Schriftliche Hausarbeit zur Abschlussprüfung der erweiternden Studien für Lehrer im Fach Informatik

VIII. Weiterbildungskurs in Zusammenarbeit mit der Fernuniversität Hagen

Eingereicht dem Amt für Lehrerausbildung – Außenstelle Gießen –

Objektorientiertes Modellieren mit Delphi

Einführung in die Konzepte der OOP anhand eines ausgewählten Beispiels

Verfasserin: Katja Weishaupt

Gutachter: Otto Wehrheim, StD

Inhaltsverzeichnis

1 Einleitung 3

2 Objektorientierte Programmierung 5

2.1 Die Idee der Objekte 5

2.2 Grundlegende Prinzipien der OOP 5

2.2.1 Datenkapselung 6

2.2.2 Vererbung 6

2.2.3 Polymorphie 6

2.3 Objektorientierte Modellierung 7

2.3.1 Objektorientierte Analyse mit UML 7

2.3.2 Realisierung in Delphi 9

3 Eine Unterrichtseinheit zur Einführung in die Konzepte der OOP 11

3.1 Vorkenntnisse 11

3.2 Zielsetzung und Begründung der Arbeit 11

3.3 Eine Uhr, objektorientiert 12

3.3.1 Überlegungen zur Objektstruktur 12

3.3.2 Der Objekttyp Zähler 13

3.3.3 Erweiterung der Zählerklasse 17

3.3.4 Zusammensetzen der Zählerobjekte zu einer größeren Einheit: Die Uhr 19

3.3.5 Die Uhr lernt laufen: Ein Timer kommt hinzu 22

3.3.6 Das Problem der Kommunikation zwischen Fachklasse und GUI-Klasse 23

3.3.7 Erweiterung der Uhrenklasse: Spezielle Uhren 25

3.3.8 Ein Wecker mit 7-Segment-Anzeige 27

3.3.9 Eine Analoguhr 31

3.4 Abschließende Überlegungen zur unterrichtlichen Umsetzung 34

3.4.1 Ein möglicher Unterrichtsverlauf 34

3.4.2 Ausblick 36

4 Literaturverzeichnis 37

5 Anhang 38

6 Inhaltsverzeichnis der CD 51

7 Erklärung 52

1 Einleitung

Die Geschichte der objektorientierten Programmierung beginnt in den 1960er Jahren mit der Programmiersprache Simula, in der die ersten Ansätze objektorientierten Programmierens entwickelt wurden. Dennoch sollte es noch eine Weile dauern, bis sich die dort erstmals verwendeten Begriffe und Verfahren schließlich durchsetzten und zu einem heute allgemein anerkannten und für die Software-Entwicklung bedeutenden Konzept weiterentwickelt wurden.

Bis etwa 1970 programmierte man vorwiegend maschinennah, das heißt, das Programm wurde durch eine Aneinanderreihung von Befehlen entwickelt. Solche Programme waren unübersichtlich, schlecht verständlich und daher sehr aufwändig in der Wartung. Diese Erkenntnis führte schließlich zur Entwicklung der Grundideen des strukturierten Programmierens, wodurch die bisherige konventionelle Programmentwicklung verdrängt wurde.

In der strukturierten Programmierung werden Probleme hierarchisch gegliedert, d. h. ein Gesamtproblem wird schrittweise durch die Einführung von Detaillierungsstufen in übersichtliche Teilprobleme zerlegt (modularisiert). Bei der Programmentwicklung beschränkt man sich auf wenige elementare Grundstrukturen für die Ablaufsteuerung. Die Programmiersprache Pascal in ihrer ursprünglichen Form ist bspw. ein typischer Vertreter der strukturierten Programmierung.

Schließlich führte die konsequente Weiterentwicklung der strukturierten Programmierung und des Modulbegriffs zur objektorientierten Programmierung. Hier stehen die Datenstrukturen im Mittelpunkt der Betrachtung. Sie haben eigene Algorithmen zur Verfügung, die sie zur Erledigung ihrer Aufgaben und Änderung ihres Zustandes brauchen. Ein entsprechendes Programm wird als Gemeinschaft miteinander kommunizierender und kooperierender Objekte formuliert.

Mitte der 1980er Jahre begann OOP immer beliebter zu werden, so dass seit dieser Zeit für viele existierende Programmiersprachen objektorientierte Erweiterungen geschaffen wurden, so z. B. auch für Pascal, das nun zu Object Pascal – der Grundlage der Entwicklungsumgebung Delphi – ausgebaut wurde.

Objektorientierung bedeutet aber mehr als ein besonders elegantes „Kochrezept“ zur Programmierung. Objektorientierung ist eine Entwurfsmethode, in welcher der zentrale und entscheidende Aspekt die Modellierung des Problembereichs ist. Im Unterricht bedeutet objektorientiertes Denken die Verlagerung der Schwerpunkte von der Programmierkompetenz hin zu einer Modellierungs- und Evaluationskompetenz [Baumann 1996, S. 281].

Die Bildung von Modellen ist ein universelles Arbeitsprinzip. Modelle dienen dazu, einen Ausschnitt eines komplexen Systems idealisiert und abstrahiert darzustellen bzw. zu repräsentieren und damit begreifbarer oder leichter ausführbar zu machen. Idealisierung bedeutet dabei, dass nur die für die jeweilige Aufgabe als wesentlich und wichtig angesehenen Teile des realen Gegenstandes wiedergegeben werden. Die Darstellung des Modells hängt von der individuellen Intention ab, die mit dem Modellierungsvorgang verbunden wird, bspw. können Modelle zur Erklärung von Vorgängen, Voraussage von Ereignissen oder als Entscheidungsgrundlage dienen.

In der Schule wird Modellbildung in nahezu allen Fächern praktiziert, häufig jedoch nicht explizit thematisiert. In der Informatik kommt der Modellierung eine Schlüsselrolle zu, sie ist eine der vier Leitlinien, die der Gestaltung des Informatikunterrichts in der gymnasialen Oberstufe zu Grunde gelegt werden (Hessisches Kultusministerium 2003, S. 3):

- Umgang mit Informationen
- Wirkprinzipien von Informatiksystemen

- Informatische Modellierung
- Wechselwirkung zwischen Informatiksystemen, Individuum und Gesellschaft

Im Lehrplan wird unter der Überschrift „Informatische Modellierung“ die Rolle der Modellbildung als ein zentrales Element des Problemlösens mit Informatiksystemen betont. Sie kommt dabei im Unterricht sowohl als Methode (Erarbeitung der grundlegenden Prinzipien von Informatiksystemen durch ihre Modellierung) als auch als Lerninhalt (Erlernen von Modellierungstechniken zur Beschreibung komplexer Systeme) vor (Hubwieser 2001, S. 86).

