

## 5 Bedienoberflächen auf dem kleinsten gemeinsamen Nenner

*From a programmer's point of view, the user is a peripheral that types when you issue a read request.*  
– Peter Williams

MIDlets sollen auf verschiedenen Endgeräten ablauffähig sein, die sich hinsichtlich ihrer Eingabemöglichkeiten und Bildschirmformate stark voneinander unterscheiden. Das Spektrum erstreckt sich von einfachen Mobiltelefonen mit kleinen, monochromen Bildschirmen geringer Auflösung und wenigen Eingabetasten bis hin zu PDAs mit verhältnismäßig großen, hochauflösenden Farbdisplays und erweiterten Tastaturen oder eingebauten Mechanismen zur Handschrifterkennung.

Die User-Interface-Bibliothek des MIDP, die sich an dem »kleinsten gemeinsamen Nenner« der Fähigkeiten dieser Geräte orientiert, wird als *LCDUI* bezeichnet. Beim Entwurf der Bibliothek wurde berücksichtigt, dass die Besitzer von Mobiltelefonen nicht notwendigerweise erfahrene Computeranwender sind und die Anwendungen deshalb einfach und intuitiv bedienbar sein müssen.

Eigenartigerweise lässt die Spezifikation offen, wofür das Akronym *LCDUI* eigentlich steht. Dementsprechend haben sich verschiedene plausible Auslegungen wie zum Beispiel »Liquid Crystal Display User Interface«, »Lowest Common Denominator User Interface« und »Limited Connected Device User Interface« eingebürgert. Die Auflösung des Rätsels findet sich in einem frühen Werk über die Java 2 Micro Edition [RTV01], dessen Mitautor Mark Vandenbrink seinerzeit den *Java Specification Request 37* eingereicht hat, aus dem das MIDP 1.0 hervorging. Demnach lautet die ursprüngliche beabsichtigte Bezeichnung »Liquid Crystal Display User Interface«.

### 5.1 Ausprägungen des LCDUI

Das *LCDUI* hat zwei Ausprägungen: das High-Level-API und das Low-Level-API.

Das *High-Level-API* eignet sich für Applikationen, deren Portabilität besonders wichtig ist. Das High-Level-API abstrahiert von den im

*High-Level-LCDUI*

jeweiligen Gerät implementierten *User Interface Widgets* und bietet Applikationen wenige Ansatzpunkte, den *Look and Feel* der Bedienoberfläche zu beeinflussen. Das kommt in den folgenden Punkten zum Ausdruck:

- MIDlets haben keine Möglichkeit, die visuelle Gestaltung der User-Interface-Komponenten vorzugeben. Form, Farbe, Zeichensatz usw. liegen im Ermessen der LCDUI-Implementierung.
- Einfache Benutzerinteraktionen wie das Blättern in einem Dialog, dessen Länge über das physische Display hinausreicht, erledigt das High-Level-LCDUI intern – ohne die Applikation zu informieren.
- Eingabegeräte werden in der Applikation nicht direkt abgefragt. Das LCDUI empfängt alle Eingaben und benachrichtigt die Applikation, wenn erforderlich über wohldefinierte Listener-Methoden.

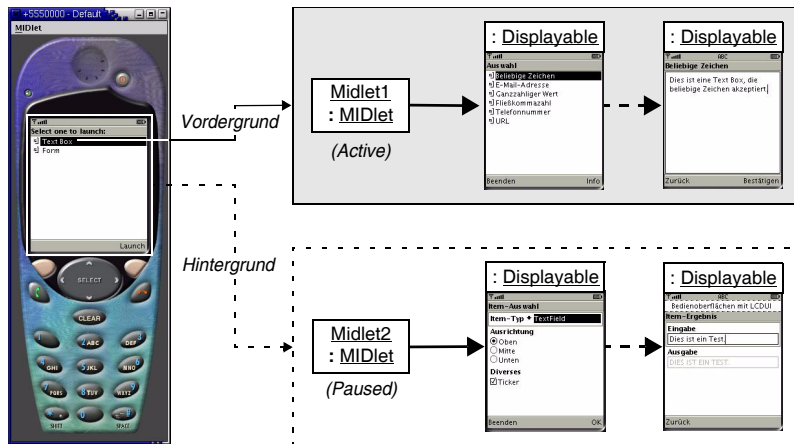
In den meisten LCDUI-Implementierungen werden die abstrakten Bedienelemente des High-Level-API auf die nativen Komponenten des Endgeräts abgebildet. Deswegen erscheinen MIDlets bei Verwendung des High-Level-API mit demselben *Look and Feel* wie die vom Gerätehersteller mitgelieferten, üblicherweise in C/C++ realisierten internen Anwendungen. Dieser Sachverhalt trägt zu einer besseren Bedienbarkeit bei: Der Benutzer findet in MIDlets die gleichen Bedienelemente vor, wie er sie aus nativen Applikationen kennt. Umgekehrt bedeutet das aber, dass sich ein und dasselbe MIDlet je nach Gerät unterschiedlich präsentieren kann.

