Grafische Benutzeroberflächen mit JControl

Grafische Benutzeroberflächen werden bei eingebetteten Systemen immer wichtiger. Sie gestalten dessen Bedienung multifunktional und trotzdem intuitiv – und sie können mit einem firmenspezifischen Design die Kundenbindung erhöhen. Mittels moderner Programmiertechniken lässt sich der Entwicklungsaufwand für grafische Benutzeroberflächen erheblich reduzieren. Dieser Artikel stellt JControl vor, eine Software-Plattform für sehr kompakte Embedded Systems.

Anforderungen an moderne Geräte

Neben einem günstigen Preis und ausreichendem Funktionsumfang erwarten Kunden vor allem zwei Dinge von einem modernen Gerät: Erstens soll es möglichst intuitiv bedienbar sein – niemand liest gerne lange Bedienungsanleitungen –, zweitens soll es auch ästhetischen Ansprüchen entgegenkommen. Für viele Hersteller bedeutet das, dass sie ihre Produkte mit immer neuen technischen Finessen ausrüsten müssen um am Markt bestehen zu können.

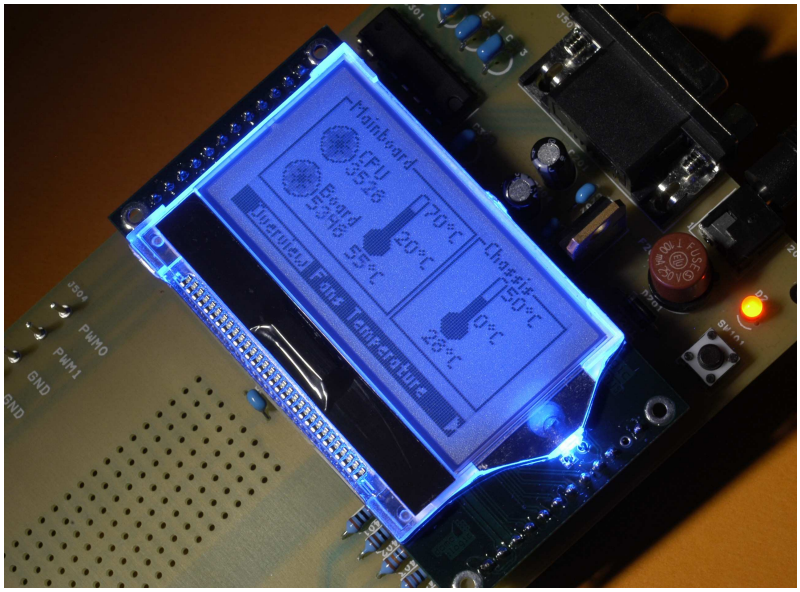


Abbildung 1: Einfache grafische Benutzeroberfläche

Dieser Prozess führt regelmäßig zu einem technologischen Generationswechsel, bei dem alte Technologien durch neue ersetzt werden, deren Fähigkeiten deutlich höher sind als die ihrer Vorgänger. Ein Generationswechsel, der sich seit 5-10 Jahren von den Flagschiffen der Consumer-Produkte auf den Niedrigpreissektor ausweitet, betrifft die Benutzerschnittstelle: An die Stelle von LEDs, Textanzeigen und Tastern mit festgelegten Funktionen (z.B. ein Pause-Taster an einem CD-Player) treten grafische Benutzeroberflächen oder auch *GUIs* (engl. *Graphical User Interface*), realisiert über Matrix-Displays, Touch-Screens oder Cursor-Tasten.

