

## 4 Das Projekt Digitaluhr

Im letzten Kapitel hast du untersucht, was Objekte sind und wie diese implementiert werden. Dabei hast du Datenfelder, Konstruktoren und Methoden kennen gelernt. Weiterhin hast du gesehen, wie Objekte einer Klasse eine gemeinsame Aufgabe erledigen können.

Nun wirst du die Zusammenarbeit von Objekten zweier verschiedener Klassen kennen lernen.

### Bemerkung:

Das Projekt *Digitaluhr* habe ich aus dem Lehrbuch *Objektorientierte Programmierung mit Java* von **M. Kölling** und **D. Barnes** übernommen und sie an meinen Unterricht angepasst.

Dieses Projekt wird eine Digitaluhr sein. Du modellierst eine 24-Stunden-Anzeige, wobei Stunden und Minuten durch einen Doppelpunkt voneinander getrennt sind.

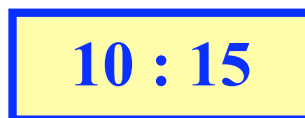


Abbildung 4.1: Anzeige einer Digitaluhr

### 4.1 Erstellung eines Modells

Zuerst wirst du nun das Modell einer Digitaluhr entwerfen.

Die Digitalanzeige kannst du auffassen als zwei getrennte Anzeigen. Die eine Anzeige erstellt ein Ziffern paar für die Stunden, die andere ein Ziffern paar für die Minuten, und beide Anzeigen sind getrennt durch einen Doppelpunkt.

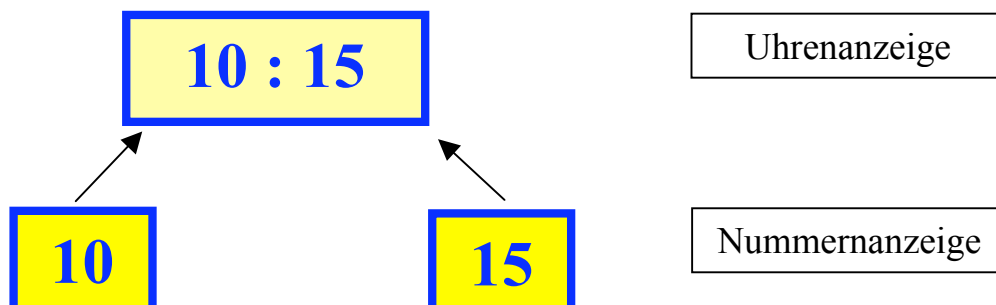


Abbildung 4.2: Modellierung einer Digitalanzeige

Die Nummernanzeige für die Stunde startet bei 0, wird jede Stunde um 1 erhöht, bis sie 23 erreicht und wieder auf 0 zurückspringt.

Die Nummernanzeige für die Minute startet bei 0, wird jede Minute um 1 erhöht, bis sie 59 erreicht und wieder auf 0 zurückspringt.

Du erkennst, dass die Aufgaben dieser beiden Nummeranzeigen identisch sind. Diese Aufgaben werden in der Klasse *Nummernanzeige* erledigt. Dazu benötigst du zwei Datenfelder:

*limit* an dem die Anzeige wieder auf 0 zurück springt,  
*wert* das den aktuellen Anzeigewert beinhaltet.



**Abbildung 4.3:** Objekte *stunden* und *minuten* der Klasse *Nummernanzeige*

```
public class Nummernanzeige
{
    private int limit;
    private int wert;

    // Konstruktor und Methoden hier ausgelassen
}
```

**Abbildung 4.4:** Datenfelder der Klasse *Nummernanzeige*

Den Rest wirst du später kennen lernen.

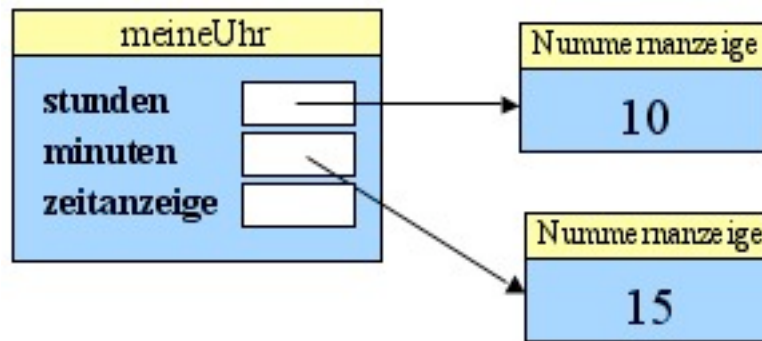
Nun zur Klasse *Uhrenanzeige*. Diese Klasse muss die beiden Nummernanzeigen für die Stunden und für die Minuten enthalten. Du definierst also in der Klasse *Uhrenanzeige* zwei Datenfelder *stunden* und *minuten* vom Typ *Nummernanzeige*. Du benutzt also die Klasse *Nummernanzeige* als den Typ für die Datenfelder *stunden* und *minuten*.

Der Typ eines Datenfelds legt also fest, welche Arten von Werten in dem Datenfeld gespeichert werden können. Wenn der Typ eine Klasse ist, kann das Datenfeld Objekte dieser Klasse enthalten.

```
public class Uhrenanzeige
{
    private Nummernanzeige stunden;
    private Nummernanzeige minuten;

    // Konstruktor und Methoden hier ausgelassen
}
```

}

**Abbildung 4.5:** Datenfelder der Klasse *Uhrenanzeige***Abbildung 4.6:** Objektdiagramm für eine Uhrenanzeige

Beachte, dass du mit zwei Klassen arbeitest, das Objektdiagramm jedoch drei Objekte anzeigt. Du erzeugst nämlich von der Klasse *Nummernanzeige* zwei Objekte.