#### Low-Level-LCDUI

Das *Low-Level-API* verwendet demgegenüber ein sehr niedriges Abstraktionsniveau. Diese Schnittstelle ist für Anwendungen ausgelegt, die eine exakte Kontrolle über Position und Aussehen der Bedienelemente ausüben und Eingaben direkt entgegennehmen möchten. Zu dieser Kategorie von Anwendungen zählen beispielsweise Spiele. Mithilfe des Low-Level-API können MIDlets:

- pixelgenau definieren, was auf das Display gezeichnet wird
- physische Tasten und andere Eingabegeräte abfragen
- Ereignisse wie das Drücken und Loslassen von Tasten empfangen

Das Low-Level-API macht gerätespezifische Eigenschaften wie die Bildschirmauflösung, Farbtiefe und die vorhandenen Tasten für die Anwendung sichtbar. Die Portabilität von MIDlets, die diese Eigenschaften ausnutzen, ist naturgemäß eingeschränkt. Nichtsdestoweniger ist es auch unter Verwendung des Low-Level-API möglich, portable MIDlets zu entwickeln. Beispielsweise lassen sich Bildschirmauflösung und Farbtiefe durch geeignete API-Aufrufe feststellen. Mit ihrer Hilfe können sich MIDlets flexibel an die Gegebenheiten der Ablaufumgebung anpassen.



**Abb. 5-1**  
Bedienoberflächen mit  
LCDUI

## 5.2 Das LCDUI-Modell

### 5.2.1 Gemeinsame Eigenschaften der UI-Komponenten

Die LCDUI-Klassenbibliothek ist in dem Paket `javax.microedition.lcdui` enthalten. Dialoge werden darin durch die Klasse `Displayable` repräsentiert. Zu jedem Zeitpunkt kann ein MIDlet höchstens ein `Displayable`-Objekt darstellen (vgl. Abbildung 5-1). Der Inhalt dieses Objekts ist jedoch nicht notwendigerweise ständig auf dem physischen Bildschirm sichtbar. Diese Einschränkung resultiert aus der Fähigkeit vieler Endgeräte, mehrere MIDlets parallel ablaufen zu lassen. Ein MIDlet befindet sich im Zustand *Active* und damit im »Vordergrund«: Das aktive MIDlet wird auf dem physischen Bildschirm präsentiert und alle Benutzereingaben werden dort verarbeitet. MIDlets im Zustand *Paused* sind im »Hintergrund«: Das `Displayable`-Objekt ist nicht sichtbar und erhält auch keine Benutzereingaben.

`javax.microedition.lcdui`

`Displayable` ist eine abstrakte Basisklasse, welche die gemeinsamen Eigenschaften der verfügbaren Dialogklassen zusammenfasst. Jede dieser Klassen kann, wie in Abbildung 5-2 zu sehen ist, die folgenden Eigenschaften haben:

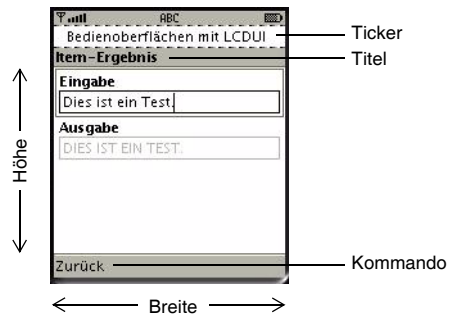
Eigenschaften

- einen Titel
- ein Laufband (engl. *Ticker*)
- eine Menge von Kommandos

`Displayable` bietet korrespondierende Methoden zum Setzen und Abfragen dieser Eigenschaften an.

Damit eine Anwendung auf Kommandos reagieren kann, bietet `Displayable` die Möglichkeit an, eine Listener-Funktion zu registrie-

**Abb. 5-2**  
Darstellung eines  
Displayable im WTK,  
in diesem Fall einer  
Form-Instanz



ren. Das LCDUI ruft diese Funktion automatisch auf, wenn der Benutzer ein Kommando auswählt. Der Umgang mit Kommandos und Listener-Funktionen wird ausführlich in Unterkapitel 5.3 besprochen.

