorInformation()`.

Die `Color`-Objekte beinhalten alle Farben, die für das Zeichnen der Peers notwendig sind. Jeder Peer greift beim Zeichnen, genauer gesagt beim Setzen der jeweiligen Farben, auf diese statischen Objekte zu. Somit stellen diese Objekte auch eine zentrale Verwaltung des AWT-Farbschemas unter JX dar.

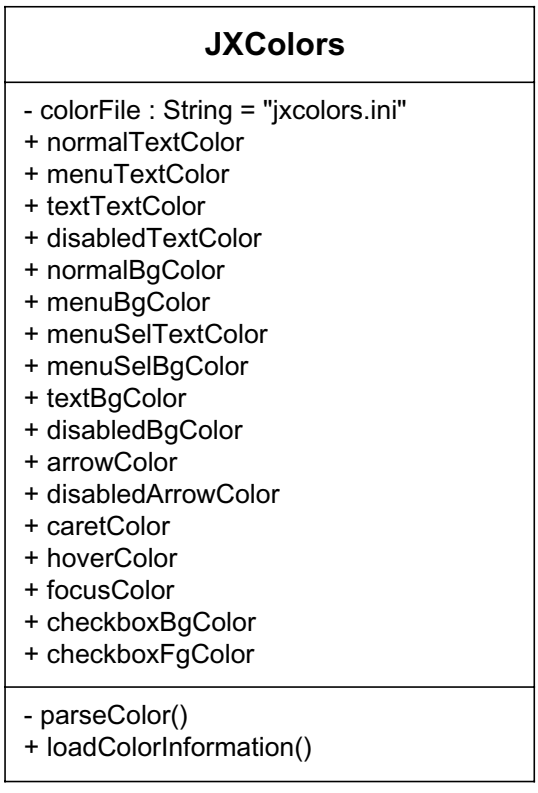


Abb. 4.44 Die Klasse JXColors

Um eigene Farben einzuführen, gibt es die Methode `loadColorInformation()`. Diese lädt bei Aufruf die Datei `jxcolors.ini` aus dem JX-Paket, interpretiert deren Inhalt, und setzt die Farbobjekte entsprechend neu. Für ein eigenes, benutzerdefiniertes Farbschema reicht es somit aus, die Datei `jxcolors.ini` dem jeweiligen Geschmack anzupassen.

Zum Laden der Datei `jxcolors.ini` verwendet die Klasse `JXColors` einige JX-spezifische Klassen, welche die Verwaltung des Boot-Filesystems von JX, kurz BootFS, übernehmen. Dieses Dateisystem enthält die Dateien, die mit dem Betriebssystem zu einer Distribution zusammengepackt wurden. Von diesem Dateisystem wird die Datei `jxcolors.ini` in ein `ReadOnlyMemory`-Objekt von JX geladen, und daraus ein `ConfigFile`-Objekt erstellt. Dieses wird verwendet, um die Datei zu interpretieren und die relevanten Farbinformationen zu extrahieren.

Sollte für ein Farbobjekt kein Eintrag in der Datei `jxcolors.ini` vorhanden sein, so wird diesem Objekt die Standartfarbe Grau mit den RGB-Werten (128, 128, 128) zugeordnet.

4.9 Implementierung der Grafik-Unterstützung

Die Unterstützung von Grafiken oder “Images” ist unter der AWT nur marginal vorhanden: Keine Komponente (und damit auch kein Peer) verwendet Grafiken in ihren Zeichenmethoden, so dass nur abgeleitete Komponenten durch Überschreiben der `paint()`-Methode Grafiken darstellen können (vgl. Abschnitt 2.2.7). Jedoch erlaubt es die Schnittstelle der AWT, Grafiken zu laden, diese durch Filter zu manipulieren und auf dem Bildschirm auszugeben.

Diese Unterstützung, v.a. das Laden und Verwalten der Grafiken, ist aber nur möglich mit Hilfe einer betriebssystemspezifischen Implementierung. Im Folgenden wird nun die vorliegende Implementierung erläutert.

Anmerkung: Wie bereits in Abschnitt 2.2.7 erklärt, verläuft das Laden einer Grafik normalerweise parallel zum eigentlichen Programmablauf. Dies wird in der vorliegenden Implementierung nicht unterstützt, d.h. ein entsprechender Aufruf einer Lade-Methode kehrt erst zurück, wenn der Ladevorgang beendet ist. Der Grund hierfür ist, dass das Scheduling unter JX zum Zeitpunkt dieser Arbeit mittels eines nichtpräemptiven Schedulers durchgeführt wurde, der es sehr schwierig machte, das gleichzeitige Abarbeiten mehrerer Vorgänge in der AWT zu realisieren.

4.9.1 Die Klasse JXImage

Zur Darstellung von Grafiken verwendet die AWT die abstrakte Klasse `Image`. Diese stellt einfache Methoden zur Verwaltung und Manipulation der repräsentierten Grafik zur Verfügung, und muss für die jeweilige Betriebssystemumgebung speziell implementiert werden.

Die Implementierung unter JX trägt den Namen `JXImage` und implementiert die wichtigsten Methoden. Es handelt sich bei ihr im Wesentlichen um eine Kapselung der Klasse `WBitmap` des Windowmanagers. Der Windowmanager verwendet die Klasse `WBitmap`, um damit seiner-

seits Grafiken zu speichern und auf den Bildschirm zu zeichnen. Die Klasse `JXImage` besteht deswegen hauptsächlich aus geerbten Methoden der Klasse `Image`, die in Methodenaufrufe für das interne `WBitmap`-Objekt umgewandelt werden, sowie einer Schnittstelle zum Erzeugen und Ansprechen desselben.