In dieser Arbeit soll eine Unterrichtseinheit beschrieben werden, die anhand eines konkreten Beispiels einen möglichen Weg zur Einführung in die Konzepte der objektorientierten Modellierung mit Hilfe der Programmiersprache Delphi aufzeigt. Das Beispiel wird bis zu einer bestimmten Stufe ausgearbeitet, ohne dabei explizit einen konkreten Unterrichtsverlauf zu entwerfen. An vielen Stellen werden jedoch Gedanken zur unterrichtlichen Umsetzung eingeflochten und mögliche Erweiterungen des vorgestellten Konzepts aufgezeigt.

2 Objektorientierte Programmierung

Die objektorientierte Sichtweise kommt der menschlichen Art, die Umwelt wahrzunehmen, besonders nahe. Objektorientierung macht komplexe Systeme deutlich überschaubarer, Anwendungen, die nach diesem Prinzip programmiert sind, können wesentlich einfacher gewartet, geändert oder erweitert werden. Darüber hinaus erleichtert das Konzept die Wiederverwendung von Teilen in anderen Projekten, was insbesondere bei der Erstellung größerer Programmsysteme zu einer erheblich höheren Effizienz (und Kosteneinsparung) führt.

2.1 Die Idee der Objekte

Das zentrale Element im objektorientierten Modell ist das Objekt. Jedes Objekt repräsentiert einen Gegenstand des zu betrachtenden Problembereichs, wobei Gegenstand nicht notwendigerweise ein fassbares Ding oder eine Person sein muss sondern auch einen Sachverhalt meinen kann.

Das Objekt wird gekennzeichnet durch charakteristische Merkmale (Attribute) und hat zu jedem Zeitpunkt einen spezifischen Zustand, der durch die aktuellen Werte aller seiner Attribute bestimmt ist.

Ein Objekt kann Botschaften empfangen und auf diese Nachrichten reagieren indem es entsprechende Aktionen ausführt. Darauf beruht das Konzept der objektorientierten Sichtweise: Ein Programm besteht aus miteinander kommunizierenden und interagierenden Objekten. Die Handlungsvorschriften zur Ausführung dieser Aktionen werden in den objekteigenen Methoden beschrieben.

Die in einem Modell vorkommenden Objekte werden nicht alle individuell entworfen. Gleichartige Objekte, also Objekte mit den gleichen Merkmalen (Attributen) und dem gleichen Verhalten (Methoden) werden in einer gemeinsamen Kategorie, der Klasse zusammengefasst. Die Klasse besitzt einen eindeutigen Namen, sie ist der Bauplan, mit dessen Hilfe die konkreten Objekte erzeugt werden.

Ein einzelnes, konkretes Objekt einer Klasse bezeichnet man als Exemplar oder Instanz dieser Klasse. Die Klasse besitzt eine Operation, die es ermöglicht, neue Objekte dieser Klasse zu erzeugen bzw. zu konstruieren. Solche Operationen bezeichnet man als Konstruktoren. Diejenige Operation, die zur Vernichtung eines konkreten Objekts führt, wird Destruktor genannt.

Neben den oben beschriebenen konkreten Klassen, von denen Objektinstanzen gebildet werden, gibt es auch so genannte abstrakte Klassen. Im Unterschied zu konkreten Klassen erzeugen abstrakte Klassen keine Objekte. Sie enthalten lediglich die Deklaration von Methoden, ohne dass diese auch implementiert werden. Abstrakte Klassen werden als Muster modelliert, aus denen die Eigenschaften speziellerer Klassen abgeleitet werden und sind besonders für die Vererbung von Bedeutung.

2.2 Grundlegende Prinzipien der OOP

Neben der Bildung von Objekten und Klassen sind die folgenden drei Prinzipien grundlegend für die objektorientierte Programmierung: Kapselung, Vererbung und Polymorphie.

2.2.1 Datenkapselung

Ein Objekt vereinigt Daten und Methoden. Jedes Objekt „weiß“ durch die ihm zur Verfügung stehenden Methoden, wie es bestimmte Aktionen ausführen muss, um die ihm zugewiesene Aufgabe ausführen zu können. Über definierte Schnittstellen kann ein Objekt Nachrichten von anderen Objekten empfangen und zur Freigabe von Zustandsinformationen oder Ausführung von Zustandsänderungen veranlasst werden. Der unmittelbare Zugriff auf die Daten eines Objekts, die Ermittlung oder Änderung seines Zustands kann nur durch das Objekt selbst, nämlich durch die nur ihm bekannte Handlungsvorschrift erfolgen.

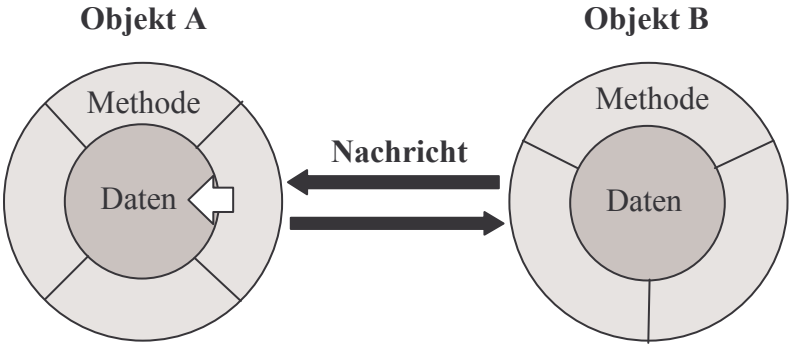


Abb. 2.1: Kommunikation zwischen Objekten

Dieses Prinzip wird als Datenkapselung bezeichnet, bei der die Daten und die darauf wirkenden Operationen in einem Objekt zusammengefasst werden und der Umgebung das im Objekt gekapselte Wissen verborgen bleibt. Man spricht in diesem Zusammenhang auch vom „Geheimnisprinzip“ (information hiding). Die Kapselung ermöglicht Unabhängigkeit: Einzelne Objekte eines Programms können in ihrem Inneren verändert werden, ohne dass sich daraus Konsequenzen für die übrigen Objekte, also das Zusammenspiel im ganzen Programm ergeben.

2.2.2 Vererbung

Vererbung bedeutet die Möglichkeit, eine Klasse von Objekten als Spezialfall einer allgemeineren Klasse zu definieren. Die Objekte dieser spezialisierten Klasse verfügen über alle Merkmale und Methoden der allgemeinen Klasse, erweitern diese aber ggf. um zusätzliche Eigenschaften (Attribute und Methoden) und können die geerbten Methoden durch überschreiben neu definieren.

Der Vorteil dieses Vorgehens besteht darin, dass beim Entwurf der Klassen mehrfach verwendete Bausteine nur jeweils einmal beschrieben werden müssen: Alle Attribute und Methoden der übergeordneten Klasse sind in den Teilklassen verfügbar, ohne dass sie dort explizit aufgeführt werden. Wird an ein Objekt eine Nachricht geschickt, so beginnt die Suche nach einer geeigneten Methode zunächst in der Klasse des Objekts. Findet sich dort nichts, so geht die Suche nacheinander in sämtlichen übergeordneten Klassen weiter, bis die passende Methode gefunden wird (oder die Suche möglicherweise erfolglos abbricht).