Um nun diese Anwendung zu starten, wird ein Objekt *meineUhr* der Klasse *Uhrenanzeige* erzeugt. Dieses erzeugt automatisch zwei Objekte *stunden* und *minuten* der Klasse *Nummernanzeige*.

#### Bemerkung:

Wenn eine Variable ein Objekt enthält, dann ist das Objekt nicht direkt in der Variablen selbst abgelegt, sondern die Variable enthält lediglich eine *Referenz auf ein Objekt* (eine *Objektreferenz*). In Abbildung 4.6 ist im Objektdiagramm die Variable durch ein weißes Feld symbolisiert und die Referenz durch einen Pfeil. Das Objekt, das referenziert wird, ist außerhalb des die Referenz haltenden Objekts gespeichert. Die Objekte sind durch die Objektreferenz miteinander verbunden.

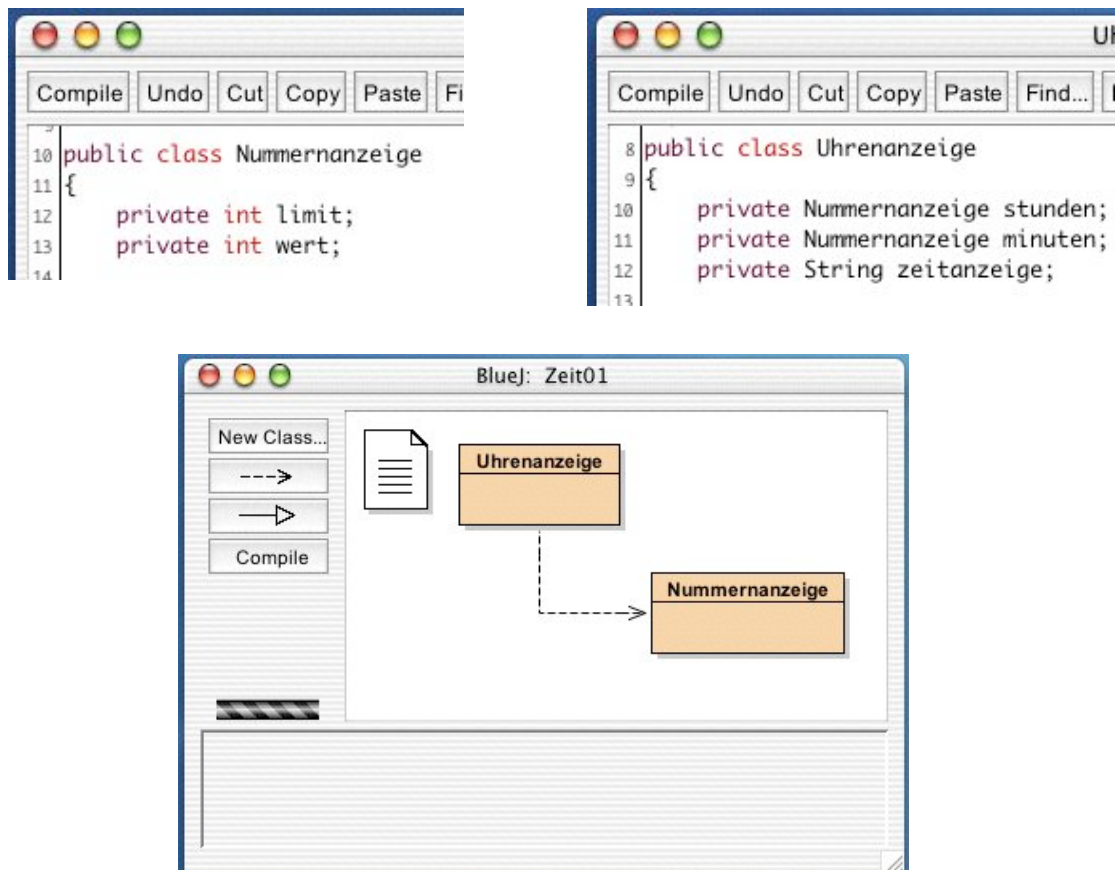


Abbildung 4.7: Klassen *Uhrenanzeige* und *Nummernanzeige* in BlueJ

Das Modell der Digitaluhr ist nun entworfen. Nun wirst du dich der Implementierung dieses Modells widmen.

## 4.2 Die Klasse *Nummernanzeige*

Was muss die Klasse *Nummernanzeige* alles können?

1. Sie muss den aktuellen Anzeigewert liefern: *gibWert()*
2. Sie muss den Anzeigewert um 1 vergrößern. Wenn das *limit* erreicht ist, muss dieser Wert auf 0 zurückspringen: *erhoehen()*

### Bemerkungen:

1. Die Klasse *Nummernanzeige* wird später noch weitere Methoden besitzen, aber diese sind vorerst die wichtigsten.
2. In der folgenden Implementierung verzichte ich auf die Kommentare. Dadurch werden die Abbildungen kleiner.

```
public class Nummernanzeige
```

```
{
    private int limit;
    private int wert;

    public Nummernanzeige(int limitH)
    {
        limit = limitH;
        wert = 0;
    }

    public int gibWert()
    {
        return wert;
    }

    public void erhoehen()
    {
        wert = (wert + 1) % limit;
    }
}
```

**Abbildung 4.8:** Quelltext der Klasse *Nummernanzeige*

#### Übung 4.1:

Erzeuge in **BlueJ** ein Projekt *Zeit01a*. Implementiere die Klasse *Nummernanzeige*. Schreibe zusätzlich ausreichende Kommentare. Untersuche ihre Methoden mit Hilfe des Inspektors. Formuliere schriftlich, was der Operator `%` berechnet. (Setze dazu den Parameter *anzeigeLimit* probeweise auf den Wert 5.)