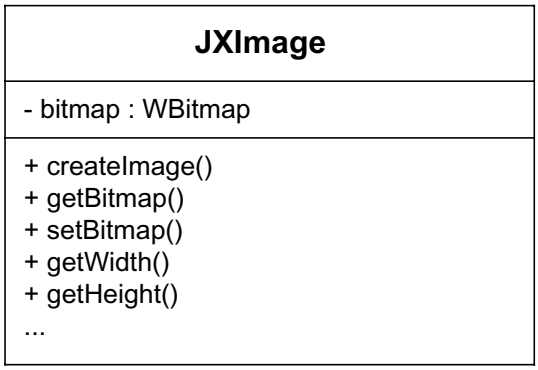


Abb. 4.45 Die Klasse JXImage

Die Klasse besitzt keinen öffentlichen Konstruktor, da, wie gleich noch näher erläutert wird, ein `Image`- bzw. `JXImage`-Objekt nicht direkt instanziiert wird, sondern nur über bestimmte Methoden erlangt werden kann, ähnlich einem `Graphics`-Objekt. Um eine Instanz erzeugen zu können, besitzt sie eine statische Methode namens `createImage()`, die von den entsprechenden Methoden verwendet wird.

4.9.2 Die Grafikschnittstelle der AWT

Wie alle anderen nativen Ressourcen, werden auch Grafiken von der globalen Instanz der Klasse `Toolkit` bezogen. Diese bietet dazu eine Reihe von Methoden an, von denen die JX-Implementierung `JXToolkit` einen kleinen Satz an grundlegenden Methoden unterstützt:

- `public Image getImage(String name)`
- `public Image createImage(ImageProducer producer)`

Beide Methoden liefern ein `Image`-Objekt zurück, welches die entsprechende Grafik beinhaltet. Die Implementierung dieser Methoden ist so gestaltet, dass einfach Objekte der Klassen `JXImageLoader` bzw. `JXImageCreator` erstellt werden, und ihnen die Arbeit übertragen wird. Die konkrete Beschreibung dieser Klassen folgt in späteren Abschnitten.

4.9.3 Das Zeichnen von Grafiken

Der Windowmanager bzw. jedes Fenster des Windowmanagers bietet von Haus aus die Möglichkeit, ein `WBitmap`-Objekt in einen bestimmten Bereich des Fensters zu zeichnen. Die AWT hingegen kennt nur in der Klasse `Graphics` eine Methode `drawImage()` zum Zeichnen eines `Image`-Objekts. Da die vorliegende Implementierung dieser Klasse, die Klasse `JXGra-`

`phics`, mit dem entsprechenden Konnektor verbunden ist, ließ sich das Zeichnen einer Grafik realisieren, indem die `JXGraphics`-Methode `drawImage()` so implementiert wurde, dass sie die entsprechende Methode des Konnektors aufruft, mit dem internen `WBitmap`-Objekt der `JXImage`-Klasse als Parameter.

4.9.4 Das Laden von Grafiken

Um eine Grafik von einem Dateisystem zu laden, wird die Methode `getImage()` der Klasse `Toolkit` verwendet. In der vorliegende Implementierung übergibt diese die Aufgabe einem neu erzeugten `JXImageLoader`-Objekt.

Die Klasse `JXImageLoader` kennt nur eine wichtige Methode namens `loadImage()`. Diese bekommt den Dateinamen übergeben, kontrolliert den gesamten Ladevorgang, und gibt zum Schluss ein `JXImage`-Objekt zurück.

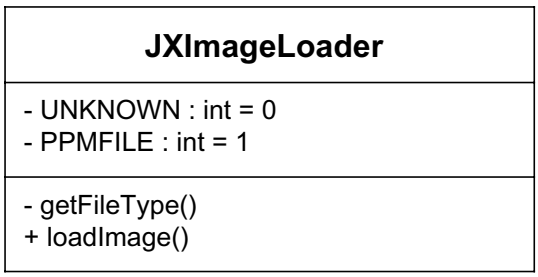


Abb. 4.46 Die Klasse JXImageLoader

Die Klasse arbeitet mithilfe von sogenannten Parsern: Diese Klassen implementieren das eigens definierte `JXImageParser`-Interface und sind für den eigentlichen Ladevorgang verantwortlich. Die Parser werden benutzt, um die Bilddaten aus der Datei zu laden und diese in ein einheitliches Format zu konvertieren. Über die Methoden des `JXImageParser`-Interfaces kommuniziert die Klasse `JXImageLoader` mit den Parsern und erhält so die Bilddaten der zu ladenden Grafik. Dieses Konzept ermöglicht die Einbindung von eigenen Parsern für neue Dateitypen.

```
public interface JXImageParser {
  public void readImageFile(String name);
  public int getImageWidth();
  public int getImageHeight();
  public int getRedAt(int x, int y);
  public int getGreenAt(int x, int y);
  public int getBlueAt(int x, int y);
}
```

Abb. 4.47 Das Interface JXImageParser

Der grobe Ablauf der Methode `loadImage()` sieht wie folgt aus:

- Der Dateityp wird anhand der Dateiendung ermittelt, und ein Objekt des zugehörigen Parsers erzeugt.
- Mittels `readImageFile()` wird der Parser angewiesen, die Datei zu verarbeiten.
- Die Dimension des zu ladenden Bildes wird vom Parser ermittelt, und ein neues `JXImage`-Objekt wird durch dessen statische Methode `createImage()` erzeugt.
- Der Übernahmevorgang beginnt: Die Bilddaten werden zeilenweise von oben vom Parser abgerufen, und mittels einiger Klassen des Windowmanagers in das `WBitmap`-Objekt der erzeugten `JXImage`-Klasse geschrieben.
- Am Ende der Übernahme wird das fertige `JXImage`-Objekt zurückgeliefert.

4.9.5 Unterstützte Bildformate

Eine vollständige Implementierung der AWT unterstützt als Grafikformate mindestens GIF und JPEG. Diese Formate wurden in der vorliegenden Implementierung jedoch nicht miteingebunden, da einerseits der Aufwand für die Implementierung der Algorithmen nicht unerheblich wäre, und andererseits diese Formate an gewisse lizenzrechtliche Bestimmungen gebunden sind.