Eine Folge dieser Entwicklung, die sich übrigens analog auch in anderen Hardware-Bereichen (z.B. bei der Vernetzung) abspielt, ist die rasante Steigerung der Softwarekomplexität. War die Steuerung einer LED oder einer 7-Segment-Anzeige eine (fast) triviale Aufgabe, so ist die Ansteuerung eines Matrix-Displays ungleich komplizierter. Zudem bieten Matrix-Displays eine Vielzahl von weiteren Möglichkeiten – man denke beispielsweise an die Darstellung nicht-romanischer Zeichensätze wie Chinesisch oder Kyrillisch. Um diese Möglichkeiten ausnutzen zu können bedarf es entsprechender Software. Hinzu kommt, dass oftmals viele Varianten der gleichen Software gepflegt werden müssen: Neben unterschiedlichen Versionen für diverse Hardware-Konfigurationen müssen Hersteller ihren Produkten häufig ein individuelles *Branding* geben, um beispielsweise das Aussehen der Benutzeroberfläche dem Corporate Design eines Großabnehmers anzupassen. Dass die Software einen immer größeren Anteil am Gesamtaufwand einer modernen Produktentwicklung einnimmt ist offensichtlich.

Steigende Komplexitäten

Damit aufgrund der steigenden Softwarekomplexität die Entwicklungskosten nicht ins Unermessliche steigen, führt der Weg an leistungsfähigeren Programmiersprachen und Entwicklungswerkzeugen kaum noch vorbei. Für eingebettete Systeme bedeutet das, dass einem Generationswechsel bei der Hardware ein Generationswechsel bei der Softwareentwicklung folgen muss. Ein Blick über den Tellerrand der eingebetteten Software hinaus in den Bereich der Desktop- und Server-Applikationen zeigt dazu Lösungsmöglichkeiten auf.

Der letzte große Generationswechsel, der sich bei Desktop- und Server-Applikationen vollzogen hat, war der Wechsel von der *prozeduralen* zur *objektorientierten Programmierung*. Diese ermöglicht dem Software-Entwickler eine intuitivere Perspektive auf seine Applikation, da viele Problemstellungen von Natur aus "*objektorientiert*" sind. Hierzu gehören auch grafische

Benutzeroberflächen, wie wir später noch sehen werden. Darüber hinaus eröffnet die objektorientierte Programmierung den Weg zu neuen Entwicklungsmethoden, die wichtige Hilfestellungen bei der Software-Entwicklung geben können - z.B. *Design Patterns* oder *UML*. Außerdem bieten die meisten objektorientierten Programmiersprachen gut durchdachte Bibliothekskonzepte, die - anders als beispielsweise bei C oder Assembler - Namenskonflikte von vornherein ausschließen. Auch ist die Programmierschnittstelle (*Application Programmer Interface*, *API*) objektorientierter Bibliotheken intuitiver verständlich, wodurch sich deren Wiederverwendbarkeit erhöht (*Design Reuse*). Beispiele für populäre objektorientierte Sprachen sind C++, Java oder C#.

Plattformen für *Managed Code*

Ein noch aktuelleres Thema bei der Programmierung ist die zunehmende Verbreitung von *Managed Code Plattformen*. Dabei werden Programme vom Compiler nicht direkt in ausführbaren "*nativen*" Maschinen-Code übersetzt, sondern in eine Zwischensprache, die von einem Interpreter (der sog. *virtuellen Maschine*) ausgeführt wird. Zwar entsteht durch das Interpretieren für das ausführende System eine zusätzliche Belastung; dem stehen aber eine Reihe von positiven Eigenschaften gegenüber, die sich positiv auf die Entwicklungszeit auswirken:

- *kontrollierte Ausführung*: Der Interpreter hat wesentlich mehr "Intelligenz" als eine in Hardware realisierte CPU und kann daher zur Laufzeit unmittelbar auf Fehlerzustände reagieren. Häufig ist es sehr schwierig, anhand der Wirkungen von Fehlern deren Ursache zu identifizieren, da diese sowohl in ganz anderen Programmabschnitten liegen als auch zeitlich weit zurückliegen kann. Beispielsweise zerstört eine versehentlich überschriebene Speicherzelle wichtige Daten oder vielleicht sogar Programm-Code und die Folgen sind nur schwer zu analysieren. Ein Interpreter kann mittels spezieller Überwachungsmechanismen den Fehler genau dann entdecken, wann er auftritt.
- *automatische Speicherverwaltung*: Der Entwickler wird davon entlastet, in seinem Programm selbsttätig Speicher anzufordern und wieder freizugeben. Stattdessen wird der Speicher implizit durch das Anlegen neuer Objekte oder Arrays alloziert. Ein spezieller Hintergrundprozess, der sog. *Garbage Collector* („Müllsammler“), gibt den Speicher wieder frei, sobald er nicht mehr benötigt wird.
- *Multi Threading*: Die Möglichkeiten, mehrere Programmabschnitte parallel ausführen zu können, sind bei *Managed Code Plattformen* meist sehr stark ausgeprägt, da die Interpreter-Technik *kontextbezogene* Unterbrechungen erleichtert. Insbesondere für die Synchronisierung mehrerer Threads können auf diese Weise komfortable Funktionen realisiert werden.
- *Plattformunabhängigkeit*: Da ein Programm nicht in spezifische Maschinenbefehle übersetzt sondern von einem Interpreter ausgeführt wird, kann es auf jedem System ausgeführt werden, welches eine entsprechende Laufzeitumgebung (virtuelle Maschine und Grundbibliotheken) bereitstellt.

Der Managed-Code-Ansatz ist im wissenschaftlichen Umfeld bereits ein alter Hut und die Entwicklung ist entsprechend weit fortgeschritten. So gibt es verschiedene Techniken, mit denen Ausführungsgeschwindigkeiten erreicht werden, die denen von konventionellen Sprachen wie C oder C++ sehr nahe kommen. Beispiele für *Managed Code Plattformen* sind Java oder .NET („*Dot Net*“).

JControl

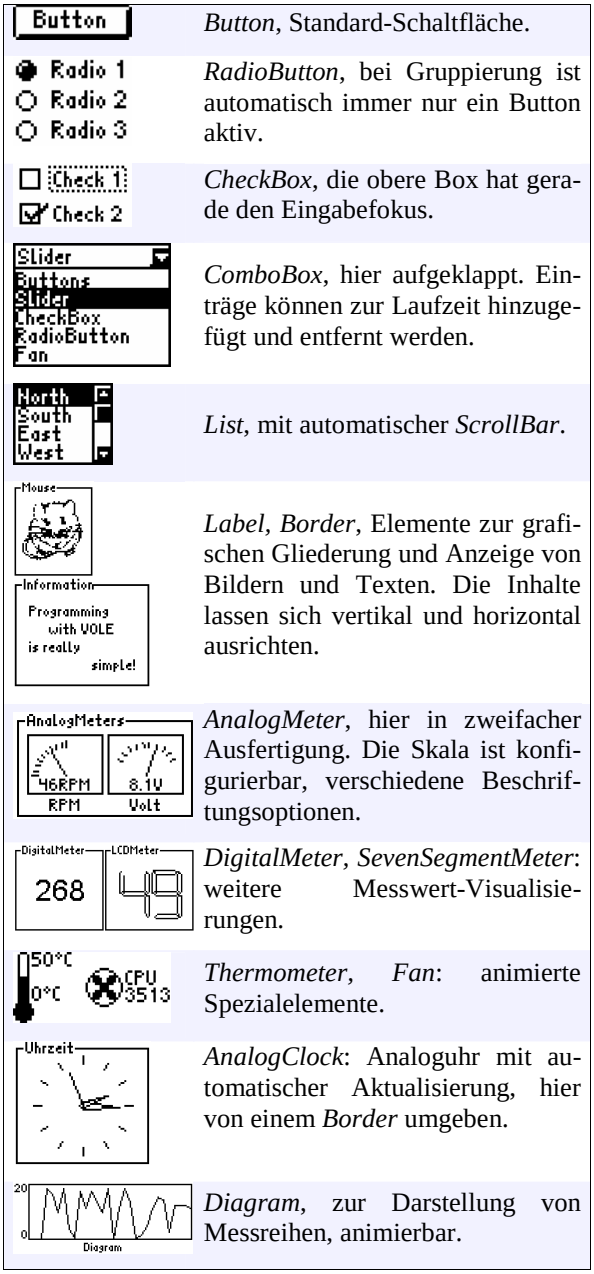
JControl ist eine Java-basierte *Managed Code Plattform* für eingebettete Systeme, die mit sehr geringen Ressourcen auskommt, und sogar auf 8-Bit-Prozessoren mit 2KB RAM einsetzbar ist. Sie besteht aus einer Basisbibliothek zur Ansteuerung von Hardware-Komponenten und einer virtuellen Maschine, die - neben den zum Java-Standard gehörenden Basisfunktionalitäten wie Garbage-Collection und Multi-Threading - spezielle Erweiterungen zur Echtzeitprogrammierung enthält.