Die Anzeige der Digitaluhr soll eine Zeichenkette aus zwei Zeichen darstellen, z.B. 03 : 05. Mit den obigen Methoden wirst du bloß die Darstellung 3 : 5 erhalten. Deswegen wird die Methode *gibAnzeigewert()* implementiert. Diese liefert den aktuellen Wert als Zeichenkette und fügt außerdem eine führende Null ein, wenn der Wert unter 10 liegt.

```
public String gibAnzeigewert()
{
    if(wert < 10)
        return "0" + wert;
    else
        return "" + wert;
}
```

**Abbildung 4.9:** Quelltext der Methode *gibAnzeigewert()* in der Klasse *Nummernanzeige*

#### Übung 4.2:

Füge die Methode *gibAnzeigewert()* der Klasse *Nummernanzeige* hinzu. Erzeuge ein Objekt und teste seine Methoden.

Du hast nun also eine Klasse erstellt, die zweiziffrige Nummern anzeigt. Ein Objekt mit *limit* = 24 stellt die Stundenanzeige, ein Objekt mit *limit* = 60 stellt die Minutenanzeige dar.

### 4.3 Die Klasse *Uhrenanzeige*

Was muss die Klasse *Uhrenanzeige* alles können?

1. Sie muss die Zeitanzeige als String darstellen: *aktualisiereAnzeige()*
2. Sie muss diesen String als Wert zurückliefern: *gibUhrzeit()*

Auch hier werden später noch weitere Methoden folgen.

```
public class Uhrenanzeige
{
    private Nummernanzeige stunden;
    private Nummernanzeige minuten;
    private String zeitanzeige;

    public Uhrenanzeige()
    {
        stunden = new Nummernanzeige(24);
        minuten = new Nummernanzeige(60);
        aktualisiereAnzeige();
    }

    public String gibUhrzeit()
    {
        return zeitanzeige;
    }

    private void aktualisiereAnzeige()
    {
        zeitanzeige = stunden.gibAnzeigewert() + ":"
            + minuten.gibAnzeigewert();
    }
}
```

**Abbildung 4.10:** Quelltext der Klasse *Uhrenanzeige*

Im Konstruktor werden die beiden Objekte *stunden* und *minuten* vom Typ *Nummernanzeige* erzeugt. Mit dem Operator *new* wird also ein neues Objekt der angegebenen Klasse *Nummernanzeige* erzeugt und der Konstruktor dieser Klasse ausgeführt. Der Konstruktor der Klasse *Nummernanzeige*

```
public Nummernanzeige(int limitH)
```

besitzt den *formalen Parameter* `int limitH`. Deshalb müssen die *new-Anweisungen*, die diesen Konstruktor aufrufen, jeweils einen *aktuellen Parameter* vom Typ `int` übergeben:

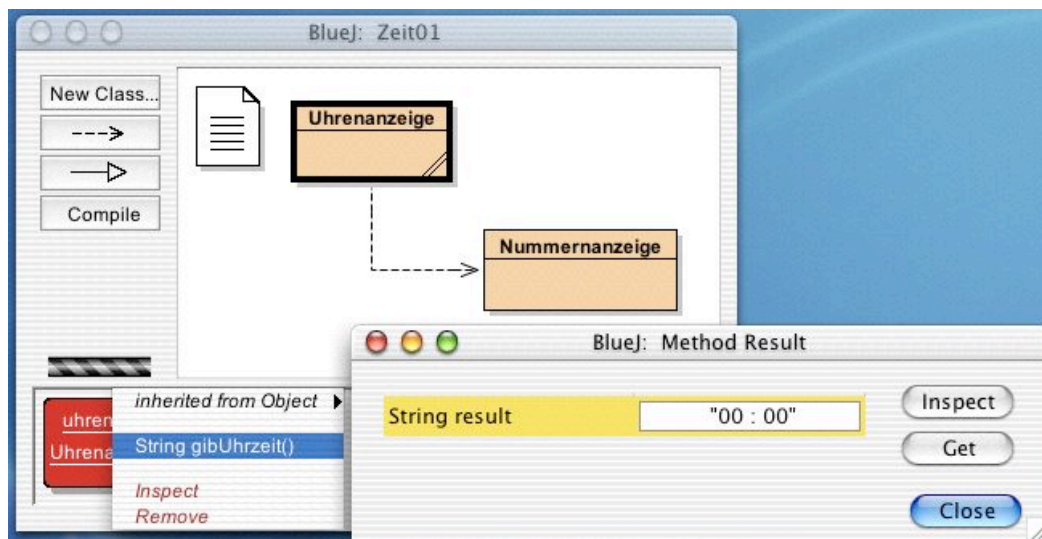
```
stunden = new Nummernanzeige(24);  
minuten = new Nummernanzeige(60);
```

Wenn also ein Objekt der Klasse *Uhrenanzeige* erzeugt wird, so wird der Konstruktor der Klasse *Uhrenanzeige* ausgeführt und dieser erzeugt automatisch zwei Objekte der Klasse *Nummernanzeige*.

Damit wird die in Abbildung 4.6 dargestellte Situation erreicht.

### Übung 4.3:

Implementiere im Projekt *Zeit01a* die Klasse *Uhrenanzeige* und teste ihre Methode *gibUhrzeit()*. Momentan liefert sie nur als Ergebnis `00:00`.



