# JAVA-basierte Benutzeroberflächen für extrem kompakte eingebettete Systeme

*Wolfgang Klingauf[1], Gerrit Telkamp[2], Helge Böhme[1]*

*[1]Abteilung Entwurf integrierter Schaltungen*
*Technische Universität Braunschweig*
*Mühlenpfordtstr. 23, 38106 Braunschweig*
*{klingauf | boehme}@eis.cs.tu-bs.de*
*Tel.: 0531-3913105, Fax: 0531-3915840*

*[2]DOMOLOGIC Home Automation GmbH*
*Rebenring 33, 38106 Braunschweig*
*g.telkamp@domologic.de*
*Tel.: 0531-3804340, Fax: 0531-3804342*

## Kurzfassung

*In diesem Papier wird Vole vorgestellt, ein JAVA-Framework zur effizienten Programmierung grafischer Benutzeroberflächen für sehr kleine und einfache eingebettete Systeme, die über ein monochromes Display mit geringer Auflösung verfügen. Vole wurde für die eingebettete JAVA-Maschine JControl entwickelt, welche als 8-Bit- und als 32-Bit-Implementierung zur Verfügung steht. Trotz seiner geringen Größe (~60 kByte) können mit Vole attraktive und komfortabel zu bedienende eingebettete Anwendungen erstellt werden. Die zu Grunde liegenden Konzepte und Strukturen werden detailliert erläutert und alternativen Technologien gegenübergestellt. Um den Entwicklungsprozess von eingebetteten Applikationen mit Vole und JControl zu demonstrieren, werden zwei Beispiele aus der Praxis angeführt.*

## 1 Motivation

Grafikfähige LC-Displays sind heute in vielen modernen Geräten zu finden – angefangen vom Handy über Autos, Waschmaschinen, HiFi-Anlagen, Barcode-Scanner oder Messgeräte bis hin zur Armbanduhr. In einigen Fällen ersetzen grafische Displays numerische oder textbasierte Anzeigesysteme, weil sich mit ihrer Hilfe auch komplexe Zusammenhänge intuitiver darstellen lassen, z.B. der momentane Zustand einer Anlage. Allgemein fällt es dem menschlichen Auge leichter, bekannte Bilder zu erkennen als Zahlen oder Texte.

Ein weiterer Grund, der für den Einsatz grafikfähiger Anzeigeelemente spricht, ist der deutlich höhere Freiraum für den Entwurf der Inhalte. So lässt sich beispielsweise auch ein firmenspezifisches Design bei der Menüführung oder bei der Darstellung von Gerätezuständen realisieren, evtl. sogar im Zusammenhang mit besonderen Schriftsätzen oder Symbolen (*Corporate Design*). Das Gerät erhält auf diese Weise ein eigenes „Gesicht“ und erscheint dem Anwender zudem „intelligenter“ und hochwertiger.

Allerdings führen grafikfähige Anzeigeelemente auch zu deutlich höheren Anforderungen bei der Software-Entwicklung, denn eine grafische Benutzeroberfläche (GUI, Graphical User Interface) stellt auf Grund ihrer vielfältigen Möglichkeiten eine Herausforderung für jeden Entwickler dar. Hier können moderne objektorientierte Programmiersprachen ihre Vorteile gegenüber den rein prozeduralen Vorgängern (z.B. C oder Assembler) voll ausspielen. Die wichtigsten Vertreter objektorientierter Programmiersprachen sind derzeit C++ und JAVA.

## 2 Java und Benutzeroberflächen

Der Einsatz von JAVA als Programmiersprache für eingebettete Anwendungen wird aus Entwicklersicht häufig mit Skepsis betrachtet. Die aus der PC-Welt bekannten Vorurteile bezüglich Ausführungsgeschwindigkeit, Speicherplatzbedarf und Stabilität veranlassen viele Entwickler dazu, sich lieber weiterhin in den hardwarenahen und wohlbekannten Programmiersprachen C und Assembler auszudrücken. Auch eine gewisse Bequemlichkeit und Furcht vor dem Neuen spielen hierbei sicherlich eine Rolle. Dabei werden die möglichen Einsparungen an Entwicklungszeit und -kosten gerne übersehen. Studien belegen, dass durch JAVA eine Effizienzsteigerung von 50-200% möglich ist [1].

Unsere Erfahrungen zeigen, dass JAVA längst das Potential hat, anspruchsvolle und komplexe Applikationen zu ermöglichen, deren Speicherplatzbedarf kaum größer ist als bei gleichwertigen Lösungen in C oder Assembler.

Neben modernen Konzepten wie automatische Speicherfreigabe (*Garbage Collection*) und dynamisches Nachladen von Programmteilen (*Class Loading*) ist der wichtigste Vorteil von JAVA wohl in der inhärenten Objektorientiertheit zu sehen. Deswegen können komplexe Systeme

wie eine aus vielen Komponenten bestehende grafische Benutzeroberfläche wesentlich transparenter modelliert werden als dies in den rein prozeduralen Programmiersprachen C und Assembler möglich wäre. Durch die Abbildung von programmtechnischen Strukturen in virtuelle *Objekte* und die Einkapselung von gemeinsamen Objekt-Merkmalen in *Klassen* mit wohldefinierten Schnittstellen (Abstraktion) gewinnt das Gesamtsystem massiv an Beherrschbarkeit, Evolutionsfähigkeit und Stabilität. Funktionen und Algorithmen, die in einer Klasse implementiert wurden, können durch Vererbung ohne Neuschreiben von Code an andere Klassen weitergegeben werden (*Code-Reuse*).