2.2.3 Polymorphie

Eine Teilklasse erbt alle Eigenschaften der ihr übergeordneten Klasse. Allerdings kann es notwendig sein, eine bestimmte geerbte Methode in der Teilklasse neu zu definieren und die

geerbte Methode damit zu überschreiben. Sie ist nun zwar namensgleich mit der Methode der übergeordneten Klasse, enthält aber eine geänderte Handlungsvorschrift.

Es ist problemlos möglich in verschiedenen Klassen den gleichen Namen für unterschiedliche Methoden zu verwenden, dies wird in der objektorientierten Programmierung mit Polymorphie (griech.: Vielgestaltigkeit) bezeichnet. Der Sender einer Nachricht muss nicht vorher prüfen, an welchen Objekttyp seine Nachricht geht, vielmehr wird das empfangende Objekt automatisch die typspezifische Reaktion zeigen.

Die Entscheidung, welche Methode aufgerufen wird, kann entweder zur Laufzeit (dynamische Bindung) oder bereits bei der Übersetzung des Programms (statische Bindung) getroffen werden. Dies hängt von der Deklaration der Methoden ab: Bei statischen Methoden bestimmt der *deklarierte* Typ der Objektvariablen, welche Implementierung aktiviert wird. Dynamische Bindung wird durch virtuelle Methoden realisiert, hier wird diese Entscheidung über den zur Laufzeit des Programms *aktuellen* Typ der Objektvariablen getroffen.

2.3 Objektorientierte Modellierung

Bei der objektorientierten Modellierung werden drei Phasen durchlaufen: Objektorientierte Analyse (OOA), objektorientiertes Design (OOD) und Implementierung in einer objektorientierten Sprache (OOP).

Der Entwurfsprozess beginnt mit der objektorientierten Analyse, deren Ziel es ist, das betrachtete Problem durch ein Objektmodell zu beschreiben. Zunächst müssen die relevanten Objekte gefunden und klassifiziert werden, die Eigenschaften (Attribute und Methoden), die für sie kennzeichnend (und für das Problem wesentlich) sind, werden bestimmt. Schließlich werden die Beziehungen der Objekte untereinander festgelegt.

Das in dieser ersten Phase erstellte Modell beinhaltet noch keine Informationen über die spätere Benutzungsoberfläche des Programms und die Darstellung der Daten auf dem Bildschirm. Der Entwurf folgt damit der so genannten Schichten-Architektur, in der das Fachkonzept (die interne Datenverarbeitung) von der grafischen Benutzungsoberfläche (GUI-Schicht) völlig getrennt wird. Die Fachkonzeptschicht besitzt kein Wissen über die Benutzungsoberfläche und muss demzufolge nicht angepasst werden, wenn sich die Benutzungsoberfläche ändert. Das eröffnet gleichzeitig die Möglichkeit, Daten auf unterschiedliche Art und Weise am Bildschirm darstellen zu können.

In der zweiten Phase des Entwurfs, dem objektorientierten Design, wird dann die GUI-Klasse modelliert und die Fachklassen werden an die Benutzungsoberfläche angebunden. Eventuell muss das Objektmodell noch an die verwendete Programmiersprache angepasst werden.

Eng verbunden mit dem objektorientierten Design ist schließlich der letzte Schritt, die Umsetzung des Entwurfs in einer objektorientierten Programmiersprache, bspw. Delphi.

2.3.1 Objektorientierte Analyse mit UML

Um den Modellierungsprozess in zweckmäßiger, einheitlicher und überschaubarer Weise darzustellen und abzubilden sind grafische Vereinbarungen notwendig. Für die objektorientierte Modellierung bietet hier die Beschreibungssprache UML geeignete Elemente für standardisierte grafische Darstellungen, die den Modellbildungsprozess begleiten.

In der UML werden die Diagramme in zwei Gruppen unterteilt, statische und dynamische Diagramme. Statische Diagramme drücken die unveränderlichen Eigenschaften und Beziehungen der vorhandenen Elemente untereinander aus, dynamische Diagramme bilden das Zusammenspiel der beteiligten Elemente ab. In der hier vorliegenden Arbeit werden von den vielen möglichen Diagrammtypen der UML nur die (statischen) Klassendiagramme verwendet. Die dafür notwendigen Symbole sollen im Folgenden kurz vorgestellt werden.

In der UML werden Klassen durch ein einfaches, in drei Segmente unterteiltes Rechteck symbolisiert.

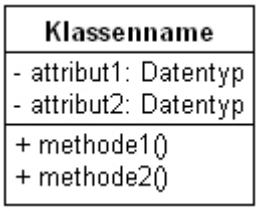


Abb. 2.2: Klassensymbol

Das erste Segment enthält den Klassennamen (in Fettdruck, beginnend mit einem Großbuchstaben), im zweiten Abschnitt folgen die Attribute der Klasse (evtl. mit den dazugehörigen Datentypen), im dritten Abschnitt stehen die zu der Klasse gehörenden Methoden.

Attribute und Methoden können zusätzlich durch Angaben bzgl. ihrer Sichtbarkeit gekennzeichnet werden. Sie legt die Zugriffsmöglichkeiten von Programmeinheiten außerhalb der aktuellen Klasse fest. Dabei unterscheidet man folgende Stufen:

- *Public* (öffentlich, für alle Klassen sichtbar und benutzbar), UML-Symbol: +
- *Protected* (geschützt, für die Klasse selbst und alle ihre Unterklassen sichtbar und benutzbar), UML-Symbol: #
- *Private* (privat, nur für die Klasse selbst sichtbar und benutzbar), UML-Symbol: –

Darüber hinaus können Methoden zusätzliche Beschreibungen erhalten, wie bspw. *constructor*, *destructor*, *virtual*, *override*, …

Vererbungsbeziehungen werden in der UML durch einen Pfeil von der Unterklasse (dem speziellen Element) zur Oberklasse (dem generellen Element) dargestellt.

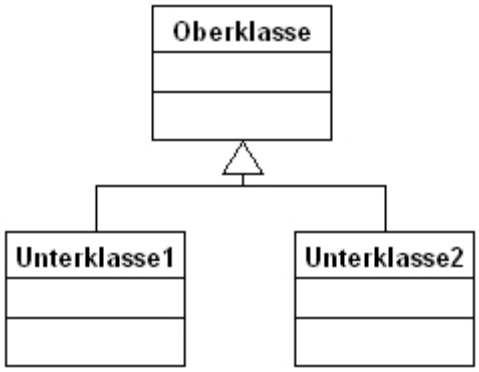


Abb. 2.3: Vererbungsbeziehung

In den Unterklassen werden nur die jeweils hinzugekommenen Attribute und Methoden beschrieben sowie diejenigen geerbten Methoden, die durch überschreiben modifiziert wurden.