Stattdessen wurde ein Parser für das Bildformat PPM entwickelt. Dieses freie Format benutzt keinerlei Komprimierungs- oder Verschlüsselungsalgorithmen, sondern speichert die Bilddaten sozusagen “roh” ab. Das macht Bilder dieses Formats im Vergleich zu anderen Bildformaten zwar sehr speicherintensiv, erlaubt jedoch einfache und nachvollziehbare Algorithmen zum Lesen und Schreiben. Das Format kennt sechs verschiedene Untertypen, die sich ergeben aus der Kombination von Farbauflösung (Schwarz-Weiß, 8-bit Graustufen, 24-bit Farbe) und internem Speicherformat (binär oder Text).

Der Parser, der in der Klasse `PPMParser` realisiert ist, unterstützt bis jetzt nur das P6-Format mit 24 Bit Farbauflösung und binär-Darstellung, kann aber theoretisch auch auf die anderen Unterformate von PPM erweitert werden.

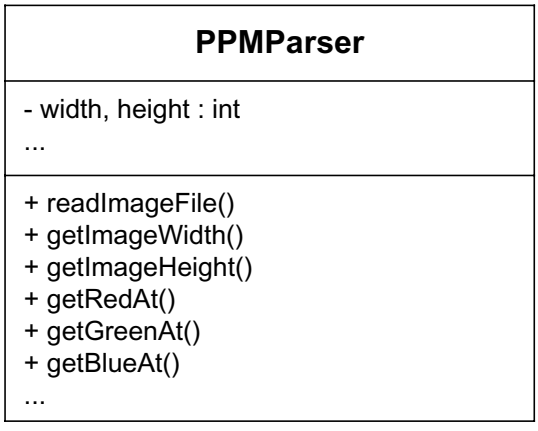


Abb. 4.48 Die Klasse PPMParser

Für die Erstellung der Testbilder wurde das freie Grafiktool XV verwendet [XV94]. Das ist insofern wichtig, als die Struktur eines PPM-Dateiheaders nicht eindeutig definiert ist, und der Parser an die Ausgabe des Programms XV angepasst wurde. Dies betrifft allerdings nur den Header, nicht die Bilddaten selbst.

4.9.6 Das Erzeugen von Grafiken

Die Methode `createImage()` der Klasse `Toolkit` dient dem Erstellen von Grafiken, wobei die Quelle ein Objekt ist, welches das `ImageProducer`-Interface implementiert. Das Konzept, welches hinter diesem Interface steht, wurde bereits in Abschnitt 2.2.8 erläutert: Dieses `ImageProducer`-Objekt erwartet, dass sich ein passendes `ImageConsumer`-Objekt bei ihm registriert und es anweist, mit der Übertragung zu beginnen. Dann überträgt das `ImageProducer`-Objekt Bilddaten, indem es diese über die entsprechenden Methoden des `ImageConsumer`-Objekts an dieses weitergibt.

Das nötige `ImageConsumer`-Objekt wurde in der Klasse `JXImageCreator` realisiert: Diese implementiert das Interface und alle wichtigen Methoden, die von einem Erzeuger aufgerufen werden könnten. Außerdem kennt sie eine eigene Methode `createImage()`, welche die ganze Arbeit erledigt und ein fertiges `JXImage`-Objekt zurückgibt. Abbildung 4.49 zeigt die Klasse `JXImageCreator` im Überblick.

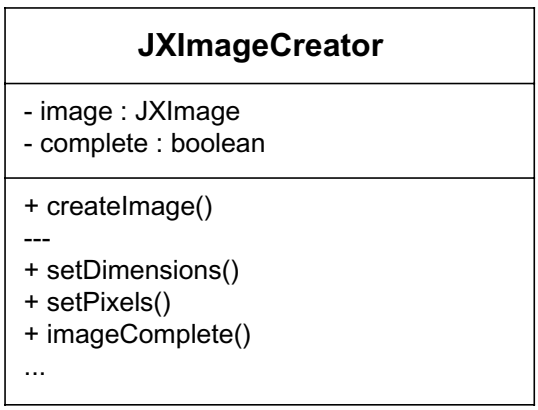


Abb. 4.49 Die Klasse JXImageCreator

Der Ablauf beim Aufruf der Methode `createImage()` der Klasse `JXToolkit` sieht folgendermaßen aus:

- Ein neues `JXImageCreator`-Objekt wird erzeugt und dessen `createImage()`-Methode aufgerufen.
- Diese registriert das Objekt beim Erzeuger und startet die Übertragung.
- Der Erzeuger ruft im Laufe der Übertragung eine Vielzahl von Methoden an dem Objekt auf, wobei die Methoden `setPixels()` und `setDimensions()` am wichtigsten sind: `setDimensions()` erzeugt das `JXImage`-Objekt, in dem die Bilddaten gespei-

chert werden, und muss dementsprechend bald aufgerufen werden. Die Methode `setPixels()` schließlich speichert die übergebenen Bilddaten in das `JXImage`-Objekt, wobei der dabei verwendete Algorithmus analog zu dem der Klasse `JXImageLoader` arbeitet (vgl. Abschnitt 4.9.4).

– Sobald der Erzeuger seine Arbeit beendet hat, wird das fertige `JXImage`-Objekt zurückgeliefert.

4.10 Eigene Erweiterungen der AWT-Schnittstelle

Diese Implementierung der AWT bietet neben den standardmäßig vorhandenen AWT-Komponenten und -Klassen noch ein paar weitere an, welche dem Anwender und Programmierer das Erstellen von grafischen Anwendungen erleichtern sollen. Diese Klassen implementieren dafür gewisse Funktionalitäten, welche in der Standard-AWT nicht oder nur ungenügend realisiert sind, in der Programmierpraxis aber von grossen Wert wären.

Anmerkung: Um diese Erweiterungen als solche zu kennzeichnen, wurden diese in ein eigenes Package **jx.awt** gepackt. Um die Erweiterungen in einer Anwendung nutzen zu können, ist es folglich notwendig, dieses Package in das Programm miteinzubinden.