Zur komfortablen Entwicklung grafischer Benutzeroberflächen steht die Bibliothek JControl/Vole bereit. Ein Auszug aus der von JControl/Vole angebotenen grafischen Komponenten ist in Abbildung 2 dargestellt.

Software-Anforderungen für grafische Benutzeroberflächen

Bei der Realisierung grafischer Benutzeroberflächen kommen viele Konzepte der objektorientierten Programmierung und des Managed-Code-Ansatzes zur Geltung. Im Desktop-Bereich ist eine objektorientierte Struktur offensichtlich: Die gesamte Oberfläche setzt sich aus einer begrenzten Anzahl von Grundelementen zusammen, die mehrfach verwendet und unabhängig voneinander in Aussehen und Verhalten angepasst werden können. Hierzu zählen z.B. Schaltflächen, Menüs oder Texteingabefelder. Des Weiteren sind viele GUIs alles andere als statisch: Ein Klick auf einen Knopf kann das gesamte Erscheinungsbild der Oberfläche verändern, indem neue Elemente hinzugefügt oder alte entfernt werden. In solchen Situationen übernimmt in einer Managed Code Plattform der Garbage Collector die Verantwortung für das "Entsorgen" nicht mehr benötigter Elemente, eine Aufgabe, die auf "manuellem" Wege nicht immer trivial ist und leicht zu fehlerhaften Speicherzugriffen oder permanenten Speicherverlusten (*Memory Leaks*) führen kann.

Auch den Bibliotheken kommt bei grafischen Benutzeroberflächen eine besondere Bedeutung zu: Mit einer Modifikation der GUI-Bibliothek lässt sich das "Aussehen" einer Anwendung nachträglich noch anpassen. Dadurch ist es z.B. möglich, die Benutzeroberfläche besser auf eine Zielarchitektur (z.B. Farb- oder Monochrom-Display, Touch-Screen oder Cursor-Tasten) abzustimmen. Aber es kann auch nachträglich noch ein *Branding* durch den Austausch von Firmenlogos, Schriftarten und anderer Design-Aspekte vorgenommen werden - und das ohne eine Zeile des Anwendungs-Codes ändern zu müssen.



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 lassen sich vertikal und horizontal ausrichten.

AnalogMeter, hier in zweifacher Ausfertigung. Die Skala ist konfigurierbar, verschiedene Beschriftungsoptionen.

DigitalMeter, *SevenSegmentMeter*: weitere Messwert-Visualisierungen.

Thermometer, *Fan*: animierte Spezialelemente.

AnalogClock: Analoguhr mit automatischer Aktualisierung, hier von einem *Border* umgeben.

Diagram, zur Darstellung von Messreihen, animierbar.