Neben der Vererbung sind die folgenden Beziehungstypen zwischen Klassen bzw. Objekten von Bedeutung:

Die Assoziation beschreibt eine Nutzungsbeziehung zwischen zwei völlig selbständigen Objekten. Sie wird auch als KENNT-Beziehung bezeichnet. In der UML wird eine solche Beziehung durch eine einfache Linie zwischen den beteiligten Objekten dargestellt. Die Zahlen über der Linie drücken die Kardinalitäten der Beziehung aus, sie geben an, wie viele Objekte aus beiden Klassen jeweils miteinander kommunizieren.

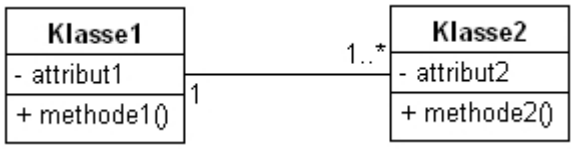


Abb. 2.4: Assoziation zweier Klassen

Die Aggregation wird auch als HAT-Beziehung bezeichnet. Sie beschreibt eine Beziehung, bei der Objekte einer Klasse Teil von Objekten einer anderen Klasse sind. Auch hier werden die zugehörigen Kardinalitäten angegeben, die ausdrücken, aus wie vielen Teilen sich ein Objekt zusammensetzt, bzw. zu wie vielen Objekten die Teilobjekte gehören.

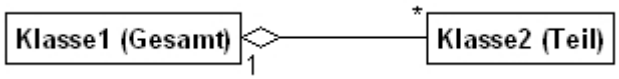


Abb. 2.5: Aggregation

Die Komposition ist ein Spezialfall der Aggregation. Hier können die Teilobjekte nicht selbständig existieren, ihre Existenz ist immer untrennbar mit der Existenz des Ganzen verbunden. Folglich kann ein Teilobjekt auch immer nur zu genau einem Aggregat gehören. Die Darstellung der Komposition entspricht der Darstellung einer Aggregation, zur Unterscheidung wird die Raute auf der Seite des ranghöheren Objekts ausgefüllt dargestellt.

Für die Erstellung von UML-Diagrammen im Unterricht ist es sinnvoll ein geeignetes Programm einzusetzen, das die notwendigen grafischen Elemente beherrscht. Die Diagramme in dieser Arbeit wurden mit UMLPad[1] erstellt, einem einfachen (englischsprachigen) Freeware-Tool, das ohne Installation auskommt und sich für das Zeichnen von Klassendiagrammen gut eignet. Noch komfortabler sind Werkzeuge, die es ermöglichen, die erstellten Klassendiagramme in den Quellcode der verwendeten Programmiersprache zu übersetzen. Ein für die Schule geeignetes Tool, das die Code-Generierung in Delphi unterstützt, ist bspw. UMLed[2]. Schulen können für dieses Programm eine kostenlose Lizenz erhalten.

2.3.2 Realisierung in Delphi

In diesem Abschnitt soll auf einige spezielle Punkte bei der Realisierung des objektorientierten Modells mit Delphi eingegangen werden.

Die Implementierung der GUI-Klasse und der Fachklassen erfolgt generell in verschiedenen Units. Die Unit der Fachklassen wird über die *Uses*-Klausel in die Unit der GUI-Klasse eingebunden.

[1] Im Internet unter http://web.tiscali.it/ggbhome/
[2] Im Internet unter http://www.kubitz-online.de/UMLed/

Die Attribute einer Klasse werden grundsätzlich als privat (*private*) deklariert. Methoden sind in der Regel öffentlich (*public*), es sei denn, sie dienen nur internen Hilfszwecken, dann werden sie als privat deklariert. Die in Delphi vorgesehene Unterscheidung der beiden Sichtbarkeitsattribute *published* und *public* spielt in der hier vorliegenden Arbeit keine Rolle, weswegen darauf nicht näher eingegangen wird. Etwas problematischer ist in Delphi die Unterscheidung zwischen *private* und *protected*: Prinzipiell kann auf *Private*-Elemente nur innerhalb der Klasse zugegriffen werden, *Protected*-Elemente sind auch in abgeleiteten Klassen sichtbar. In Delphi können Unterklassen jedoch auch dann auf *Private*-Elemente zugreifen, wenn sich die verwandten Klassendeklarationen in derselben Unit befinden. Im Unterricht muss darauf ggf. gesondert eingegangen werden.

Die für die Benutzeroberfläche erzeugten Ereignisprozeduren werden von Delphi automatisch im *Published*-Teil der Deklaration angelegt. Selbst vereinbarte Methoden der GUI-Klasse werden dagegen (aus Gründen der Übersichtlichkeit) immer als privat deklariert.

Bei Klassen, die als Attribute Objekte andere Klassen enthalten (Aggregation) muss die Konstruktormethode die Erzeugung und Initialisierung der aggregierten Objekte übernehmen. Für das Löschen mittels der Destruktormethode gilt das entsprechende.

3 Eine Unterrichtseinheit zur Einführung in die Konzepte der OOP

3.1 Vorkenntnisse

Die im Folgenden vorgestellte Unterrichtseinheit ist für einen Informatik-Kurs der Jahrgangsstufe 12 im ersten Kurshalbjahr gedacht. Dabei werden Kenntnisse und Fertigkeiten im Umgang mit der Programmierumgebung Delphi und Grundlagen von Object Pascal vorausgesetzt, wie sie in den Informatik-Kursen der Jahrgangsstufe 11 vermittelt werden.

So wird angenommen, dass die Schüler Übung in der Erstellung von Formularen mit Standardkomponenten haben, die Standarddatentypen mit den darauf zulässigen Operationen kennen und mit Kontrollstrukturen arbeiten können. Die Schüler sollten die Unterschiede zwischen Funktionen und Prozeduren kennen und beide Methodentypen richtig einsetzen können.

3.2 Zielsetzung und Begründung der Arbeit

Die Methoden und Werkzeuge der Informatik, die Denk- und Herangehensweisen sowie die Nutzung von Informatiksystemen findet inzwischen in fast allen Gebieten von Wissenschaft, Wirtschaft und Technik Eingang [Hessisches Kultusministerium 2003, S. 2]. Die Automatisierung weiter Teile unserer Lebenswelt erfordert, dass diese verstanden und formal beschrieben werden können. Einen zentralen Lerninhalt des Faches Informatik bildet daher die Kenntnis exemplarischer Methoden und Verfahren zur Modellierung eines Ausschnittes der Wirklichkeit. Modell soll hierbei eine vereinfachte struktur- und verhaltenstreue Beschreibung eines realen Systems bedeuten [Baumann 1996, S. 161].

Der hessische Lehrplan für das Fach Informatik führt den Themenkomplex Objektorientiertes Modellieren verbindlich für das 1. Kurshalbjahr der Jahrgangstufe 12 auf. Aufbauend auf den in der Jahrgangsstufe 11 geschaffenen Grundlagen für das Programmieren sollen sich die Schüler nun mit den Konzepten, Methoden und Verfahren der objektorientierten Modellbildung auseinandersetzen [Hessisches Kultusministerium 2003, S. 13].