Um eine Programmierbibliothek für grafische Benutzeroberflächen wie das in diesem Papier vorgestellte GUI-Framework Vole in JAVA zu realisieren, bietet sich die im Folgenden beschriebene objektorientierte Modellierung an:

- Einzelne Bausteine für grafische Benutzeroberflächen (GUI-Elemente) wie Schaltflächen, Regler, Bilder und Textfelder werden in Form von Objekten implementiert, die von der Klasse *Component* erben. Diese enthält alle ihnen gemeinsamen Eigenschaften und Algorithmen, so dass redundanter Quelltext vermieden wird.
- *Container*-Objekte erfüllen verwaltende Funktionen und fassen mehrere Komponenten zu einer Gruppe zusammen. Durch Mechanismen zur Interaktion zwischen Eingabe (Tastatur, Touchscreen), *Components* und Ausgabe (Display) bilden sie die Schnittstelle zwischen Mensch und Maschine.
- Die Klasse *Frame* bildet den Rahmen einer Applikation. Sie wird vom Programmierer benötigt, um Zugriff auf das GUI-Framework zu bekommen.

In Bild 1 ist die daraus resultierende Objektstruktur dargestellt. Die Umsetzung der oben genannten Konzepte bei Vole wird in Abschnitt 4 ausführlich erörtert. Bekannte Implementierungen ähnlicher Modelle für den PC sind AWT (Abstract Window Toolkit) und JFC (JAVA™ Foundation Classes) von Sun Microsystems sowie MFC (Microsoft Foundation Class Library) von Microsoft [4,5]. Auf Grund ihrer hochgradigen Komplexität kommen diese für den Einsatz auf eingebetteten Systemen allerdings kaum in Frage.

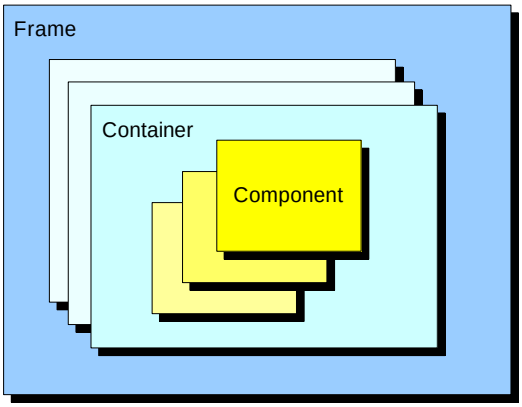



**Bild 1:** Objektstruktur für GUI-Frameworks

## 3 JControl

Standardisierte JAVA-Laufzeitumgebungen stellen hohe Anforderungen an eingebettete Systeme. Ein 32-Bit-Prozessor ist die Mindestvoraussetzung, mit Grafikunterstützung werden zudem leicht mehrere Megabyte an Hauptspeicher benötigt. Unterliegt das Gerät besonderen Anforderungen hinsichtlich Herstellungskosten, Stromverbrauch oder Baugröße, wie im Embedded-Bereich üblich, lässt es sich mit herkömmlichen JAVA-Lösungen nicht mehr realisieren.

JControl ist eine JAVA-basierte Programmierumgebung, die speziell für den Einsatz auf sehr kleinen Systemen mit leistungsarmen Prozessoren und geringer Speicherausstattung konzipiert wurde. Das zentrale Element ist die *JControl Virtual Machine*, ein auf Geschwindigkeit optimierter und sehr kompakter JAVA Bytecode-Interpreter. JControl sieht daneben eine umfangreiche Klassenbibliothek mit besonderen Merkmalen für Anwendungen aus dem Umfeld von Mess-, Steuer- und Regelsystemen vor. Dazu stehen Klassen für die direkte Ansteuerung der Hardware sowie für den Zugriff auf diverse Schnittstellen und Feldbussysteme zur Verfügung. Eine weitere Besonderheit ist die Möglichkeit, mehrere Programmfäden gleichzeitig ausführen zu können (*Multi-Threading*) sowie die Unterstützung weicher Echtzeit-Anforderungen (*Soft-Realtime*). Derzeit ist JControl in zwei unterschiedlichen Varianten für 8-Bit und 32-Bit-Systeme verfügbar [3].

Neben der kompakten virtuellen Maschine und den Klassenbibliotheken umfasst JControl weiterhin leistungsfähige Entwicklungswerkzeuge, z.B. das Projektverwaltungs-Werkzeug *JCManager*. Mit dem JCManager lassen sich Projekte zusammenstellen und in das Embedded System laden. Ein Simulator ermöglicht die exakte Simulation der Geräte schon während der Ent-

wicklungsphase am PC, womit sich die Entwicklungszyklen deutlich verkürzen lassen. Außerdem stehen Editoren für Bilder, Schriftsätze und Melodien zur Verfügung, die auch spezielle gerätespezifische Dateiformate sowie Display-Auflösungen unterstützen.