Abbildung 2: Ausgewählte Komponenten der grafischen Benutzeroberfläche Vole

Beispiel: Temperaturanzeige

Am Beispiel der Java-basierten Managed Code Plattform *JControl* illustrieren wir im Folgenden, wie mit sehr geringem Entwicklungsaufwand eine grafische Benutzeroberfläche für ein eingebettetes System realisiert werden kann. Dabei zeigen wir ein kleines Programm, mit dem die Umgebungstemperatur über einen I2C-Bus-Temperaturfühler (TMP75 von *Texas Instruments*) gemessen und grafisch auf einem LC-Display angezeigt wird. Als Hardware kommt das *JControl/SmartDisplay* zum Einsatz, eine kleine in Java programmierbare Anzeigeeinheit mit einer Auflösung von 128x64 Pixel. An die vorgesehene I²C-Bus-Schnittstelle kann der Temperaturfühler direkt angeschlossen werden.

Listing 1 zeigt, mit wie wenig Aufwand sich das Programm realisieren lässt - der gesamte Umfang beträgt weniger als 70 Zeilen Quelltext.

Zunächst werden alle benötigten Bibliothekselemente (*Klassen*)

importiert. Jede Klasse ist in einem Paket (*Package*) mit eindeutigem Namen angeordnet, damit sie leichter gefunden werden kann. So werden Klassen für die Ansteuerung der lokalen Peripherie beispielsweise im Paket `jcontrol.io` zusammengefasst, oder Klassen für die Ansteuerung der am I²C-Bus angeschlossenen Bauteile in einem Paket namens `jcontrol.comm.i2c`. Eindeutige Paketnamen haben noch einen weiteren Vorteil: Auf diese Weise werden Namenskonflikte weitgehend vermieden (*Name Clashes*), wie sie bei der Programmiersprache "C" häufig vorkommen.

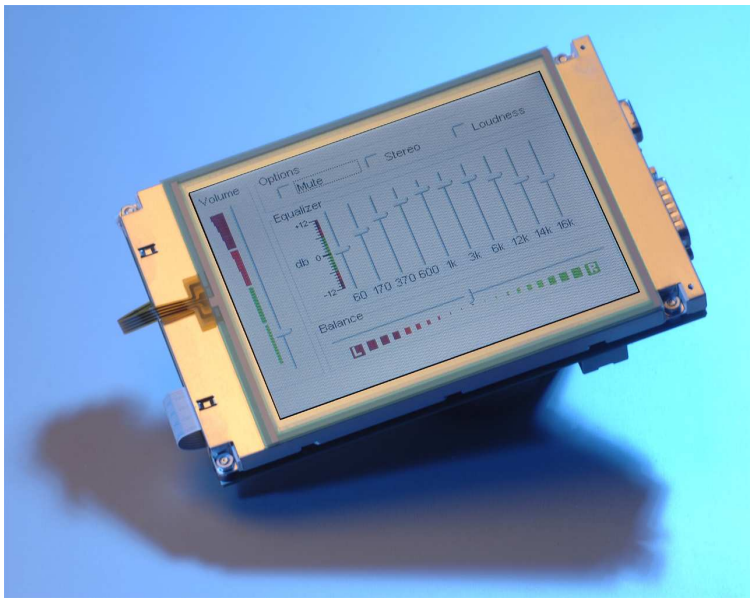


Abbildung 3: JControl-Gerät mit Touch-Panel

Gekapselte Funktionalität

Jede der importierten Klassen stellt eine in sich *gekapselte Funktionalität* dar - angefangen von Klassen für die Ansteuerung der Hardware (Hintergrundbeleuchtung, Temperaturfühler TMP75), über die Klassen zur grafische Darstellung von Messdaten (Diagramm, grafisches Thermometer) bis hin zu Hilfsklassen für die zeitliche Ablaufsteuerungen oder mathematische Funktionen. Der Umfang der verfügbaren Klassen kann sich von Plattform zu Plattform unterscheiden – so machen Netzwerkfunktionen natürlich nur auf solchen Geräten Sinn, die auch einen Netzwerkanschluss bereitstellen.