4.10.1 Die Klasse ExtendedLabel

Ein großer Nachteil, der sich bei der Verwendung der Klasse `Label` ergibt, ist die Tatsache, dass sie nur einen einzeiligen Text erlaubt, während es oft sinnvoll und wünschenswert wäre, einen ganzen Absatz an Text auf den Bildschirm zu bringen. Dieses Manko soll die Klasse `ExtendedLabel` ausgleichen: Sie erlaubt die Ausgabe von mehrzeiligem Text auf dem Bildschirm.

Die Benutzung dieser Klasse verläuft vollkommen analog zu der der Klasse `Label`. Der einzige Unterschied zwischen beiden Klassen ist, dass der Anzeigetext für `ExtendedLabel` sogenannte “new line”-Zeichen beinhalten darf (unter JX ’\n’), an denen dann der Text für die Anzeige umgebrochen wird. Ansonsten kann die Klasse wie jede andere normale AWT-Komponente auch behandelt werden.

Die Klasse ist von der AWT-Komponente `Label` abgeleitet, besitzt aber weder eigene Variablen noch eigene Methoden. Stattdessen überprüft der Peer `JXLabelPeer`, ob es sich bei seiner zugehörigen Komponente um die Klasse `Label` oder `ExtendedLabel` handelt, und verhält sich entsprechend (vgl. Abschnitt 4.4.3).

4.10.2 Die Klasse ExtendedPanel

Die Klasse `ExtendedPanel` erweitert die `Panel`-Klasse um die Möglichkeit, einen sichtbaren Rahmen um das Panel zu zeichnen. Die Klasse kennt dafür einen Satz verschieden aussehender Rahmen sowie die Möglichkeit, diese mit einem Titel zu versehen.

Auch diese Klasse wird wie eine normale AWT-Komponente behandelt. Sie kennt verschiedene Konstruktoren, wobei der “Ausführlichste” folgende Parameter erwartet:

- Einen Layoutmanager, der die eingebetteten Klassen ordnen soll.
- Den Rahmentyp. Jeder Rahmen hat eine Dicke von 5 Pixel und wird dem Typ entsprechend gezeichnet. Momentan kennt die Klasse folgende Rahmentypen:
  - BORDER_NONE: Keine weitere Dekoration
  - BORDER_LINE: Zeichnet eine einfache Linie als Rahmen
  - BORDER_RAISED: Zeichnet einen 3D-Rahmen, der das Panel optisch nach oben hebt
  - BORDER_LOWERED: Zeichnet einen 3D-Rahmen, der das Panel optisch nach unten drückt
  - BORDER_ETCHED: Eine Mischung der beiden vorhergehenden Rahmentypen, die den Rahmen wie einen “Schnitt” in der Oberfläche aussehen lässt
- Den Titel des Rahmens. Falls diese Referenz null ist, ist der Rahmen titellos, ansonsten wird der Text rechts neben die linke obere Ecke des Rahmens geschrieben.

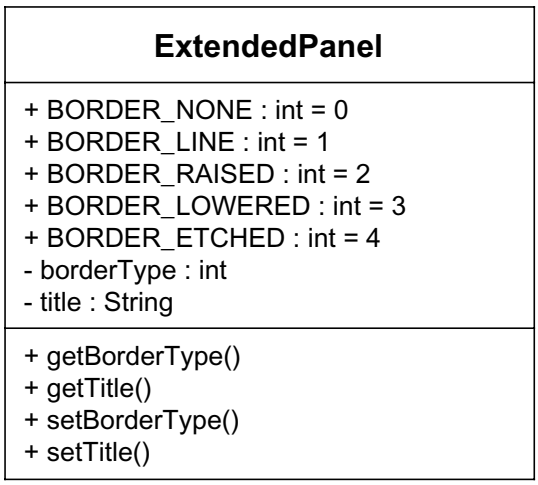


Abb. 4.50 Die Klasse ExtendedPanel

4.10.3 Die Klasse MessageDialog

Diese Klasse ist keine erweiterte AWT-Komponente, sondern soll eine Reihe verschiedener Dialogfenster zur Verfügung stellen, welche die Interaktion mit dem Benutzer erleichtern sollen. Abbildung 4.51 zeigt die Klasse im Überblick.

Momentan kennt die Klasse nur die statische Methode `infoMessageDialog()`. Diese stellt eine Art einfaches Dialogfenster dar, das für einfache Hinweise und Warnungen verwendet werden kann. Es wird vor dem Sichtbarmachen erst auf dem Bildschirm zentriert, so dass der Be-

nutzer es nicht übersehen kann. Dieses Fenster beinhaltet einen Infotext für den Benutzer, und darunter einen Knopf, der das Fenster bei Benutzung verschwinden lässt. Schließen lässt sich das Fenster auch durch Klicken auf dessen Close-Button.

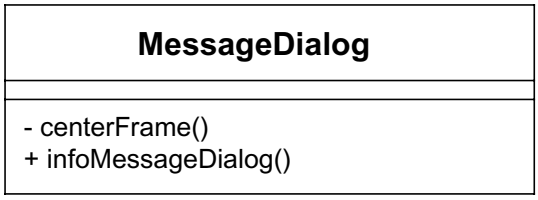


Abb. 4.51 Die Klasse MessageDialog

Als Parameter werden drei Zeichenketten übergeben: Ein Titel für das Fenster, der Infotext, der angezeigt werden soll, sowie der Text, der auf dem Knopf stehen soll. Da für den Infotext ein `ExtendedLabel`-Objekt benutzt wird, darf der Text auch mehrzeilig sein (vgl. Abschnitt 4.10.1).

4.11 Zusammenfassung

In diesem Kapitel wurde die Implementierung sämtlicher Peers und sonstiger Klassen besprochen. Es wurde dabei u. a. auf den Aufbau von grundlegenden Klassen, sowie den Peers und Konnektoren für normale Komponenten und für Komponenten des Fenster- und Menüsystems eingegangen. Daneben wurde die Implementierung der Grafik-Unterstützung erläutert, und zum Abschluss noch auf die zusätzlich entwickelten Erweiterungen für die AWT eingegangen.