JControl ist aufgrund der Kompaktheit auch ideal für die Ansteuerung kleiner grafischer Displays geeignet, z.B. mit 128x64 Pixel. Ein Beispiel ist das Produkt *JControl/SmartDisplay* der Firma DOMOLOGIC [2].

# 4 Vole

Ziel bei der Entwicklung von Vole war, die Programmierung von grafisch ansprechenden Benutzeroberflächen für die unterschiedlichsten Systeme und Anwendungen so einfach wie möglich zu machen. Gleichzeitig war ein derart kompaktes und minimalistisches Design gefragt, dass auch kleinste Systeme mit nur wenigen Kilobyte RAM unterstützt werden. So musste ein Kompromiss zwischen Funktionsvielfalt und Ressourcenbedarf gefunden werden, der genügend Spielraum für kundenspezifische Anwendungen bietet und dennoch keine höheren Anforderungen an die Hardware stellt als eine maßgeschneiderte Applikation.

Während das JControl-API nur grundlegende Zugriffsfunktionen auf die Grafik-Hardware wie *setPixel* oder *drawLine* zur Verfügung stellt, sollten mit Vole komplette interaktive Oberflächen mit Menüs, Schaltflächen und Messwert-Visualisierungen schnell und sicher implementiert werden können. Um das GUI-Framework evolutionsfähig und möglichst plattformunabhängig zu gestalten, wurde viel Wert auf hierarchisch organisierte objektorientierte Strukturen gelegt.

## 4.1 Aufbau

Aus Programmierersicht präsentiert sich das Vole-Framework als umfangreiche Bibliothek von fertiggestellten GUI-Elementen. Im Hintergrund arbeiten Funktionen zum selbständigen Zeichnen und zur Koordination der Benutzeroberfläche sowie grundlegende Mechanismen zur Auswertung von Benutzereingaben.

### *4.1.1 Component*

Jede Vole-Komponente erbt von der Klasse *Component*, die grundlegende Funktionen wie *setBounds*, *setGraphics* oder *transferFocus* bereitstellt. Allen GUI-Elementen gemeinsame Instanzparameter wie x- und y-Koordinate oder benutzerdefinierte Schriftart werden nebst entsprechender set-Methoden ebenfalls durch *Component* bereitgestellt. Auf get-Methoden wurde weitestgehend verzichtet, um Speicherplatz zu sparen.

### *4.1.2 Container*

Wie *Component* wurde auch die Klasse *Container* nach dem in Abschnitt 2 beschriebenen Schema realisiert. Bei den enthaltenen Komponenten kann es sich wiederum um *Container* handeln, so dass eine mehrstufige Baumstruktur entsteht. *Container* und *Component* realisieren gemeinsam ein ausgefeiltes Fokus-Management-Konzept, welches keine zusätzlichen Datenstrukturen benötigt und auf der bereits vorhandenen Baumstruktur arbeitet. Das Fokus-Management ist dafür verantwortlich, dass stets nur eine GUI-Komponente auf Benutzereingaben reagiert. Darüber hinaus werden Methoden zur Verfügung gestellt, die selbständig den Fokus von Komponente zu Komponente weiterreichen.

### *4.1.3 Frame*

Ein besonderer *Container* ist die Klasse *Frame*. Sie dient als Ausgangspunkt für alle Vole-basierten Applikationen und enthält eine Nachrichtenschleife, welche die Interaktion zwischen Benutzeroberfläche und Anwender automatisiert.

## 4.2 Beispielprogramm

Wir möchten nun die Programmierung mit Vole an Hand des Beispielprogramms *VoleButton-example* demonstrieren. Es erzeugt eine einfache grafische Benutzeroberfläche mit verschiedenen *Button*-Typen.

Kasten 1 enthält den Quelltext des Beispielprogramms. Durch Erben von der Klasse *Frame* zeichnet sich die Klasse *VoleButtonExample* als Vole-Applikation aus und stellt damit sicher, dass notwendige Initialisierungen automatisch durchgeführt und Benutzereingaben verarbeitet werden.