Doch weiter mit der Beispielanwendung in Listing 1. Nachdem wir alle benötigten Klassen importiert haben, wird die eigentliche Anwendung deklariert. Diese ist eine eigene Klasse mit dem Namen `TemperaturDiagram`. Da es sich um die Hauptklasse einer Anwendung mit grafischer Benutzeroberfläche handelt, werden alle Eigenschaften der „Oberklasse“ `Frame` geerbt (Schlüsselwort "`extends`"). Die Klasse `Frame` gehört zur grafischen Benutzeroberfläche "JControl/Vole" und ist in der Importliste unter `jcontrol.ui.vole.Frame` zu finden. Unsere Hauptklasse `TemperaturDiagram` kann über den Vererbungs-Mechanismus direkt auf die Funktionen der grafischen Benutzeroberfläche zugreifen.

Die `main` Methode

Der Einsprungpunkt der Beispielanwendung ist die Methode `main` (im Fußbereich des Listings). Diese wird beim Einschalten des Systems automatisch gesucht und gestartet. Zunächst schalten wir in der Methode `main` die Display-Hintergrundbeleuchtung ein (über den Befehl `Backlight.setBrighness ...`) und legen als nächstes mit dem Schlüsselwort `new` eine Instanz der Anwendungsklasse `TemperatureDiagram` an.

Die eigentliche Funktionalität der Beispielanwendung steckt im sog. *Konstruktor*, der in der Programmiersprache Java den gleichen Namen wie die Klasse selbst tragen muss (d.h. `TemperatureDiagram`). Der Konstruktor wird immer dann aufgerufen, wenn eine Instanz der Klasse angelegt wird (hier passiert das in der Methode `main`). Als nächstes wird im Konstruktor der Bildschirm aufgebaut. Hierbei lassen sich die Vorteile einer objektorientierten Programmiersprache voll ausreizen: Die aus den Bibliotheken importierten fertigen Komponenten wie z.B. der Rahmen (Klasse `Border`), das grafische Thermometer (Klasse `Thermometer`) und das Temperatur-Diagramm (Klasse `Diagram`) müssen nur noch angelegt, konfiguriert und mit der Methode `add` (geerbt von der Klasse `Frame`) zu einer grafischen Benutzeroberfläche zusammengebaut werden. Die Parameter der einzelnen Methoden beschreiben vor allem die Bildschirmposition und die Größe der Komponenten. Zu guter Letzt werden alle Komponenten mit dem Befehl `setVisible(true)` sichtbar gemacht.

Um einen Temperaturfühler „TMP75“ des Herstellers *Texas Instruments* ansprechen zu können, wird nun eine Instanz der Klasse `TMP75` erzeugt. Diese gehört ebenfalls zu den importierten Klassen und stellt die Methode `getTemp()` bereit. Die Methode misst die Temperatur und liefert den Messwert zurück. Bevor wir jedoch die Temperatur bestimmen können, müssen wir mit dem Befehl `new` eine Instanz der Klasse erzeugen und dabei die I2C-Bus-Adresse angegeben (hier: `0x9e`).

Anzeigen der Temperatur

In einer Endlos-Schleife (eingeleitet durch den Befehl `for (;;)`) wird nun der gemessene Temperaturwert ausgelesen, in

der Variablen `temp` zwischengespeichert und an die grafischen Komponenten weitergegeben. Dabei wird nach jedem Messvorgang eine Pause von 1 Sekunde eingelegt (`ThreadExt.sleep(1000)`) für 1000 Millisekunden). Dieser Programmabschnitt ist in einen sog. `try...catch`-Block eingefasst, in dem ggf. unerwartete Fehler behandelt werden können.

Die hier gezeigte Anwendung kann mit der Entwicklungsumgebung JControl/IDE ausprobiert und simuliert werden, die von der Seite http://www.jcontrol.org kostenlos heruntergeladen werden kann (siehe Screenshot in Abbildung 4).