Die hier vorgeschlagene Unterrichtseinheit ist für den Einstieg in die Inhalte dieses Kurshalbjahres gedacht. Anhand eines ausgewählten Themas – die Programmierung einer Uhr – beschäftigen sich die Schüler mit den zentralen Begriffen und Konzepten der objektorientierten Programmierung. Gleichzeitig lernen sie mit der grafischen Modellierungssprache UML eine Möglichkeit kennen und anwenden, um die Ergebnisse des Entwurfsprozesses geeignet strukturiert zu visualisieren. Von den verschiedenen Diagrammtypen der UML werden hier jedoch nur die Klassendiagramme thematisiert.

Die Programmierung einer Uhr als Leitidee für eine Einführung in die Konzepte der objektorientierten Programmierung ist meiner Ansicht nach aus mehreren Gründen gut geeignet: Uhren sind den Schülern selbstverständlich vertraut, die Modellierung der unterschiedlichen Uhren-Objekttypen ist einfach und kann von den Schülern in weiten Teilen selbständig erarbeitet werden. Das Thema Uhren ist vielschichtig genug, um eine angemessen umfangreiche Klassenhierarchie zu erstellen und die zentralen Ideen der objektorientierten Programmierung daran zu verdeutlichen. Die programmiertechnische Umsetzung ist unkompliziert und beinhaltet keine größeren algorithmischen Probleme, so dass der Unterricht sich auf die wesentlichen Fragestellungen konzentrieren kann (dennoch ist es natürlich möglich, auch hier kniffligere Varianten zu finden). Darüber hinaus ist das Thema auch geeignet, den Erwartungen der

Schüler im Hinblick auf die Entwicklung einer „ansprechenden“ Anwendung (bezogen auf Darstellung und Funktionalität) gerecht zu werden.

Der hier beschriebene Vorschlag orientiert sich im Wesentlichen an den Anforderungen, wie sie für einen Grundkurs formuliert werden (themenorientiert, exemplarisch, anwendungsbezogen). Gleichwohl kann er natürlich auch für den Einstieg eines Leistungskurses in das Thema verwendet werden. Hier könnten die einzelnen Entwicklungsschritte für ein vertieftes Verständnis der fachwissenschaftlichen Begriffe, Konzepte und Strukturen, wie es für einen Leistungskurs gefordert wird, durch ausführlichere theoretische Betrachtungen stärker problematisiert werden.

3.3 Eine Uhr, objektorientiert

Es gibt viele verschiedene Uhrentypen. Abgesehen von ihrer Größe und optischen Ausgestaltung (z. B. Armbanduhren, Taschenuhren, Kaminuhren, Standuhren, usw.) und der Darstellung der Zeitanzeige (analog oder digital) unterscheiden sie sich vor allen Dingen in ihrer Funktionalität (z. B. Uhren mit einfacher Zeitanzeige, mit oder ohne Läutwerk, Wecker, Stoppuhren, Weltzeituhren). Doch trotz dieser unterschiedlichen Ausgestaltung verknüpfen wir mit allen diesen Objekten den Begriff Uhr, ein Zeitmesser, der uns Auskunft über die abgelaufenen Stunden, Minuten und Sekunden gibt.

Diese Idee bildet den Ausgangspunkt für die hier beschriebene Unterrichtseinheit. Ausgehend von den Basiseigenschaften und Methoden, die allen Uhren gemeinsam sind und den Objekttyp Uhr definieren, soll im Laufe der Unterrichtseinheit die Uhrenklasse erweitert werden. Spezielle Uhrentypen mit charakteristischen zusätzlichen Methoden und Eigenschaften, wie sie bspw. allen Weckern gemeinsam sind, werden vom Basistyp Uhr abgeleitet und in die Objekttyphierarchie eingefügt.

3.3.1 Überlegungen zur Objektstruktur

Der Entwurfsprozess beginnt mit Überlegungen zur Objektstruktur. Die zu entwickelnde Uhr soll zunächst nur über diejenigen Eigenschaften verfügen, die prinzipiell alle Uhrentypen in sich vereinen, die also kennzeichnend für den Objekttyp Uhr sind. Vorab müssen die Schüler sich daher über folgende Fragen Gedanken machen:

- Welche Merkmale sind charakteristisch für eine Uhr, also allen Uhrentypen gemeinsam?
- Welche Funktionalitäten gehören zu den Basiseigenschaften einer Uhr? Anders gefragt: Welche Dienste muss eine Uhr zur Verfügung stellen, um ihre Aufgabe zu erfüllen?
- Lässt sich das Objekt Uhr noch zerlegen? Enthält es weitere Objekte mit spezifischen Merkmalen und Methoden, die zur Erledigung von Teilaufgaben einer Uhr geeignet sind?

Die letzte der oben genannten Fragen ist sicherlich die wichtigste für den Modellierungsprozess. Im Laufe ihrer Überlegungen wird für die Schüler deutlich werden, dass sich eine Uhr als ein aus Teilobjekten zusammengesetztes Objekt auffassen lässt. Zentrales Element einer Uhr ist ein Zähler, genauer: Für jede Zeiteinheit (Stunden, Minuten, Sekunden) kann ein eigener Zähler eingesetzt werden. Es ist also sinnvoll, in einem ersten Schritt zunächst das Zählerobjekt zu modellieren, es anschließend zu implementieren und nachfolgend zu testen.

Wichtig ist es, gleich zu Beginn darauf aufmerksam zu machen, dass im Entwurf die spätere Darstellung in einer Anwendung (zunächst) keine Rolle spielen soll. Die Ansicht soll von der

Anwendung frei gewählt werden können. Das zu entwickelnde Modell enthält nur alle für die interne Datenverarbeitung einer Uhr notwendigen Informationen, also das reine Fachkonzept.

3.3.2 Der Objekttyp Zähler

Welche Eigenschaften des Objekts Zähler sind für die Beschreibung in einem Programm von Interesse? Die Analyse könnte zu folgenden Ergebnissen führen:

- Der Zählerstand: Er soll nur positive Werte (bzw. 0) annehmen können.
- Zählen bedeutet, dass der Zählerstand in Einerschritten erhöht wird.
- Der Zähler kann wieder in den Startzustand (Zählerstand 0) zurückversetzt werden.
- Der Zähler kann auf einen bestimmten (Anfangs-)Zählerstand gesetzt werden.
- Der Zählerstand kann ausgelesen werden.

Der Zählerstand beschreibt den Zustand des Zählers zu einem ganz bestimmten Zeitpunkt. Es handelt sich also um eine Zustandseigenschaft, ein Attribut. Zählen, Zählerstand auslesen, usw. sind dynamische Eigenschaften, denn sie beschreiben die Reaktion des Zählers auf eine bestimmte Nachricht. Hier handelt es sich also um Operationen (Methoden).