5 Anmerkungen & Einschränkungen der Implementierung

5.1 Überblick

Dieses Kapitel geht auf einige Themen ein, die für diese Arbeit ebenfalls von Bedeutung waren. Bei diesen Themen handelt es sich um die Verwendung von Klassen des Betriebssystems JX, um Modifikationen an der verwendeten Classpath-Implementierung, sowie um Einschränkungen der vorliegenden AWT-Implementierung.

5.2 JX-spezifische Elemente

Bei der Erstellung dieser Arbeit konnte größtenteils auf die Verwendung von speziellen Klassen des Betriebssystems JX verzichtet werden, denn die Schnittstelle zum Windowmanager war bereits auf einem ausreichend hohen Niveau, so dass ein Zugriff auf spezielle Komponenten des Betriebssystems nicht notwendig war. Dennoch haben sich im Laufe der Arbeit einige Klassen ergeben, die an die spezielle Laufzeitumgebung angepasst werden mussten.

Die erste Ausnahme bildet hier die Klasse `EventQueue` der AWT. Diese musste komplett umgestaltet werden, da das Betriebssystem den Java-Standard nicht zu 100% unterstützt. Eine genauere Beschreibung der Implementierung findet sich im Abschnitt 4.3.2.3.

Als zweites musste die Klasse `JXMenuThread` angepasst werden. Auch diese selbstgestaltete Klasse benötigt eine gewisse Funktionalität des Java-Standards, die JX (noch) nicht zur Verfügung stellt, aber dafür eigene Konstrukte implementiert, die anstelle des Standards verwendet wurden. Abschnitt 4.6.9 enthält genauere Informationen über die Implementierung der Klasse.

Als drittes wurden die Klassen `PPMParser` und `JXColors` ein wenig modifiziert. Diese greifen auf externe Dateien zu, jedoch wird dieser Zugriff unter JX nicht durch die üblichen Java-Klassen gehandhabt. Die Modifikationen dazu finden sich in Abschnitt 4.9.5 bzw. Abschnitt 4.8.2.

5.3 Modifikationen an der Classpath-Implementierung

Eines der Ziele dieser Arbeit war, während der JX-Implementierung der AWT die AWT-Klassen des Classpath Projekts möglichst wenig zu modifizieren. Der Gedanke dahinter war, dass es möglich sein sollte, die momentan verwendete Version der AWT-Klassen zu einem späteren Zeitpunkt möglichst ohne Modifikationen durch eine neuere Version derselben zu ersetzen. Ganz ohne Modifikationen an den Klassen war eine Implementierung jedoch nicht möglich, vor allem aus drei Gründen:

- Obwohl die für diese Arbeit verwendete Version 0.03 der AWT-Klassenbibliothek des Classpath Projekts schon sehr ausgereift und umfangreich implementiert worden war, befand sie sich dennoch immer noch in einem Beta-Stadium, und war dementsprechend nicht ganz frei von Fehlern.
- Einige Komponenten waren bis zu diesem Zeitpunkt noch zu unzureichend implementiert worden, um voll funktionsfähig zu sein. So fehlten z.B. häufig entscheidende Methoden für das 1.1-Ereignismodell der AWT.
- Viele der bereits vorhandenen Methoden(-rümpfe) wurden entfernt oder modifiziert, da das JX-Betriebssystem keine ausreichende Unterstützung für diese Methoden bereitstellte. Ein Problem hier war z.B. die Tatsache, dass der Java-Nativecode-Compiler von JX den Datentyp `long` nicht unterstützt, so dass alle Methoden, welche diesen Typ verwendeten, entsprechend angepasst werden mussten.

Eine genaue und vollständige Liste aller Modifikationen aufzuführen, von denen die meisten auf jeweils eine Zeile im Code beschränkt sind, würde den Rahmen dieser Arbeit sprengen. Die wichtigsten Modifikationen sind jedoch an der jeweils entsprechenden Stelle in dieser Arbeit beschrieben worden.

5.4 Einschränkungen der Implementierung

Trotz einiger Modifikationen an der Classpath-AWT und am Windowmanager war es nicht möglich, die komplette AWT für das Betriebssystem JX zu implementieren. Welche Klassen und Konzepte aus welchen Gründen nicht realisiert wurden, ist in der folgenden Liste aufgeführt:

- Die Klasse `Dialog` und deren Ableitungen wurden nicht implementiert. Der Grund dafür war die fehlende Unterstützung für voneinander abhängige Fenster in den Klassen `Frame` und `Dialog` der Classpath AWT.
- Die Klasse `Applet` wurde nicht implementiert, da das Betriebssystem JX bis jetzt keine Unterstützung für die Ausführung von Applets vorsieht.
- Das 1.0-Ereignismodell der AWT wurde nicht implementiert. Dieses Modell ist mittlerweile veraltet, und wird in aktuellen Java-Anwendungen so gut wie gar nicht mehr verwendet.
- Die Klasse `JXGraphics` unterstützt nicht die komplette `Graphics`-Schnittstelle. Dies liegt v.a. an der unzureichenden Unterstützung des Windowmanagers, der nur wenige grundlegende Zeichenoperationen durchführen kann.
- Die Peers sind teilweise speziell auf den Standardfont des Windowmanagers angepasst. Wegen fehlender anderer Zeichensätze konnten keine allgemeinen Routinen zur Handhabung von Zeichensätzen mit fester oder variabler Buchstabenbreite getestet werden.

- Das Laden von Grafiken von einem Dateisystem geschieht im Gegensatz zum AWT-Standard nicht nebenläufig zur Programmausführung, sondern wird synchron durchgeführt, was durch das von JX verwendete Scheduling bedingt ist (vgl. Abschnitt 4.9).
- Die vorhandene Unterstützung für Grafiken erlaubt keine Verwendung von GIF- oder JPG-Dateien, sondern verarbeitet nur das PPM-Format (vgl. Abschnitt 4.9.5).