**Abbildung 4.11:** Klasse *Uhrenanzeige* mit ihrer Methode *gibUhrzeit()*

Nun musst du noch ein Taktsignal simulieren. In der realen Welt sollte dieses jede Minute von einem Quarz erzeugt werden. In deinem Modell wirst du dich selbst bemühen müssen.

```
public void gibTaktsignal()  
{  
    minuten.erhoehen();  
    if(minuten.gibWert() == 0) {  
        stunden.erhoehen();  
    }  
}
```

```

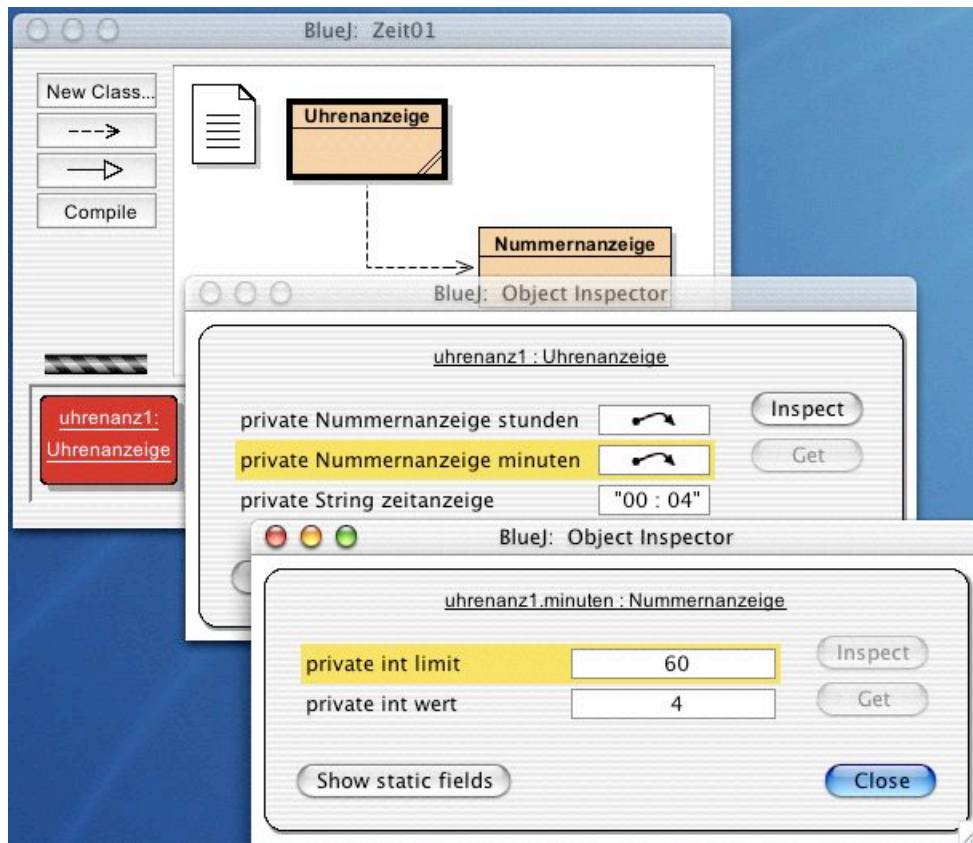
    aktualisiereAnzeige();
}

```

**Abbildung 4.12:** Quelltext der Methode *gibTaktsignal()* in der Klasse *Uhrenanzeige*

#### Übung 4.4:

Füge die Methode *gibTaktsignal()* der Klasse *Uhrenanzeige* hinzu und teste ihre Methoden „live“ mit Hilfe des Inspektors.



**Abbildung 4.13:** Untersuchung der Klasse *Uhrenanzeige* mit Hilfe des Inspektors

#### 4.4 Setzen der Uhrzeit

Noch ist die Digitaluhr nicht komplett, du kannst die Uhrzeit nicht einstellen.

Dazu musst du in der Klasse *Nummernanzeige* direkt auf das Datenfeld *wert* mit einer verändernden Methode zugreifen. In einer Sicherheitsabfrage werden nur sinnvolle Werte zugelassen.

```

public void setzeWert(int werth)
{
    if((werth >= 0) && (werth < limit))
        wert = werth;
}

```

**Abbildung 4.14:** Die Methode *setzeWert()* in der Klasse *Nummernanzeige*.

### Übung 4.5:

Erstelle in **BlueJ** ein neues Projekt *Zeit01b*. Dazu hast du zwei Möglichkeiten:

1. Im Menüpunkt *Project > Save as...* das ursprüngliche Projekt unter dem neuen Namen *Zeit01b* abspeichern.
2. Ein neues Projekt *Zeit01b* erstellen und mit dem Menüpunkt **Edit > Add Class from File ...** die Klassen aus dem ursprünglichen Projekt *Zeit01a* übernehmen.

Implementiere in der Klasse *Nummernanzeige* die Methode *setzeWert()* und teste diese Klasse *Nummernanzeige* mit Hilfe des Inspektors.

Nun musst du noch in der Klasse *Uhrenanzeige* eine Methode *setzeUhrzeit()* implementieren. Diese Methode erhält als formale Parameter *stunde* und *minute*. Dann ruft sie die Methode *setzeWert()* der jeweiligen Objekte *stunden* und *minuten* aus der Klasse *Nummernanzeige* auf und verändert deren *wert*.

```
public void setzeUhrzeit(int stunde, int minute)
{
    stunden.setzeWert(stunde);
    minuten.setzeWert(minute);
    aktualisiereAnzeige();
}
```

**Abbildung 4.15:** Die Methode *setzeUhrzeit()* in der Klasse *Uhrenanzeige*

### Übung 4.6:

Implementiere in der Klasse *Uhrenanzeige* die Methode *setzeUhrzeit()* und teste sie mit Hilfe des Inspektors.