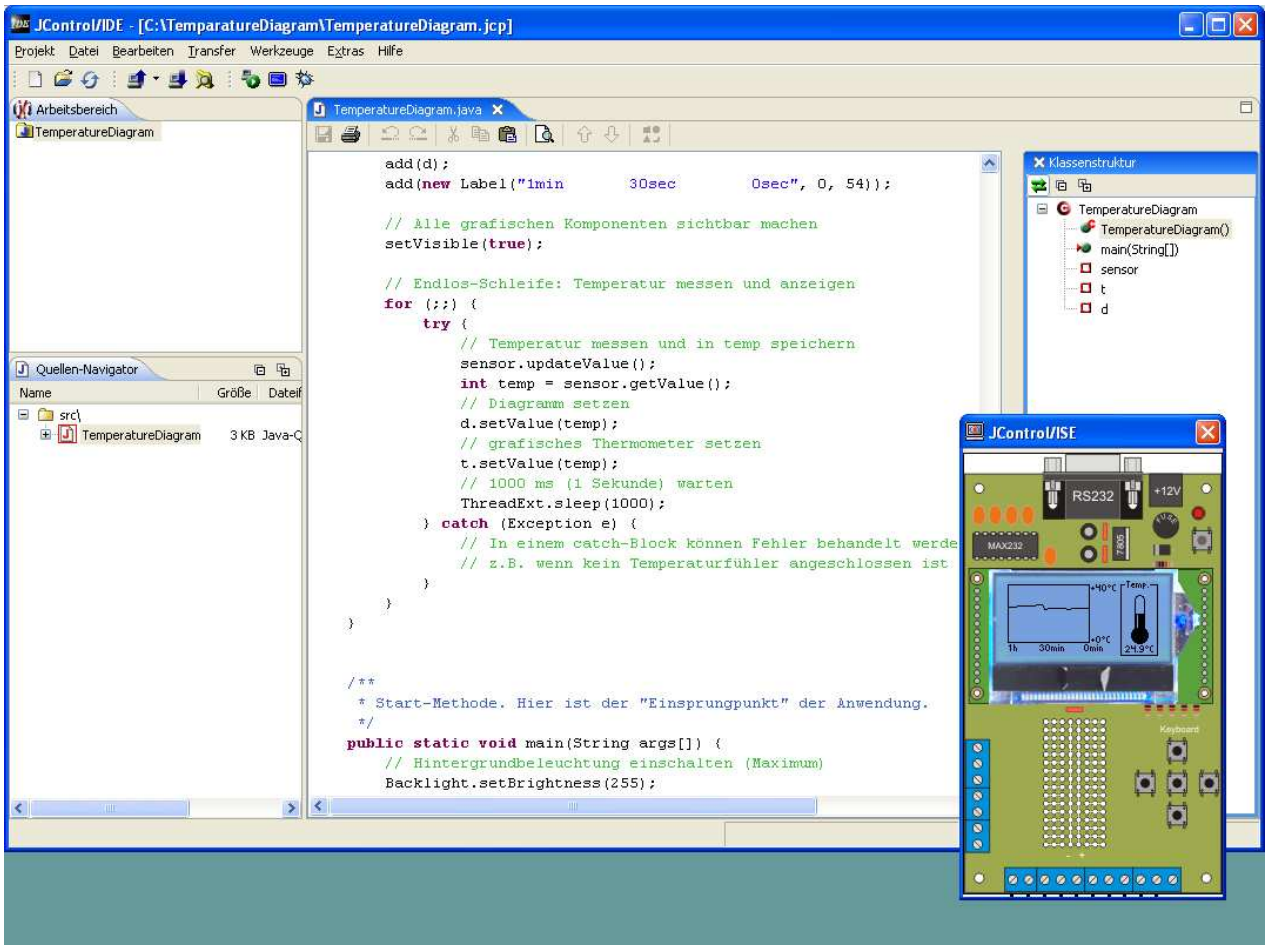


Abbildung 4: Screenshot der integrierten Entwicklungsumgebung "JControl/IDE" mit Simulation des Anwendungsbeispiels

```
// Alle benötigten Bibliothekselemente importieren
import jcontrol.io.Backlight;
import jcontrol.bus.i2c.TMP75;
import jcontrol.ui.vole.Frame;
import jcontrol.ui.vole.Border;
import jcontrol.ui.vole.Label;
import jcontrol.ui.vole.meter.Thermometer;
import jcontrol.ui.vole.graph.Diagram;
import jcontrol.lang.ThreadExt;

// Anwendungsklasse deklarieren
public class TemperatureDiagram extends Frame {

    // Der Konstruktor TemperatureDiagram wird beim Anlegen der Klasse
    // TemperatureDiagram aufgerufen
    public TemperatureDiagram() {

        // Rahmen für grafisches Thermometer anlegen
        add(new Border("Temp.", 96, 0, 32, 64));
        // grafisches Thermometer anlegen und nummerische Anzeige konfigurieren
        Thermometer t = new Thermometer(99, 12, 27, 50, 0, 400);
        t.setNumericDisplay(4, 1, "°C");
        add(t);

        // Temperatur-Diagramm anlegen und beschriften
        Diagram d = new Diagram(0, 3, 94, 50, 0, 400, 60);
        d.setCaption("+0°C", "+40°C", Diagram.ALIGN_RIGHT);
        add(d);
        add(new Label("1min       30sec        0sec", 0, 54));

        // Alle grafischen Komponenten sichtbar machen
        setVisible(true);

        // Temperatur-Fühler vom Typ "TMP75" liegt auf dem I2C-Bus an Adresse 0x9e
        TMP75 sensor = new TMP75(0x9e);

        // Endlos-Schleife: Temperatur messen und anzeigen
        for (;;) {
            try {
                // Temperatur messen und in temp speichern
                int temp = sensor.getTemp();
                // Diagramm setzen
                d.setValue(temp);
                // grafisches Thermometer setzen