5.5 Zusammenfassung

Es wurde gezeigt, dass die vorliegende Implementierung der AWT für JX mit sehr wenigen JX-spezifischen Klassen auskommt. Außerdem wurde erklärt, dass es sich manchmal nicht vermeiden ließ, einige Modifikationen an der verwendeten Classpath-Implementierung vorzunehmen, um die gewünschte Funktionalität zu erreichen. Abschließend wurden einige Punkte erläutert, die sich trotz dieser Bemühungen nicht implementieren ließen.

6 Zusammenfassung

6.1 Ziele dieser Arbeit

Aufgabe dieser Studienarbeit war es, eine JX-spezifische Version der AWT von Java zu implementieren. Diese soll es Entwicklern ermöglichen, mittels der bekannten AWT-Schnittstellen eigene, grafisch ansprechende Anwendungen für das Java-Betriebssystem JX zu entwerfen.

Der Schwerpunkt der Arbeit lag auf der Entwicklung der Peer-Klassen, und damit der Realisierung der entsprechenden AWT-Komponenten, welche den Kern jeder AWT-basierten Anwendung bilden. Es war nicht Ziel der Arbeit, eine komplette, AWT-kompatible Bibliothek aus dem Nichts aufzubauen, sondern die bestehende Implementierung des Classpath Projekts unter Verwendung des Windowmanagers für JX zu einer funktionsfähigen JX-Version zu ergänzen.

6.2 Stand der JX-Implementierung

Das Vorhaben, die AWT so vollständig wie möglich unter JX zu implementieren, ließ sich sehr gut realisieren, auch wenn es nicht möglich war, die komplette AWT zu implementieren. Gründe hierfür waren u.a. fehlende Strukturen in der verwendeten Betaversion von Classpath, fehlende Unterstützung durch den Windowmanager oder durch JX selbst. Dennoch stellt die JX-Implementierung der AWT ein ausreichendes Fundament dar, um die Entwicklung einfacher, AWT-basierter Anwendungen für JX zu ermöglichen.

Nachfolgend werden nochmal alle Klassen und Konzepte aufgeführt, welche in dieser Arbeit realisiert wurden:

- Es wurde ein Großteil aller normalen Komponenten der AWT funktionsfähig implementiert, als da wären: `Button`, `Canvas`, `Checkbox`, `Choice`, `Label`, `List`, `Scrollbar`, `TextArea`, `TextField`, `Panel`, `ScrollPane` und `Frame`. Ebenso wurden die dazu notwendigen abstrakten Komponenten implementiert, die wie folgt lauten: `Component`, `Container`, `TextComponent` und `Window`. Zu all diesen Komponenten wurden entsprechende Peers und Konnektoren geschrieben, welche die jeweiligen Komponenten auf Betriebssystemebene realisieren (vgl. Abbildung 6.1).
- Ein Menüsystem wurde realisiert, indem alle Menü-Komponenten funktionsfähig implementiert wurden: `MenuComponent`, `MenuBar`, `MenuItem`, `CheckboxMenuItem`, `Menu` und `PopupMenu`. Auch für diese Klassen wurden entsprechende Peers und Konnektoren entwickelt (vgl. Abbildung 6.1). Das Menüsystem unterstützt eine beliebige Anzahl von Einträgen, sowohl in der Menüleiste als auch in jedem Menü selbst, die Kennzeichnung eines Menüs als “Hilfe”-Menü, sowie die Verwendung von Untermenüs.
- Für die abstrakten Klassen `Toolkit` und `Graphics` wurden entsprechende Implementierungen geschrieben, welche die Schnittstelle zum darunterliegenden Betriebssystem sowie zum Windowmanager realisieren.

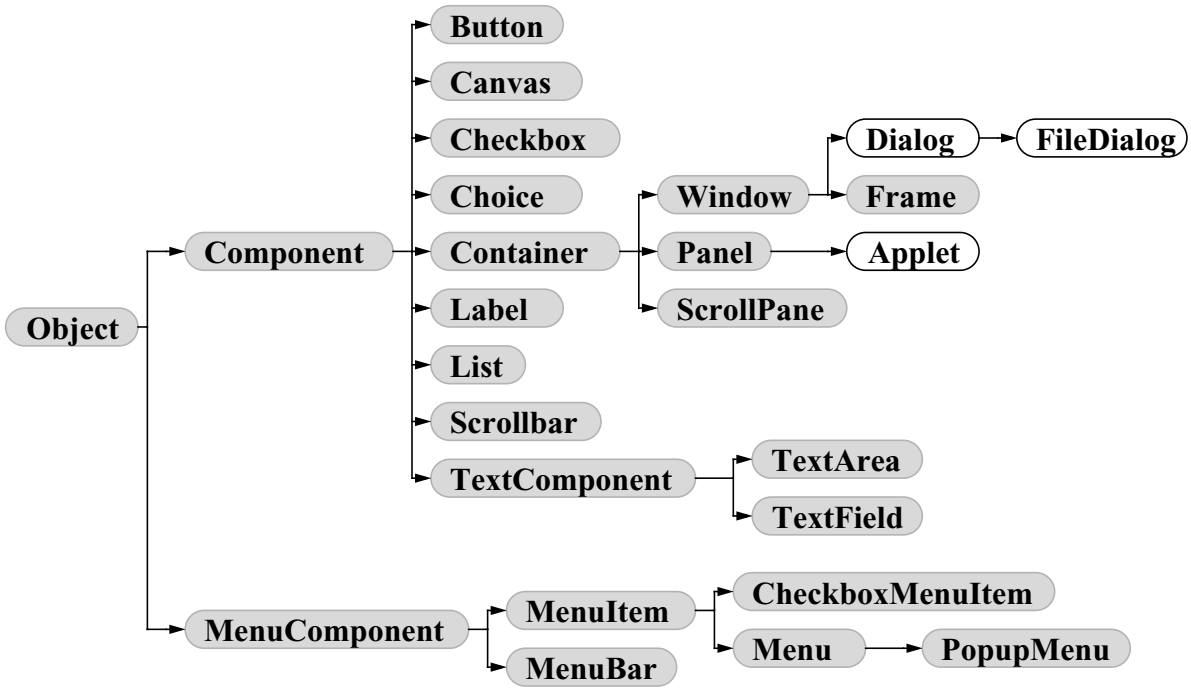


Abb. 6.1 Die Liste der implementierten Komponenten (grau bedeutet implementiert)