Im Konstruktor der Klasse *VoleButtonExample* werden mehrere Vole-Komponenten instanziert und den ebenfalls dort generierten *Container*-Instanzen mit Hilfe der Methode *add* hinzugefügt. Die so erzeugte Baumstruktur ist in Bild 2 skizziert. Das gewünschte Layout, also die Anordnung der grafischen Komponenten auf dem Display, wird durch die Angabe absoluter Koordinaten bestimmt. Auf Layout-Manager, wie sie von komplexeren GUI-Frameworks angeboten werden, wurde bei Vole bewusst verzichtet.

```java
/**
 * VoleButtonExample: Example using the three Vole button types
 */
public class VoleButtonExample extends Frame {

  public VoleButtonExample() {
    // create a simple button and add it to the frame
    Button simpleButton = new Button("Press me!", 30, 45, 65, 12);
    this.add(simpleButton);

    // create a Container with three RadioButtons and a Border around it
    Container c1 = new Container();

    RadioButton rb1 = new RadioButton("Radio 1", 5, 8);
    RadioButton rb2 = new RadioButton("Radio 2", 5, 18);
    RadioButton rb3 = new RadioButton("Radio 3", 5, 28);

    // add the RadioButtons to the Container
    c1.add(rb1);
    c1.add(rb2);
    c1.add(rb3);

    // add a Border
    c1.add(new Border("RadioButtons", 0, 0, 60, 40));

    // add the Container to the Frame
    this.add(c1);

    // create a Container with two CheckBoxes and a Border around it
    Container c2 = new Container();
    CheckBox cb1 = new CheckBox("Check 1", 69, 10);
    CheckBox cb2 = new CheckBox("Check 2", 69, 23);

    // add the CheckBoxes to the Container
    c2.add(cb1);
    c2.add(cb2);

    // add a Border
    c2.add(new Border("CheckBoxes", 64, 0, 60, 40));

    // add the second Container to the Frame
    this.add(c2);
  }

  /**
   * Instantiate and run the VoleButtonExample.
   */
  public static void main(String[] args) {
    VoleButtonExample vbe = new VoleButtonExample();
    // make the Frame visible
    vbe.show();
  }
}
```

**Kasten 1:** Quelltext des Java-Programms *VoleButtonExample*

Bild 2 zeigt den Komponentenbaum, der durch das *VoleButtonExample* erzeugt wird. Einen Screenshot des laufenden Programms, erzeugt mit dem JControl-Simulator, zeigt Bild 3.

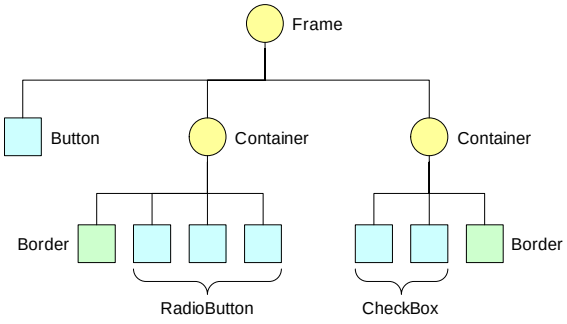



**Bild 2:** Komponentenbaum des Beispielprogramms *VoleButtonExample*


**Bild 3:** Das Beispielprogramm auf dem *JControl/SmartDisplay*

## 4.3 Fokus-Management

Ein großer Vorteil grafischer Benutzeroberflächen ist ihre intuitive Bedienbarkeit. Hierzu ist ein zuverlässig arbeitendes Fokus-Management wichtig, das den Benutzer sicher zwischen den dargestellten Schaltflächen und anderen Interaktions-Elementen navigieren lässt. Bei eingebetteten Systemen kommt erschwerend hinzu, dass meist nur sehr einfache Eingabegeräte wie z.B. ein Wippschalter für die Bedienung verfügbar sind.

Das Fokus-Management von Vole arbeitet auf dem inhärent vorhandenen Komponentenbaum des Anwendungsprogramms und benötigt keinen zusätzlichen Speicherplatz. Da GUI-Elemente zur Laufzeit entfernt und hinzugefügt werden können, müssen verschiedene Sonderfälle beachtet werden.

Die Bewegung des Eingabefokus kann vorwärts und rückwärts erfolgen. Die Art und Weise, wie der Fokus grafisch visualisiert wird, ist den einzelnen Komponenten überlassen. Um den Programmierer von der Hardware unabhängig zu machen, werden Eingabedaten in der Klasse *KeyEvent* auf abstrakte Kommandos abgebildet. Ein Touchpanel unterscheidet sich somit aus Sicht des Programmierers nicht von einem 4-Achsen-Joystick.

## 4.4 Event-Handling

Mechanismen zur Ereignisbehandlung stellen neben dem Fokus-Management das zweite wichtige Standbein des Vole-Frameworks dar. Programmierer von Vole-Applikationen müssen lediglich einen *ActionListener* implementieren, um auf Benutzeraktivitäten zu reagieren. Eine umständliche und plattform*ab*hängige Abfrage der Eingabegeräte durch die Applikation selbst ist nicht nötig.

In Bild 4 ist das Prinzip des Event-Handling dargestellt. Eine Nachrichtenschleife, die als *Thread* im Hintergrund arbeitet und durch die Klasse *Frame* automatisch installiert wird, wertet Tastatur-Ereignisse aus und versendet *KeyEvent*-Nachrichten an den aktuellen Fokus-Inhaber. Dieser erzeugt, falls angebracht, ein passendes *ActionEvent*, das an das Anwendungsprogramm gesendet wird. Im *ActionEvent* enthaltene Informationen geben Aufschluss über die vom Benutzer getätigte Aktion (z.B. Auswahl eines Menüeintrags oder Druck auf eine Schaltfläche).