Bereits beim Erzeugen eines Objekts der Klasse *Uhrenanzeige* kann dieses auf eine bestimmte Uhrzeit eingestellt sein. Dieses „automatische Einstellen“ wird mit Hilfe eines weiteren Konstruktors durchgeführt.

```
public Uhrenanzeige(int stunde, int minute)
{
    stunden = new Nummernanzeige(24);
    minuten = new Nummernanzeige(60);
    setzeUhrzeit(stunde, minute);
}
```

**Abbildung 4.16:** Ein weiterer Konstruktor in der Klasse *Uhrenanzeige*

## Übung 4.7:

Implementiere in der Klasse *Uhrenanzeige* den neuen Konstruktor und teste ihn mit Hilfe des Inspektors. (vgl. Abbildung 4.17)

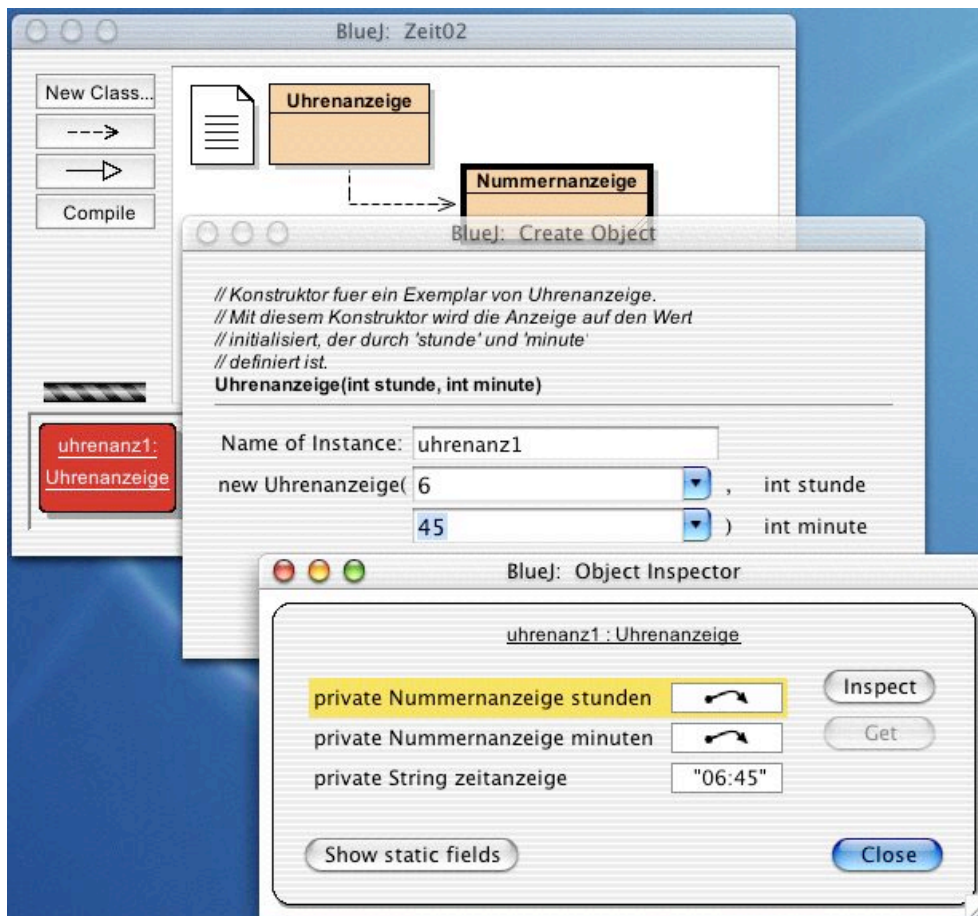


Abbildung 4.17: Konstruktor setzt die Uhrzeit

## Übung 4.8:

Vervollständige nun das Projekt *Zeit01b*. Verwende Kommentare. Teste es ausgiebig!

Was passiert, wenn du negative Zahlen verwendest?

## Übung 4.9

Programmiere eine Digitaluhr mit Stunden- Minuten- und Sekundenanzeige!

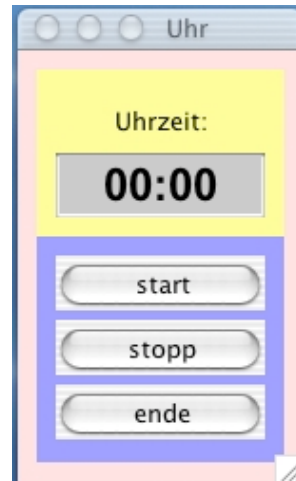
## Übung 4.10: (Für Experten)

Programmiere eine Digitaluhr mit Timer oder Alarm!

## 4.5 Grafische Benutzeroberfläche

Nachdem die Uhr nun funktioniert, kannst du dich der grafischen Benutzeroberfläche widmen. Die Abbildung 4.18 zeigt eine einfach gestaltete grafische Benutzeroberfläche. Diese wird in Java-Swing gestaltet.

Die Gestaltungsmöglichkeiten mit Swing werden im *Handbuch der Java-Programmierung* von Guido Krüger, im *The JFC Swing Tutorial, Second Edition* von Kathy Walrath, u. a oder in ähnlichen Büchern beschrieben.



**Abbildung 4.18:** Einfache grafische Benutzeroberfläche der Digitaluhr

In Swing werden die Fensterkomponenten in einer Art Container, so genannte Panels, gepackt. In der Abbildung 4.18 bezeichne ich den Teil des gesamten Fensters, in dem gezeichnet werden kann (also ohne der Menüleiste) als Hauptpanel *mainP*. Dieser Teil ist der hellrote Bereich.

Dieses Hauptpanel ist wiederum unterteilt in das

- gelbe Zeitpanel *zeitP* mit einem Label und einem Textfeld
- blaue Buttonpanel *buttonP* mit den drei Buttons *start*, *stopp* und *ende*

Jeder dieser Button bekommt einen *ActionListener*, der das Anklicken überwacht und dann die entsprechenden Aktionen auslöst. Die Aktionen von z.B. *start* werden in der Methode *startAction()* implementiert, somit kann die Uhr starten.

### Übung 4.11:

Hole vom Schulserver das Projekt *ZeitGUI* und speichere dieses als Projekt *Zeit02*. Studiere den Quelltext der Klasse *Ansicht*. Suche die Komponenten *zeitTF* und *start*.