- Das 1.1-Ereignismodell der AWT wurde in den Komponenten komplett implementiert, sowie beinahe alle vorhandenen Event-Klassen und deren zugehörige "EventListener" eingebunden. Folgende Event-Klassen wurden dabei eingebunden: `ActionEvent`, `AdjustmentEvent`, `ComponentEvent`, `ContainerEvent`, `FocusEvent`, `InputEvent`, `ItemEvent`, `KeyEvent`, `MouseEvent`, `PaintEvent`, `TextEvent` und `WindowEvent`. Dazu werden folgende "EventListener" unterstützt: `ActionListener`, `AdjustmentListener`, `ComponentListener`, `ContainerListener`, `FocusListener`, `ItemListener`, `KeyListener`, `MouseListener`, `MouseMotionListener`, `TextListener` und `WindowListener`. Jede implementierte Komponente verhält sich gemäß der Vorgabe der Originals, d.h. sie sendet die gleichen Events zu den gleichen Zeitpunkten, wie eine Original-Komponente der AWT.
- Die grundlegende Unterstützung von `Image`-Objekten wurde realisiert: Grafiken können mittels der `Toolkit`-Methoden `getImage()` oder `createImage()` von einem Dateisystem geladen oder aus einem Speicherbereich erzeugt werden, und durch die entsprechenden Methoden eines `Graphics`-Objekts dargestellt werden. Das `JXImageParser`-Interface erlaubt es dabei, selbst Parser für weitere Grafikformate zu schreiben, und diese dann ebenfalls unter der JX-AWT verfügbar zu machen. Für die Erzeugung aus einem Speicherbereich wurde außerdem die Klasse `MemoryImageSource` miteingebunden.

- Der Tastaturfokus wird unterstützt: Die Komponenten `Button`, `Checkbox`, `Choice`, `List`, `TextArea` und `TextField` unterstützen die Verwendung und Darstellung des Tastaturfokus, und der Benutzer hat die Möglichkeit, durch Drücken der Taste <**Tab**> den Fokus zwischen diesen Komponenten weiterzureichen, sofern sie sich in einem gemeinsamen Fenster befinden.
- Die Verwendung von benutzerdefinierten Komponenten, die z.B. von `Canvas` abgeleitet wurden, wird ebenfalls unterstützt.
- Die bestehende Implementierung bietet Unterstützung für verschiedene Farbschemata. Durch die Klasse `JXColors` ist es möglich, eine einheitliches Farbschema für die AWT-Komponenten zu wählen (vgl. Abschnitt 4.8.2).
- Als Ergänzung der Standard-Komponenten der AWT wurden die Komponenten `ExtendedLabel` und `ExtendedPanel` entwickelt, welche es erlauben, mehrzeilige Textbereiche bzw. Kontainer mit Umrandung und/oder Überschrift in einem Fenster zu verwenden. Außerdem wurde die Klasse `MessageDialog` geschrieben, die ein Dialogfenster für einfache Hinweise zur Verfügung stellt.

Nachfolgend sind noch diejenigen Konzepte aufgelistet, welche ebenfalls funktionsfähig sind, jedoch nicht durch diese Arbeit, sondern bereits durch die Classpath AWT implementiert wurden:

- Die interne Verwaltung eines Kontainers
- Das Layoutmanagement eines Kontainers

6.3 Anwendungsbeispiele

Für die AWT-Implementierung unter JX wurden zwei Anwendungen zur Demonstration der Fähigkeiten dieser Implementierung entworfen:

- Eine einfache grafische Oberfläche, ähnlich denen von bekannten kommerziellen oder frei verfügbaren Betriebssystemen, die eine Art Taskleiste am unteren Bildschirmrand beinhaltet, und die Verwendung eines Hintergrundbildes unterstützt. In der Taskleiste ist links unten ein Knopf integriert, welcher ein Demo-Menü öffnen kann. Durch die Einträge dieses Menü ist es möglich, verschiedene Fenster zu öffnen, welche die Fähigkeiten der einzelnen AWT-Komponenten demonstrieren. Abbildung 6.2 zeigt einen Screenshot der grafischen Oberfläche.
- Ein MineSweeper-Klon, der von den wichtigsten Konzepten, wie Komponenten, Kontainern, benutzerdefinierten Komponenten, dem Menüsystem, sowie dem 1.1-Ereignismodell Gebrauch macht. Ein Screenshot des Spiels ist in Abbildung 6.3 dargestellt.