Die Analyse der Zählereigenschaften muss nicht zwangsläufig zum oben dargestellten Ergebnis führen. Beispielsweise ist für das Rückversetzen des Zählers in den Anfangszustand nicht notwendigerweise eine eigene Methode erforderlich (die Liste enthält eine weitere Methode – Zählerstand setzen – die für diese Aufgabe eingesetzt werden könnte). Ebenso wäre es möglich, für den Zähler auch negative Zählerstände zuzulassen. Den Schülern sollte bewusst gemacht werden, dass der Objektentwurf die individuellen Interessen und Erwartungen an die spätere Simulation wiedergibt. Daher ist es notwendig, sich zuvor ein genaues Bild von den Anforderungen zu machen.

Die vereinbarten Eigenschaften werden nun in der Klasse *TCounter*[3] zusammengefasst. Sie ist gewissermaßen die Schablone, die angibt, wie Zählerobjekte auszusehen haben. Zusätzlich zu den oben beschriebenen Eigenschaften erhält die Klasse *TCounter* eine Methode, die es erlaubt, Zählerobjekte zu erzeugen, den Konstruktor *Create*. Er reserviert Speicher für ein neues Zählerobjekt und initialisiert den Zählerstand mit 0. Der Destruktor *Destroy* sorgt dafür, dass bei Zerstörung des Zählerobjektes der reservierte Speicher wieder freigegeben wird.

Es ist für die Zählerklasse zwar nicht notwendig, eine klasseneigene Definition eines Destruktors zu erstellen (er unterscheidet sich nicht von dem von *TObject* automatisch geerbten Destruktor), die Schüler sollten aber bereits an dieser Stelle die Bedeutung des Destruktors kennen lernen, so dass es sinnvoll erscheint, ihn explizit in die Klassenmethoden aufzunehmen.

Eine Darstellung der Klasse in UML ergibt folgendes Bild:

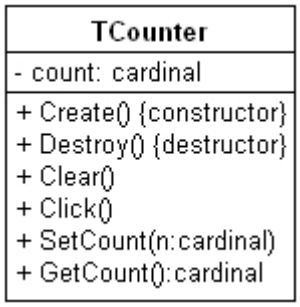


Abb. 3.1: Klassendiagramm

[3] In den Fachklassen werden aus Gründen der Einheitlichkeit grundsätzlich englische Bezeichner gewählt.

Der Zählerstand (*count*) repräsentiert den privaten Datenbereich der Klasse, er soll von außen nicht direkt manipuliert werden können. Dies darf nur über die entsprechenden – öffentlichen – Methoden möglich sein.

Als nächster Schritt folgt die Implementierung der Zählerklasse. Zunächst werden die Attribute und Methoden der Zählerklasse im *interface*-Abschnitt der Unit deklariert:

```
unit UCounter;

interface

type
  TCounter = class                        //von TObject abgeleitet
  private
    count: cardinal;                      //Zählerstand
  public
    constructor Create;                   //Zählerobjekt erzeugen
    destructor Destroy; override;         //Zählerobjekt löschen
    procedure Clear;                      //Zählerstand zurück setzen
    procedure Click;                      //Zählerstand um 1 erhöhen
    procedure SetCount(n: cardinal);      //Zählerstand auf Anfangswert setzen
    function GetCount: cardinal;          //Zählerstand auslesen
  end;
```

Die Bedeutung der Direktive *override* bei der Deklaration des Destruktors soll zu diesem Zeitpunkt noch nicht thematisiert werden, dies ergibt sich an späterer Stelle.

Im Implementations-Teil werden die Methoden definiert, dabei wird der Klassenname dem Methodennamen vorangestellt:

```
implementation

constructor TCounter.Create;
begin
  count:= 0;
end;

destructor TCounter.Destroy;
begin
  inherited Destroy;
end;

procedure TCounter.Clear;
begin
  count:= 0;
end;

procedure TCounter.Click;
begin
  Inc(count);
end;

procedure TCounter.SetCount(n: cardinal);
begin
  count:= n;
end;

function TCounter.GetCount: cardinal;
begin
  GetCount:= count;
end;

end.
```

Im Anschluss an die Implementierung der Zählerklasse sollte ein erster Test durchgeführt werden, der die Funktionsweise des Zählers demonstriert und zeigt, wie die Zählerklasse in eine Anwendung eingebunden werden kann. Eine mögliche Anwendungsoberfläche könnte bspw. so aussehen:

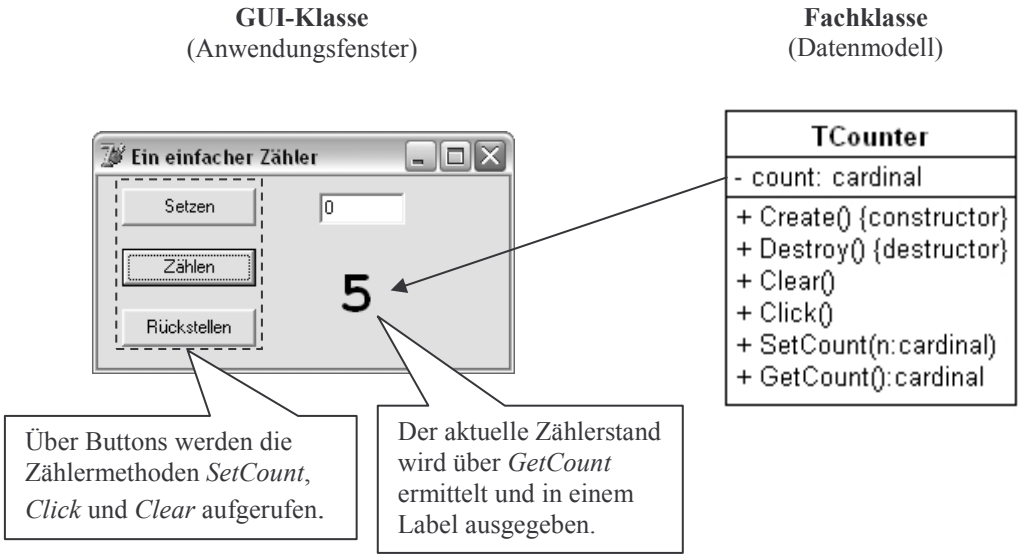


Abb. 3.2: Anwendungsfenster

Bei der Erstellung der Anwendungsoberfläche arbeiten die Schüler wiederum mit Objekten, deren Klassen von Delphi in diversen Units bereitgehalten und beim Erstellen einer neuen Formularanwendung automatisch zur Verfügung gestellt werden. Das Hauptformular und die grafischen Komponenten sind Objekte, um deren Erzeugung und Vernichtung man sich nicht konkret kümmern muss, da das Delphi-System hier automatisch die notwendigen Maßnahmen ergreift. Für die Erzeugung des Zählerobjektes muss in der Anwendung jedoch selbst gesorgt werden. Dazu wird im Implementations-Teil der GUI-Klasse zunächst eine Variable des entsprechenden Objekttyps vereinbart. Die Unit der Fachklasse muss zuvor über die *Uses*-Klausel in die Unit der GUI-Klasse eingebunden werden. In den Methoden *FormCreate* bzw. *FormDestroy* wird das Zählerexemplar schließlich durch Aufruf der entsprechenden Konstruktor- bzw. Destruktormethode erzeugt oder vernichtet.