In vielen grafischen Oberflächen wird die Anordnung der Komponenten durch Angaben absoluter Koordinaten vorgenommen. Da jedoch Java-Programme auf unterschiedlichen Plattformen laufen sollen, ist eine solche Vorgehensweise nicht sinnvoll. Zur Anordnung der Komponenten verwendet man in Java verschiedene so genannte *Layoutmanager*. Diese sorgen dafür, dass die Benutzeroberfläche auf allen Plattformen das gleiche Aussehen hat (weitestgehend).

Ich verwende in diesem Beispiel den *GridLayout-Manager* und den *BoxLayout-Manager*. Im *GridLayout* werden die Komponenten innerhalb eines rechteckigen Gitters angeordnet. Die Parameter beim Aufruf des Konstruktors bestimmen die vertikale und die horizontale Anzahl der Elemente. Im *BoxLayout* werden die Komponenten standardmäßig untereinander in der benötigten Größe angeordnet. Weitere Einzelheiten werden in den oben genannten Büchern beschrieben.

Die folgenden Übungen werden dich schrittweise vertraut machen mit der Programmierung von grafischen Oberflächen in Java. Zum Schluss wirst du eine Oberfläche programmiert haben wie in Abbildung 4.18 dargestellt. Verwende so oft wie möglich das Verfahren „Copy and Paste“!

#### Übung 4.12:

Ändere in der Methode *setTitle()* das Argument „Test“ um in „Uhr“.  
Ändere in der Methode *setBounds()* die ursprünglichen Werte (50, 50, 150, 230) in (300, 200, 400, 250) bzw. in (50, 50, 600, 400). Formuliere schriftlich, welche Veränderungen dadurch stattfinden. Setze in der Methode *setBounds()* wieder die ursprünglichen Werte ein.

#### Übung 4.13:

Das *zeitP* soll nun den zusätzlichen Text „Uhrzeit:“ enthalten.. Bezeichne das neue Label als *zeitL* und füge es analog dem Textfeld *zeitTF* dem Zeitpanel hinzu. Vergiss nicht den zusätzlichen Eintrag im Variablenkatalog.

#### Übung 4.14:

Ändere den Text des Buttons in „start“.

#### Übung 4.15:

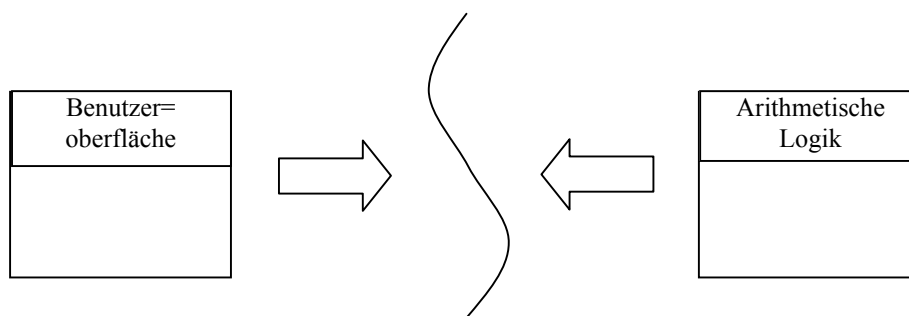
Füge nun die beiden weiteren Button *stopp* und *ende* hinzu. Vergiss nicht die entsprechenden *ActionListener*-Methoden, auch wenn diese noch leer sind.

Verändere probeweise die letzten beiden Werte 5 in der Parameterliste der Anweisung *new GridLayout(0, 1, 5, 5)*; in die Werte 20. Formuliere die Bedeutung dieser beiden Parameter.

Nach Abschluss dieser Übungen sollte das Fenster des Projekts *Zeit02* der Abbildung 4.18 gleichen.

#### 4.6 Verbindung zwischen Uhrenanzeige und Ansicht

Das Digitaluhr-Projekt ist also in zwei größere Teile zerlegt worden: Der eine Teil (die in Kapitel 4.5 erstellte Benutzeroberfläche) ermöglicht dem Benutzer, die Uhr mit Hilfe der Buttons zu steuern, der andere Teil (die in Kapitel 4.3 erstellte Uhr *Zeit01a*) implementiert die arithmetische Logik für die Berechnungen der Uhrzeit.



**Abbildung 4.19:** Festlegung der Schnittstelle

Die Abbildung 4.19 zeigt, dass für beide Teile klar definiert sein muss, wie sie den jeweils anderen Teil benutzen können – die Schnittstelle zwischen ihnen muss definiert werden. In größeren Projekten muss es also klare Richtlinien geben, welches Entwicklungsteam für welche Aufgaben zuständig ist und wie die verschiedenen Teile schließlich in der Gesamtanwendung zusammenspielen sollen. Es wird also notwendig sein, die Schnittstellen festzulegen, bevor an der Implementierung der Teile gearbeitet werden kann.

In dieser Einführung des Projekts habe ich aus didaktischen Gründen den umgekehrten Weg beschritten.

Damit die Digitaluhr auch mit Hilfe der Buttons bedient werden kann, musst du in der Klasse *Ansicht* noch einen Timer einführen. Im Konstruktor erzeugst du den Timer *timer*.

```
timer = new Timer(1000, new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        timerAn ();
    }
});
```

**Abbildung 4.20:** Erzeugung des Timers *timer* im Konstruktor der Klasse *Ansicht*

Alle 1000 ms ruft dieser Timer *timer* die Methode *timerOnT()* auf, deren Anweisungen dann durchgeführt werden.

**Bemerkung:**

Hier wird ein Timer verwendet, der alle 1000 ms eine Aktion durchführt. Das Projekt *Zeit01a/b* verwendet jedoch Stunden und Minuten. Hier machst du also noch einen Fehler, der jedoch in Übung 4.19 beseitigt wird.