## 4.5 Komponentenbibliothek

Obgleich für Vole leicht eigene GUI-Elemente erstellt werden können, wird ein GUI-Framework erst durch sein Angebot an sofort verwendbaren Komponenten attraktiv. Vole wurde für den Einsatz in der Mess- und Automatisierungstechnik konzipiert und bietet neben einer Palette von Standardschaltflächen verschiedene Messwert-Visualisierungen und Histogramm-Funktionen. Ferner stehen Komponenten zur Darstellung von Texten und Bildern zur Verfügung, die auch animiert werden können.

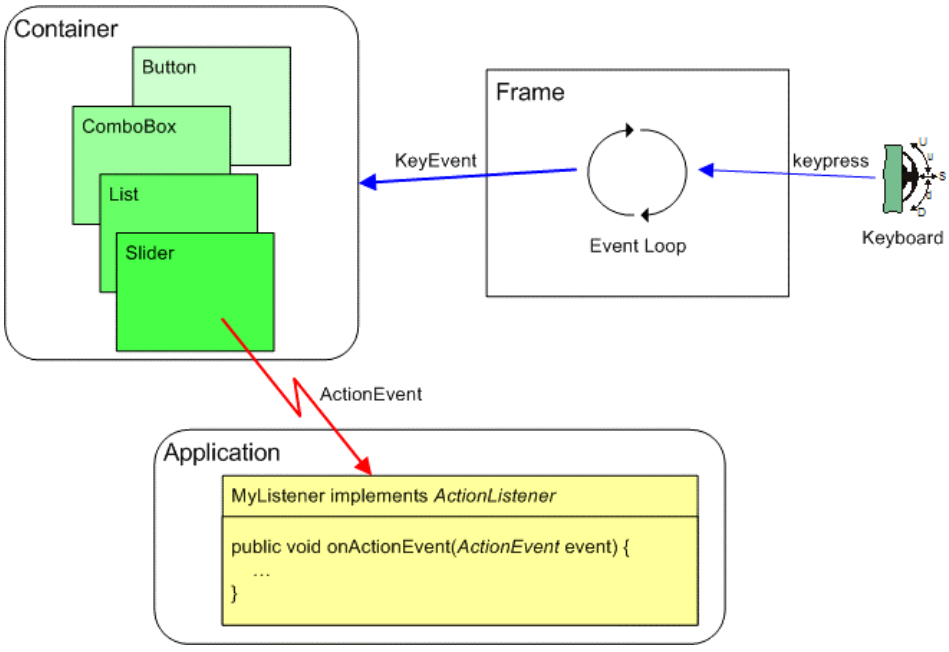



**Bild 4:** Event-Handling bei Vole

Alle GUI-Elemente wurden für die Darstellung auf kleineren Monochrom-Displays entworfen und sind frei skalierbar. Vole bietet ein einheitliches Look-and-Feel, dass sich an grafische PC-Oberflächen anlehnt. Bei Text anzeigenden Elementen kann jede benutzerdefinierte Schriftart eingesetzt werden, die mit dem JControl-Programm *FontEdit* (siehe [2]) erzeugt wurde. Scrollbalken und andere grafische Features werden automatisch berechnet und angezeigt. Das *Animatable*-Interface ermöglicht bewegte Grafiken, wie z.B. einen sich drehenden Ventilator.

Tabelle 1 zeigt eine Auswahl der wichtigsten Vole-Komponenten.

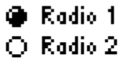
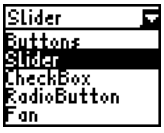
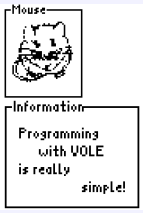
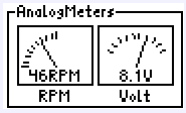
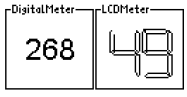
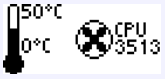
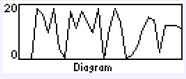
| | |
|---|---|
|  | *Button*, Standard-Schaltfläche. |
|  | *RadioButton*, bei Gruppierung ist automatisch immer nur ein Button aktiv. |
|  | *CheckBox*, die obere Box hat gerade den Eingabefokus. |
|  | *ComboBox*, hier aufgeklappt. Einträge können zur Laufzeit hinzugefügt und entfernt werden. |
|  | *List*, mit automatischer *ScrollBar*. |
|  | *Label*, *Border*, Elemente zur grafischen Gliederung und Anzeige von Bildern und Texten. Die Inhalte können vertikal und horizontal ausgerichtet werden. |
|  | *AnalogMeter*, hier in zweifacher Ausfertigung. Die Skala ist konfigurierbar, verschiedene Beschriftungsoptionen. |
|  | *DigitalMeter*, *LCDMeter*, weitere Messwert-Visualisierungen. |
|  | *Thermometer*, *Fan*, animierte Spezialelemente. |
|  | *AnalogClock*, Analoguhr mit automatischer Aktualisierung, hier von einem *Border* umgeben. |
|  | *Diagram*, zur Darstellung von Messreihen, animierbar. |

**Tabelle 1:** Ausgewählte Vole-Komponenten

## 4.6 Menüs

Neben der Komponentenbibliothek bietet Vole vier verschiedene Klassen von Benutzermenüs. Menüs können wie Komponenten instanziert und dem *Frame* hinzugefügt werden. Das Menü besitzt daraufhin so lange den Eingabefokus, bis ein Menüpunkt vom Anwender ausgewählt wurde.