Ein Diagramm, das die Beziehungen der in der Anwendung verwendeten Klassen untereinander beschreibt, sieht wie folgt aus:

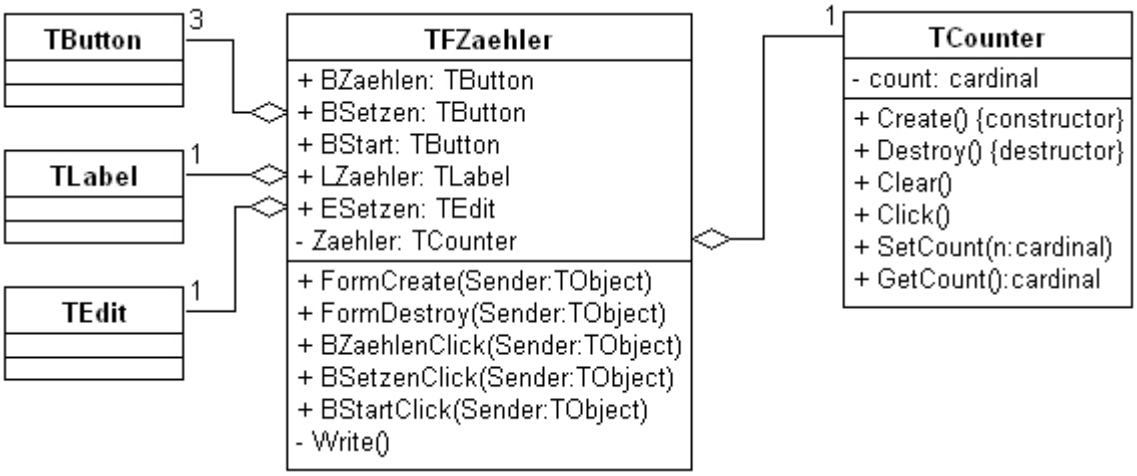


Abb. 3.3: Beziehungsdiagramm

Die Realisierung soll hier nur ausschnittsweise vorgestellt werden:

```
unit UZaehler;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, UCounter;   //TCounter

type
  TForm1 = class(TForm)
    ...                     //Grafische Komponenten und Ereignisprozeduren
  private
    Zaehler: TCounter;      //Referenzvariable für das Zählerobjekt
    procedure Write;        //Ausgabe des Zählerstandes im Formular
  end;

var
  Form1: TForm1;            //Referenzvariable für das Formularobjekt

implementation
{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Zaehler:= TCounter.Create;     //Erzeugung des Zählerobjektes, Zaehler
  Write;                         //erhält einen Zeiger auf das Objekt
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  Zaehler.Destroy;               //Das Zählerobjekt wird zerstört
end;

...
end.
```

In der hier vorgestellten Beispielanwendung bleiben die Aktivitäten des Konstruktors und Destruktors für die Schüler im Verborgenen (abgesehen davon, dass in der Anwendung – wie gewünscht – das Zählerobjekt zur Verfügung steht). Soll das Geschehen beim Erzeugen und Vernichten des Zählerobjektes etwas genauer betrachtet werden, bietet sich z. B. folgende Alternative an: Statt das Zählerobjekt mit dem Formular zu erzeugen bzw. zu löschen werden der Anwendung zwei weitere Buttons hinzugefügt, die diese Aufgabe übernehmen. In der Unit *System* werden zwei Variablen zur Verfügung gestellt, mit deren Hilfe man zu jeder Zeit feststellen kann, wie viele Speicherblöcke die Anwendung aktuell belegt (*AllocMemCount*) und wie viel Byte des Speichers die Anwendung verwendet (*AllocMemSize*). In zwei zusätzlichen Edit-Feldern können diese Werte ausgegeben werden, so dass sich nun beobachten lässt, wie beim Aufruf des Konstruktors zusätzlicher Speicher belegt und dieser beim Aufruf des Destruktors wieder freigegeben wird[4].

Diese Variante lässt die Schüler noch einen weiteren interessanten Effekt beobachten: Mehrfaches Betätigen des Button zum Erzeugen eines Zählerobjektes erhöht jedes Mal den belegten Speicher, denn bei jedem Aufruf von *Create* wird eine neue Instanz der Zählerklasse erstellt. Das Löschen funktioniert jedoch nur ein einziges Mal: Die zuvor erzeugten Zählerobjekte können nicht mehr angesprochen werden und verbleiben im Speicher. Dies hängt damit zusammen, dass die in der GUI-Klasse vereinbarte Variable *Zaehler* lediglich einen Verweis auf ein Zählerobjekt beinhaltet. Mit jedem neu erzeugten Zählerobjekt ändert sich aber die in *Zaehler* gespeicherte Verweisadresse und die Referenz auf die vorher erzeugten Instanzen geht verloren.

[4] Ein entsprechendes Anwendungsbeispiel befindet sich auf der beiliegenden CD.

3.3.3 Erweiterung der Zählerklasse

Für die Zählerklasse bietet sich – insbesondere im Hinblick auf das angestrebte Ziel, nämlich die Verwendung in einer Uhr – die folgende Verfeinerung an: Nach 60 Minuten ist eine Stunde erreicht und der Minutenzähler beginnt wieder bei 0, nach 24 Stunden wiederum soll der Stundenzähler erneut bei 0 starten, eine zweckmäßige Erweiterung des Zählers legt also eine Obergrenze fest, bis zu der gezählt werden soll (und lässt den Zähler bei Erreichen der Grenze wieder bei 0 starten).

Die neue Zählerklasse soll *TModCounter* heißen. Ihre Eigenschaften unterscheiden sich von der ursprünglichen Zählerklasse in folgenden Punkten:

- Es gibt ein weiteres Attribut, in dem die Zählergrenze gespeichert wird.
- Es wird eine zusätzliche Methode zur Festlegung der Zählergrenze zur Verfügung gestellt.
- Beim Zählen soll das Erreichen der Obergrenze überprüft werden.
- Es soll nicht möglich sein, den Startwert des Zählerstandes über der Zählergrenze festzulegen.

Die Klasse *TModCounter* ist eine Spezialisierung der ursprünglichen Klasse *TCounter*. Sie erbt alle Attribute und Methoden ihres Vorfahren und fügt neue Eigenschaften hinzu. Die letzten beiden Punkte zeigen, dass es auch notwendig ist, bereits vorhandene Methoden in der Klasse *TCounter* zu ergänzen: Die Methoden *Click* und *SetCount* sollen in der neuen Klasse überschrieben werden.