Die Methode *timerOnT()* wird vorläufig noch leer gelassen.

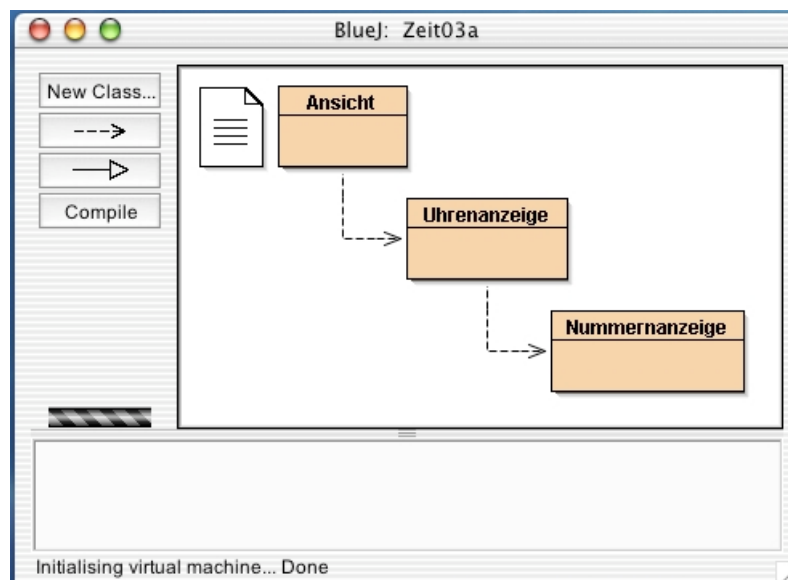
```
private void timerAn()  
{  
    //leer  
}
```

**Abbildung 4.21:** Die Methode *timerOnT()* in der Klasse *Ansicht*

**Übung 4.16:**

Erstelle in **BlueJ** ein neues Projekt *Zeit03a*. Mit Hilfe des Menüpunkts *Edit > Add Class from File ...* übernehme die Klasse *Ansicht* aus dem Projekt *Zeit02* und die Klassen *Uhrenanzeige* und *Nummernanzeige* vom Projekt *Zeit01a*.

Nun implementiere den Timer in der Klasse *Ansicht*.



**Abbildung 4.22:** Klassendiagramm von *Zeit03a*

In der Klasse *Ansicht* existieren nun vier leere Aktion-Methoden für die drei Buttons und für den Timer. In diese Methoden legst du nun die entsprechenden Aktionen.

```
private void startAction()
{
    timer.start();
}

private void stoppAction()
{
    timer.stop();
}

private void endeAction()
{
    timer.stop();
    System.exit(0);
}

private void timerAn()
{
    uhr.gibTaktsignal();
    zeitTF.setText(uhr.gibUhrzeit());
}
```

**Abbildung 4.23:** Die vier Aktions-Methoden

Der Button *start* startet den Timer, entsprechend stoppt der Button *stoppB* den Timer. Um die Anwendung mit *endeB* zu beenden, muss der Timer zuerst gestoppt und dann die Anwendung verlassen werden.

Alle 1000 ms wird vom Timer die Methode *timerOnT()* aufgerufen. Diese gibt der Uhr das Taktsignal und schreibt die aktuelle Zeit dann in das Textfeld *zeitTF*:

#### Bemerkung:

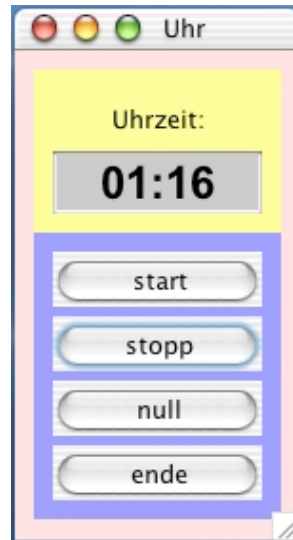
Der Timer wird als Thread behandelt. Um diesen Timer zu starten bzw. anzuhalten, müssen deswegen die spezifischen Thread-Methoden *start()* bzw. *stop()* verwendet werden.

#### Übung 4.17:

Implementiere im Projekt *Zeit03a* die vier Aktions-Methoden aus Abbildung 4.23 in der Klasse *Ansicht*.

#### Übung 4.18:

Speichere das Projekt *Zeit03a* unter dem Namen *Zeit03b* und implementiere zusätzlich eine Methode, die die Uhr auf „00:00“ zurückstellt. Verwende dazu das Projekt *Zeit01b*.



**Abbildung 4.24:** Projekt *Zeit03b* kann die Uhr auf Null zurücksetzen.

#### Übung 4.19:

Implementiere nun ein Projekt *Zeit03c*, das die Stunden, Minuten und Sekunden zählt. Beachte hierzu die Bemerkung und Übung 4.9. Hast du schon überprüft, wie genau diese Uhr geht?

### 4.7 Java ohne BlueJ

Du hast *BlueJ* verwendet, um eine Java-Anwendung zu entwickeln und auszuführen. Dafür gibt es auch einen guten Grund: *BlueJ* bietet einige Werkzeuge, die typische Aufgaben bei der Entwicklung erleichtern. Insbesondere ermöglicht *BlueJ* die direkte Ausführung einzelner Methoden von Objekten – das ist nützlich, um schnell ein neues Stück Quelltext zu testen.

Anwendungen, die an Benutzer ausgeliefert werden, werden ohne *BlueJ* gestartet. Man bezeichnet solche Anwendungen als *stand-alone-Applikation*. Sie haben üblicherweise einen Einstiegspunkt, an dem die Ausführung beginnt, wenn der Benutzer die Anwendung mit einem Doppelklick auf ein Icon startet. Das Java-System sucht dann nach einer Methode mit der folgenden Signatur:

```
public static void main(String[] args)
```

Der Name *main* wurde willkürlich von den Java-Entwicklern gewählt, steht jedoch fest: Die Methode muss so heißen!