Zwei textbasierte und zwei Piktogramm-orientierte Menütypen stehen zur Verfügung. Während die *MenuBar* eine Menüzeile mit Texteinträgen am oberen oder unteren Bildschirmrand darstellt, blendet das *TextMenu* eine vertikale Liste von Menüpunkten ein.

Mit Hilfe der Klassen *BigImageMenu* und *MultiImageMenu* können grafische Menüs mit Bild-Ressourcen als Menüpunkte erstellt werden. Das *BigImageMenu* zeigt bildschirmfüllende Grafiken mit automatisch eingeblendeten Navigationspfeilen. Das *MultiImageMenu* ist für mehrere gleichzeitig darzustellende Piktogramme optimiert. In Kombination liefern diese beiden Menütypen eine intuitive, an die Bedienung moderner Handys erinnernde Menüführung. Bild 5 zeigt einen Screenshot des *MultiImageMenus* auf dem äußerst kompakten 8-Bit-System *JControl/Sticker* [2].


**Bild 5:** Das *MultiImageMenu* auf dem JControl/Sticker

## 4.7 Platzbedarf

JControl-Anwendungen bestehen aus JAVA-Bytecode in Form von .class-Dateien, die durch einen JAVA-Compiler (z.B. *javac* von Sun Microsystems) unter Einbindung der JControl-Klassenbibliothek erzeugt wurden. Der Platzbedarf einer Anwendung ergibt sich aus der Summe der Größen der Benutzerklassen zuzüglich den Größen der benötigten Bibliotheksklassen.

Projekt-Management-Tools wie *JCManager* aus der JControl-Entwicklungsumgebung unterstützen den Software-Designer bei der Generierung von Image-Dateien, die das Anwendungspro-

gramm inklusive aller notwendigen Bibliotheksklassen enthalten und auf das Zielgerät geladen werden können. Da meist nur einige Vole-Komponenten verwendet werden, beträgt ihr Speicherplatzbedarf nur wenige Kilobyte. Im ungünstigsten Fall, wenn sämtliche Klassen des GUI-Frameworks durch das Anwendungsprogramm referenziert werden, beläuft sich der zusätzliche Speicherbedarf auf ca. 90 kByte.

Der Arbeitsspeicher der von uns verwendeten JControl-Geräte beträgt 2kB. Folglich ist die Zahl der gleichzeitig darstellbaren GUI-Elemente begrenzt. Unsere Beispielanwendungen (siehe [2]) zeigen, dass Bildschirm füllende Anwendungen (bei 128x64 Pixel) den Speicher nicht voll auslasten. Probleme können aber auftreten, wenn die restliche Anwendungslogik schon sehr ressourcenhungrig ist. In diesem Fall ist es zwar möglich, die GUI-Klassen nach dem Zeichnen sofort wieder zu entladen, dies schmälert die Performance aber deutlich.

### 4.8 Portierbarkeit

Das vorgestellte GUI-Framework ist vollständig in JAVA implementiert. Die Schnittstelle zur unterliegenden JAVA-VM JControl wurde bewusst so schlank aufgebaut, dass eine Portierung auf andere JAVA-Umgebungen jederzeit möglich ist.

Der Zugriff auf Eingabegeräte wird durch die Klasse *Keyboard* gekapselt, welche Methoden zur Abfrage durch das Event-Handling bereitstellt. Für die Ansteuerung des Anzeigegeräts zeichnet die Klasse *Display* verantwortlich. Sie bietet einfache Methoden zum Setzen einzelner Pixel, Zeichnen von Linien und Rechtecken sowie Ausgabe von Text.

Um Vole auf eine andere JAVA-Plattform zu portieren, müssen lediglich diese beiden Klassen an die neue Architektur angepasst werden. Für den *JControl-Simulator* (siehe [2]), der unter JAVA 1.4.2 von Sun Microsystems arbeitet, haben wir dies bereits durchgeführt.

### 4.9 Einschränkungen

Die Zielsetzung, mit weniger als 60kB Speicherplatz auskommen zu wollen und nur 2kB RAM zur Verfügung zu haben, führt zwangsläufig auch zu Einschränkungen seitens der Funktionalität. Vole kann sich daher nicht mit den High-End-Lösungen wie z.B. dem JFC von Sun Microsystems messen, das allerdings auch bis zu 4MB belegt [4].

So fehlt beispielsweise die Unterstützung relativer Koordinaten. Vole-Programmierer müssen stets mit absoluten Bildschirmkoordinaten rechnen und, mangels Layout-Managern, GUI-Elemente von Hand anordnen. Bei Display-Ausmaßen von beispielsweise 128x64 Pixel ist dieser Mehraufwand aber durchaus vertretbar.