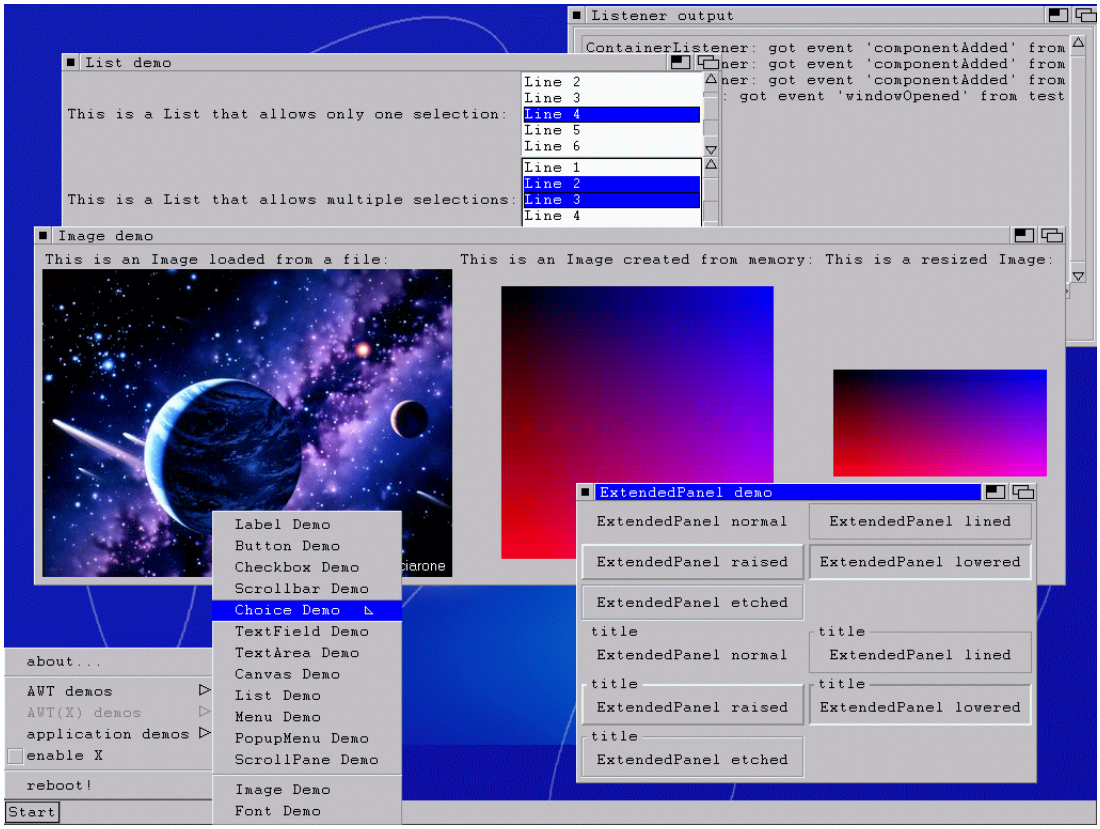


Abb. 6.2 Die grafische Demo-Oberfläche

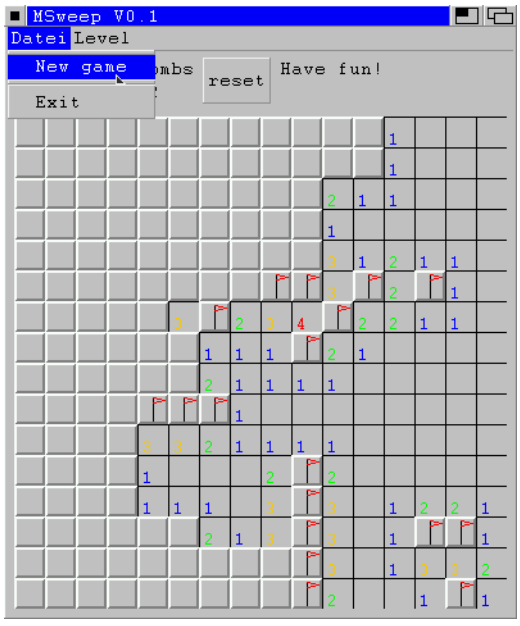


Abb. 6.3 Der MineSweeper-Klon MSweep

7 Literaturverzeichnis

Verwendete Literatur:

GF+02 Michael Golm, Mike Felser, Christian Wawersich, Jürgen Kleinöder: *The JX Operating System*. 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, CA, pp. 45-58

GK00 Michael Golm, Jürgen Kleinöder: *JX: Eine adaptierbare Betriebssystemarchitektur.* GI-Herbsttagung: Mobile Computing, 16./17.11.2000, München

JO02 Jürgen Obernolte: *Entwurf und Implementierung eines Windowmanagers für das Java-Betriebssystem JX*. Studienarbeit an der Universität Erlangen-Nürnberg, Februar 2002

PR98 Patrick Chan, Rosanna Lee: *The Java Class Libraries, Second Edition, Volume 2*. Addison-Wesley, 1998.

SM98 Sun Microsystems: *Creating a User Interface (AWT only)*, 1998. Link unter: http://java.sun.com/docs/books/tutorial/download/tut-OLDui.zip

Verwendete Software und Quellcode:

CP03 *GNU Classpath*, eine freie Implementierung der Java-Standardbibliothek, Beta-Version 0.03. Autor: Classpath Project. Homepage: www.gnu.org/software/classpath , Download unter: ftp://alpha.gnu.org/gnu/classpath/classpath-0.03.tar.gz

XV94 *XV*, ein universeller Bildbetrachter und -konverter. Autor: John Bradley. Homepage: www.trilon.com/xv , Download unter: ftp://ftp.cis.upenn.edu/pub/xv

Weiterführende Literatur:

GKB01 Michael Golm, Jürgen Kleinöder, Frank Bellosa: *Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System*. HotOS 2001, May 20-23, 2001, Elmau/Oberbayern, Germany

GOL02a Michael Golm: *Overview about the JX architecture.* Universität Erlangen-Nürnberg, 2002

GOL02b Michael Golm: *Programmer's Introduction to the JX architecture*. Universität Erlangen-Nürnberg, 2002

GW+01 Michael Golm, Christian Wawersich, Jörg Baumann, Meik Felser, Jürgen Kleinöder: *Understanding the Performance of the Java Operating System JX using Visualization Techniques*. Workshop on Software Visualization, OOPSLA 2001, Tampa, FL, October 15, 2001

JZ97 John Zukowski: *The Java AWT Reference*. O'Reilly, April 1997

SM99 Sun Microsystems: *The AWT in 1.0 and 1.1*, April 1999. Link unter: http://java.sun.com/products/jdk/awt/

Weiterführende Software und Quellen:

BGG02 *BISS AWT*, eine freie Implementierung der AWT mit eigenem "Look & Feel" und eigenen Erweiterungen. Autor: BISS GmbH Germany. Homepage und Download unter: http://www.biss-net.com/biss-awt.html