Größe Mit den `Displayable`-Methoden `getWidth()` und `getHeight()` lässt sich die Breite beziehungsweise Höhe des Bildschirmausschnitts (als Anzahl von Bildpunkten) ermitteln, welcher der UI-Komponente zur Verfügung steht. Die Bereiche für Titel, Ticker, Kommandos usw. sind darin nicht eingeschlossen. Ändert sich der zugedachte Bildschirmbereich, so wird `sizeChanged()` mit der neuen Breite und Höhe als Parameter aufgerufen. Der Rumpf dieser Methode ist in der Basisklasse `Displayable` leer, kann aber in abgeleiteten Klassen überschrieben werden. `isShown()` gibt Auskunft, ob die UI-Komponente derzeit sichtbar ist.

<b>Displayable</b>	<b>javax.microedition.lcdui</b>
Object	
↳ Displayable	
2	String <b>getTitle()</b>
2	void <b>setTitle()</b>
2	Ticker <b>getTicker</b>
2	void <b>setTicker(Ticker ticker)</b>
	void <b>addCommand(Command cmd)</b>
	void <b>removeCommand(Command cmd)</b>
	void <b>setCommandListener(CommandListener l)</b>
2	int <b>getHeight()</b>
2	int <b>getWidth()</b>
	boolean <b>isShown()</b>
2	void <b>sizeChanged(int width, int height)</b>



Die Klasse `Screen` verfügt im MIDP 2.0 (anders als im MIDP 1.0) über keine zusätzlichen Methoden. Im MIDP 1.0 waren in `Screen` Funktionen zum Setzen und Abfragen des Titels sowie des Tickers vorgesehen. Diese Methoden befinden sich ab Version 2.0 in `Displayable` und sind somit nun auch in Low-Level-Komponenten einsetzbar.

*Screen-Kategorien*

Bei den Unterklassen von `Screen` lassen sich, auch wenn dies in der Vererbungshierarchie nicht explizit zum Ausdruck kommt, wiederum zwei Kategorien unterscheiden:

- `List`, `Alert` und `TextBox` repräsentieren UI-Dialoge, die eine feste Struktur aufweisen und in diesem Rahmen ein recht hohes Maß an Funktionalität erbringen. Dadurch ist die Anwendung dieser Klassen sehr einfach. Andererseits können keine neuen Strukturelemente in diese Komponenten eingefügt werden: `List` ist stets eine Folge von Elementen, `TextBox` bleibt immer ein Eingabefeld.
- `Form` ist flexibler und erfordert einen höheren Programmieraufwand. Diese Klasse erlaubt es, eigene Formulare aus individuellen `Item`-Objekten zusammenzustellen. Wie aus Abbildung 5-3 zu entnehmen ist, stehen dazu acht verschiedene `Item`-Unterklassen zur Wahl. Neue Bedienelemente können als Subklassen von `CustomItem` integriert werden.

Die Details der Programmierschnittstellen dieser Klassen werden in Unterkapitel 5.4 an einigen Beispiel-MIDlets erläutert.

### 5.2.3 Die Klasse `Display`

Die Klasse `Display` bildet das softwaremäßige Abbild des physischen Bildschirms. Beim Starten wird dem MIDlet eine dedizierte `Display`-Instanz zugeordnet. Diese 1:1-Beziehung zwischen MIDlet und `Display` bleibt während des ganzen Programmlaufs bestehen.

*getDisplay()*

Die statische Methode `getDisplay()` gibt das `Display`-Objekt für eine gegebene MIDlet-Instanz zurück. Diese Methode kann während der ganzen Lebensdauer des MIDlets, vom Betreten des Konstruktors bis hin zum Erreichen des *Destroyed*-Zustands, aufgerufen werden. Das Funktionsergebnis ist stets dieselbe `Display`-Instanz.

*setCurrent()*

Ein `Displayable` wird durch einen Aufruf der `Display`-Methode `setCurrent()` sichtbar gemacht.

Zum Anzeigen einer `TextBox` genügen die folgenden Anweisungen:

```
Display display = Display.getDisplay(this); // "this" ist das MIDlet
TextBox textBox = new TextBox("Test", null, 100, TextField.ANY);
display.setCurrent(textBox);           // Displayable anzeigen
```