Weiterhin bietet Vole im Vergleich zu komplexen GUI-Frameworks nur rudimentäre Clipping-Funktionen zum Überlappen von Komponenten und keine Farb-Unterstützung, aufwendigen Schattierungen, 3D-Effekte oder flexiblere Gestaltungsmöglichkeiten bei Menüs. Auch die Installation alternativer Look-and-Feels und andere weitreichende Individualisierungsoptionen werden durch Vole nicht unterstützt. Allerdings erlaubt die offene Programmierschnittstelle, jederzeit neue GUI-Komponenten und Funktionen hinzuzufügen.

## 5 Praxisbeispiel: Hardware-Monitor

Im Rahmen des Vole-Projekts wurden verschiedene Applikationen für JControl-basierte 8-Bit-Systeme entwickelt, um die Praxistauglichkeit des Frameworks demonstrieren. Sie können auf der JControl-Homepage [2] zusammen mit Vole heruntergeladen werden.

Vole-Hardware-Monitor ist ein Programm zur Überwachung von PC-Systemeigenschaften wie Prozessor- und Gehäusetemperatur, Lüfterdrehzahlen und Spannungsversorgung. Diese Messwerte können mit einem *JControl/SmartDisplay* (siehe [2]) über den IIC-Bus des PC-Mainboards ausgelesen werden. Anforderungen an das Programm waren eine grafisch ansprechende Visualisierung der genannten Messwerte auf dem 128x64 Pixel fassenden Display und eine einfache Benutzerführung.

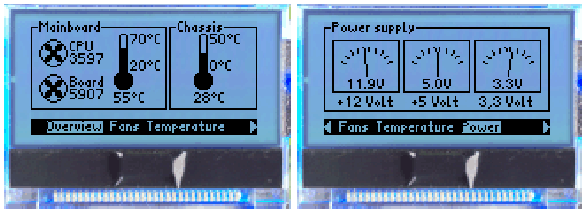



**Bild 6:** Screenshots des Vole-Hardware-Monitors

Bild 6 zeigt zwei Bildschirmseiten des Vole-Hardware-Monitors: Links ist eine Übersicht der wichtigsten Messwerte dargestellt, rechts die Überwachung der Spannunsversorgung. Die einzelnen Seiten können über eine *MenuBar* am unteren Bildschirmrand angewählt werden.

Für die Realisierung der grafischen Oberfläche wurden ausschließlich Vole-Komponenten verwendet. Das Einlesen der Messwerte erfolgt über IIC-Klassen des JControl-APIs. Der Quelltext umfasst knapp 400 Zeilen. Er wurde innerhalb eines Arbeitstages implementiert und ausführlich getestet. Das Bytecode-Image für das Zielsystem, welches neben dem Anwendungsprogramm alle zur Ausführung notwendigen Vole- und JControl-Klassen enthält, belegt nach dem Strippen (entfernen von Debug-Informationen) 39kB. Die Anwendung benutzt Multithreading, um kontinuierlich Messwerte zu erfassen, grafische Komponenten zu animieren und auf Benutzereingaben zu reagieren. Sie kommt mit 2kB Arbeitsspeicher aus und läuft flüssig auf dem *JControl/Smartdisplay*, welches auf einer 8-Bit-JControlVM mit 2 MIPS Verarbeitungsgeschwindigkeit basiert (siehe [6]).

## 6 Andere Embedded-GUI-Lösungen

Der Markt verfügbarer Embedded-GUI-Lösungen ist ausgesprochen vielfältig und unübersichtlich. Sowohl kommerzielle als auch nicht-kommerzielle Software ist hier zu finden, angefangen von rudimentären Grafikbibliotheken, die etwas mehr können als einzelne Pixel zu setzen bis hin zu komplexen Betriebssystemen. Diese bieten grafische Benutzeroberflächen, die sich ganz ähnlich bedienen lassen wie man es als Anwender vom PC gewohnt ist. Damit stellen Sie eine Basis für tragbare „Universal-Systeme“ wie TabletPCs und manche PDAs dar. Die Programmier-Schnittstelle ist zumeist objektorientiert und weitgehend komfortabel. Auf der anderen Seite stellen Lösungen dieser Kategorie aber auch hohe Anforderungen an die Hardware-Ausstattung: Einige Megabyte an Speicher sowie ein schneller Prozessor (100MHz und aufwärts) sind Pflichtprogramm. An kommerziellen Vertretern könnten hier WindowsCE, QT/Embedded, Zinc/VxWorx oder die Reihe freier X-basierter Lösungen für Embedded Linux aufgezählt werden.

Kompakte Lösungen für kleine und mittlere Systeme gibt es ebenso unzählig viele. Diese können natürlich in Puncto Funktionsumfang nicht so ganz mit den „großen“ Embedded GUIs mithalten, sind aber für viele Anwendungen vollkommen ausreichend und lassen sich vor allem auch bei Systemen mit wenig Speicherplatz und geringerer Prozessorleistung einsetzen.