Um aus dem Projekt *Digitaluhr* eine ausführbare Anwendung zu machen, benötigst du also

1. eine Klasse, die die Methode *main()* enthält,
2. einen Mechanismus, der alle Teile eines Projekts zu einer einzelnen Datei zusammenfasst, zum so genannten *Java Archive Format (jar)*

**Bemerkung:**

Da das Erstellen von ausführbaren Anwendungen im Informatikunterricht nicht im Vordergrund steht, habe ich mich entschlossen, die Methode *main()* in eine separate Klasse *Simulation* zu legen. Somit können die Schüler diese Klasse *Simulation* für alle weiteren Projekte einfach übernehmen.

```
public class Simulation
{
    Ansicht ansicht;

    public Simulation()
    {
        ansicht = new Ansicht();
    }

    public static void main(String[] args)
    {
        Simulation q = new Simulation();
    }
}
```

**Abbildung 4.25:** Die Klasse *Simulation* enthält die Methode *main()*

Die Abbildung 4.25 zeigt, dass die Klasse *Simulation* die Methode *main()* enthält. Wird die Methode *main()* also aufgerufen, wird ein Objekt *q* (Name ist von mir willkürlich gewählt) der Klasse *Simulation* erzeugt. Der Konstruktor von *q* erzeugt das Objekt *ansicht* der Klasse *Ansicht*, also genau das, was du bisher immer manuell in *BlueJ* getan hast.

Die Abbildung 4.26 zeigt, wie die Methode *main()* in *BlueJ* aufgerufen wird

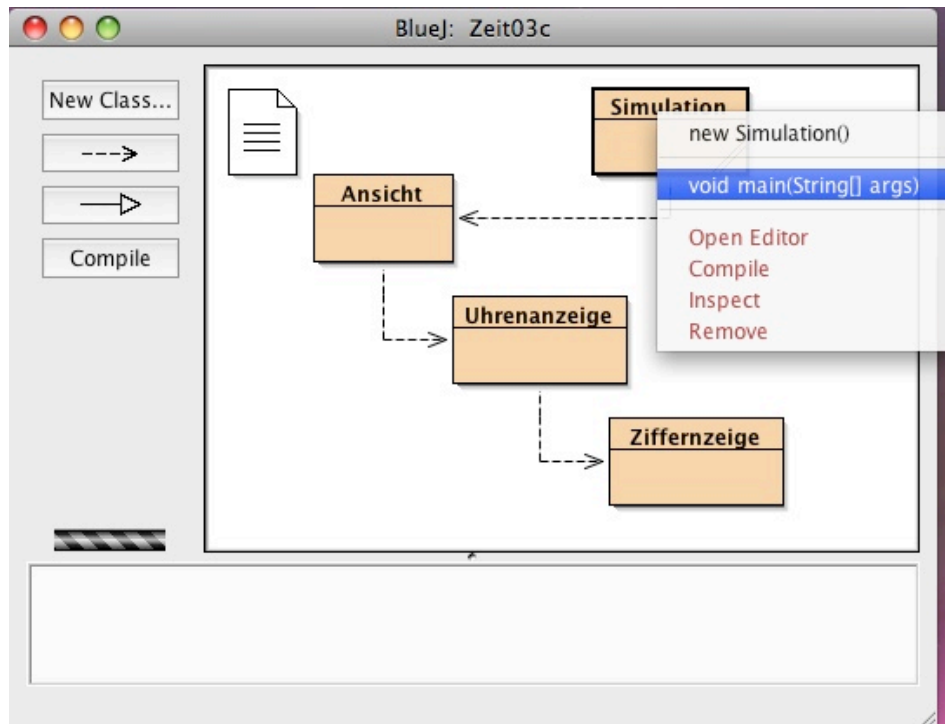


Abbildung 4.26: Aufruf der Methode *main()*

### Übung 4.20:

Erstelle die Klasse *Simulation* und teste die Methode *main()*.

Um eine ausführbare jar-Datei in *BlueJ* zu erstellen, musst du im Menü *Project* den Menüpunkt *Create Jar File...* aufrufen. Im folgenden Dialog wählst du die Klasse, die die Methode *main()* enthält, in diesem Fall also die Klasse *Simulation*. Optional kannst du noch entscheiden, ob du dem Benutzer den Quelltext und die BlueJ Project-Dateien mitliefern willst. Anschließend gibst du der jar-Datei noch einen geeigneten Namen und legst den Speicherort fest.

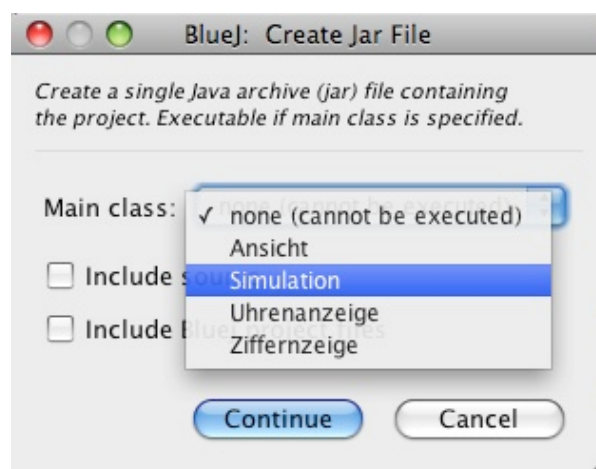


Abbildung 4.27: Erstellen einer jar-Datei

### Übung 4.21:

Erstelle eine jar-Datei *Digitaluhr.jar*.

Beende anschließend *BlueJ* und teste, ob *Digitaluhr.jar* eine stand-alone-Applikation ist.