                t.setValue(temp);
                // 1000 ms (1 Sekunde) warten
                ThreadExt.sleep(1000);
            } catch (Exception e) {
                // In einem catch-Block können Fehler behandelt werden (gespeichert in "e"),
                // z.B. wenn kein Temperaturfühler angeschlossen ist
            }
        }
    }

    /**
     * Start-Methode. Hier ist der "Einsprungpunkt" der Anwendung.
     */
    public static void main(String args[]) {
        // Hintergrundbeleuchtung einschalten (Maximum)
        Backlight.setBrightness(255);
        // Anwendungsklasse instanziieren und starten
        new TemperatureDiagram();
    }

}
```

Listing 1: Grafische Anzeige eines Temperatur-Messwertes

Literaturhinweise und weitere Informationen zum Thema

- *Bruno Hausherr:* „Messen, Steuern und Regeln mit der Java Control Unit“, Franzis Verlag, 2005, ISBN 3-7723-5357-6
- *David Flanagan:* „Java in a Nutshell, 4. Auflage“, deutsche Übersetzung von Matthias Kalle Dalheimer & Harald Selke, O'Reilly, 2002, ISBN 3-89721-332-x
- JControl-Homepage, http://www.jcontrol.org
- JControl-API-Dokumentation, http://www.jcontrol.org/documentation/index_de.html
- *W. Klingauf, G. Telkamp, H. Böhme:* „JAVA-basierte Benutzeroberflächen für extrem kompakte eingebettete Systeme“, 2003, http://www.jcontrol.org/documentation/files/user_interfaces_with_vole_de.pdf
- „GUI-Programmierung mit JControl/Vole“, http://www.jcontrol.org/tutorials/gui_vole/index_de.html
- *Erich Gamma*, "Design Patterns. Elements of Reusable Object-Oriented Software.", Addison-Wesley Professional, 1997, ISBN 0201633612
- *Craig Larmann*, "Applying UML and Patterns", Prentice Hall PTR, 3. Auflage, 2004, ISBN 0131489062
- *James W. Cooper*, "The Design Patterns Java Companion", Addison-Wesley, 1998,