In der Kombination mit oder als Erweiterung für ein bestimmtes Betriebssystem findet man die meisten kommerziellen Embedded-GUI-Lösungen: z.B. PalmOS, SymbianOS oder kwikPeg. Die freien Projekte wiederum bieten fast ausschließlich C-Bibliotheken, die für unterschiedliche Betriebssysteme, Prozessor-Architekturen oder Display-Größen portiert werden können (z.B. Microwindows, PicoGUI, MiniGUI usw.). Dabei haben Bibliotheken einen Vorteil: Sie sind leicht „skalierbar“, d.h. nur die für eine Anwendung benötigten GUI-Bestandteile müssen in das Embedded System geladen werden.

Objektorientiere Strukturen hingegen sind bei kleinen und mittleren Embedded GUIs kaum zu finden, was sicherlich daran liegt, dass die entsprechenden Systeme auch nicht das klassische Umfeld für objektorientierte Software-Entwicklung sind. Allenfalls das *Java Mobile Information Device Profile* (MIDP) ist hier vielleicht eine Ausnahme (das MIDP kommt derzeit bei Handys zum Einsatz und beinhaltet ebenfalls eine Embedded-GUI).

## 7 Fazit und Ausblick

Vole ist eine objektorientierte Embedded-GUI für kleine und mittlere Embedded Systems mit Monochrom-Grafikdisplay. Im Vergleich mit anderen Embedded-GUIs gehört Vole sicherlich zu denjenigen mit den geringsten Hardware-Anforderungen und stellt in diesem Sinne auch einen Kompromiss aus Funktionalität und Komplexität dar.

Im Vergleich zu C-basierten Lösungen bietet Vole dank JAVA die Vorzüge der objektorientierten Programmierung: Flexibel erweiterbare und zu einem großen Teil tatsächlich plattformunabhängige Programmierbibliotheken können die Systementwicklung nicht nur massiv beschleunigen, sondern geben auch mehr Spielraum für kreative Ideen. Von den Speicherplatzanforderungen belegen die Vole-Klassen nur die Hälfte des Speicherplatzes, den die GUI-Klassen des JAVA MIDP belegen; allerdings zielt Vole mit kompakten Messgeräten oder Bedien- und Informationsanzeigen auch auf einen ganz anderen Anwendungsfokus.

Ein und derselbe JAVA-Bytecode kann sowohl auf dem PC als auch auf dem Zielsystem ausgeführt werden. Dies erlaubt echtes Hardware-Software-Codesign, bei dem die Software bereits entwickelt wird, während die Hardware noch gar nicht fertiggestellt ist. Schnelles Prototyping, um dem Kunden frühzeitig verschiedene Ansätze zu demonstrieren, wird mit JAVA realistisch.

Wir haben gezeigt, dass mit JAVA ein Framework für Embedded GUIs erstellt werden kann, das auf 8-Bit-Systemen mit nur 2kB RAM lauffähig ist. Anwendungen, deren Benutzeroberflächen auf Vole basieren, belegen weniger als 60kB Speicherplatz. Eine umfangreiche Komponentenbibliothek unterstützt bei der Komposition attraktiver Benutzeroberflächen. Funktionen zum Einbinden eigener Schriftarten, Bilder und Animationen helfen, dem Produkt ein eigenes Gesicht zu verleihen; spezielle Komponenten erleichtern die Visualisierung von Meßwerten.

Vole eröffnet neue Forschungsbereiche, die wir in aktuellen Projekten bereits bearbeiten. Mit RAPS (Rapid Application Programming Software) entwickeln wir eine offene Entwicklungssoftware, mit der grafische Benutzerschnittstellen auch ohne Programmierkenntnisse erstellt werden können. Schaltflächen, Menüs und andere GUI-Elemente lassen sich per Drag-and-Drop anordnen und mit Aktionen wie z.B. „Ausgang auf 1 setzen“ oder „Signalgeber einschalten“ verknüpfen.

Ein weiterer Arbeitsbereich ist das XUI-Projekt, in dem ein anspruchsvolles GUI-Framework entsteht. Dabei untersuchen wir, wie stark sich Clipping-, Double-Buffering- und Alpha-Blending-Mechanismen auf die Performance auswirken. Austauschbare Look-and-Feels sollen eine weitere Individualisierung ermöglichen. Zielplattformen sind komfortable Bedienterminals mit Touchpanel-Display.

## Literatur


[1] *Flanagan, D.*; Java Power Reference; O’Reilly 1999

[2] JControl-Homepage; *http://www.jcontrol.org*

[3] *Böhme, H., Klingauf, W., Telkamp, G.*; JControl – Rapid Prototyping und Design Reuse mit Java; 11. E.I.S.-Workshop, Erlangen, 2003

[4] *Sun Microsystems*; JAVA™ Foundation Classes; *http://java.sun.com/products/jfc/index.html*

[5] *Microsoft Corp.*; Microsoft Developer Network (MSDN); *http://msdn.microsoft.com*

[6] *DOMOLOGIC Home Automation GmbH*; JControl/Smartdisplay Datasheet; http://www.jcontrol.org/download/datasheets/jcontrol_smartdisplay.pdf