Das folgende Klassendiagramm zeigt die Vererbungsbeziehung zwischen der ursprünglichen Klasse *TCounter* und der abgeleiteten Klasse *TModCounter*. Dabei werden in der abgeleiteten Klasse nur die neu hinzugekommenen Eigenschaften und die veränderten geerbten Eigenschaften aufgeführt.

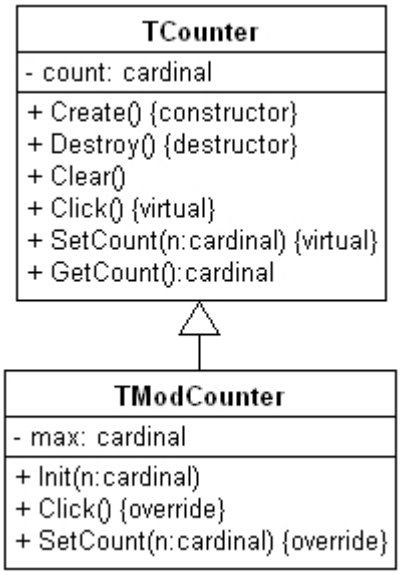


Abb. 3.4: Vererbungsbeziehung

Im Klassendiagramm sind die beiden Methoden *Click* und *SetCount* der ursprünglichen Klasse *TCounter* mit dem Zusatz *virtual* gekennzeichnet, in der abgeleiteten Klasse *TModCounter* erhalten die beiden entsprechenden Methoden den Zusatz *override*. Der Zusatz *virtual* ändert die ursprünglich statischen Methoden *Click* und *SetCount* in virtuelle Methoden, dies bedeutet: Diese Methoden können in abgeleiteten Klassen überschrieben werden (Schlüsselwort *override*), so dass die gleichnamigen überschriebenen Methoden in der Unterklasse nun an die Stelle der geerbten Methoden treten.

Prinzipiell ist es auch möglich auf die beiden Zusätze *virtual* und *override* zu verzichten. Die namensgleichen statischen Methoden der Unterklasse ersetzen dann die entsprechenden statischen Methoden aus der Oberklasse. Der deklarierte Typ der Objektvariablen in der Anwendung bestimmt in diesem Fall, welche der beiden Methoden aktiviert wird (Kompilationszeit-Polymorphie), bei virtuellen Methoden wird die Entscheidung über den zur Laufzeit des Programms aktuellen Typ getroffen (Laufzeit-Polymorphie).

Polymorphie wird im Lehrplan für die Jgst. 12 als fakultativer Inhalt aufgeführt, als mögliches Beispiel wird die Manipulation graphischer Objekte genannt. Das Konzept der Polymorphie kann erst durch den Einsatz virtueller Methoden effizient realisiert werden. Für Schüler sind die Vorteile virtueller Methoden zu diesem frühen Zeitpunkt sicher nicht so ohne weiteres einzusehen. Man muss sich dabei auch fragen, inwiefern es sinnvoll ist, die Schüler von Beginn an mit zu vielen Begriffen zu konfrontieren. Soll der Unterschied zwischen statischer und dynamischer Bindung dennoch schon an dieser Stelle etwas genauer betrachtet werden, so kann dies an einem einfachen Programm demonstriert werden, im Anhang befindet sich ein Arbeitsblatt[5], das zu diesem Zweck eingesetzt werden kann.

Generell ist es sinnvoll, Klassenmethoden immer nach folgender Regel zu erstellen: Bei Klassen, die durch Vererbung erweiterbar sein sollen, werden Methoden als virtuelle Methoden definiert, so dass sie in Unterklassen überschreibbar sind. Nach diesem Prinzip wird im weiteren vorgegangen.

Die Deklaration der Klasse *TModCounter* sieht nun wie folgt aus:

```
TModCounter = class(TCounter)                  //Nachkomme von TCounter
private
  max: cardinal;                               //Zählergrenze
public
  procedure Init(n: cardinal);                 //Zählergrenze festlegen
  procedure Click; override;                   //Click wird überschrieben
  procedure SetCount(n: cardinal); override;   //SetCount wird überschrieben
end;
```

Es schließt sich die Definition der Methoden im Implementations-Abschnitt an:

```
procedure TModCounter.Init(n: cardinal);
begin
  max:= n;
end;

procedure TModCounter.Click;
begin
  inherited Click;            //Aufruf der gleichnamigen Methode von TCounter
  if GetCount >= max then Clear;
end;

procedure TModCounter.SetCount(n: cardinal);
begin
  inherited SetCount(n);      //Aufruf der gleichnamigen Methode von TCounter
  if GetCount >= max then Clear;
end;
```

Die Angabe *inherited* in den überschriebenen Methoden bewirkt, dass zunächst die gleichnamigen Methoden der Oberklasse aufgerufen werden, anschließend wird überprüft, ob die Zählergrenze erreicht wurde und ggf. die folgenden Anweisungen ausgeführt.

An dieser Stelle sollte noch einmal die Bedeutung privater Elemente betont werden: In der Klasse *TModCounter* darf kein unmittelbarer Zugriff auf das Attribut *count* der Oberklasse

[5] Siehe Anhang

erfolgen, dieses ist eben privat, also nur für die Klasse *TCounter* selbst sichtbar und benutzbar. In der abgeleiteten Klasse darf auf den Zählerstand daher nur über die dafür zur Verfügung stehenden Methoden zugegriffen werden.

Leider ist es in Delphi möglich, private Elemente aus Unterklassen heraus auch dann direkt anzusprechen, wenn sich die Klassen in derselben Unit befinden. So kann es passieren, dass die Schüler die oben genannte Regel nicht einhalten, ohne dass der Compiler darauf reagiert. Daher ist es besonders wichtig, den Schülern die Bedeutung der verschiedenen Schutzklassen klar zu machen. Eine Alternative könnte es sein, jede Klasse in eine eigene Unit zu schreiben. Bei zunehmender Anzahl der Klassen führt das allerdings schnell zu Unübersichtlichkeit. Trotzdem ist die Variante zumindest an dieser Stelle vielleicht sinnvoll, die Bedeutung der Schutzklassen könnte dann nämlich – möglicherweise motiviert durch einen entsprechenden Fehler eines Schülers – im Unterricht von selbst noch einmal zum Thema werden.

Auch bei der neuen Klasse *TModCounter* sollte die Funktionsfähigkeit zum Abschluss wieder durch eine geeignete Testanwendung überprüft werden.

3.3.4 Zusammensetzen der Zählerobjekte zu einer größeren Einheit: Die Uhr

Im nächsten Schritt wird nun endlich das eigentliche Ziel in Angriff genommen, die Uhr. Ein vorerst vereinfachtes Modell sieht vor, dass sie zunächst nicht (eigenständig) laufen muss. Als Beispiel könnte man sich eine Uhr vorstellen, die den Beginn der nächsten Vorstellung in einem Kino anzeigt.

Aus den ursprünglichen Überlegungen ergibt sich, dass die Uhr als grundlegende Elemente mehrere Zähler besitzt und zwar jeweils einen für das Zählen der Sekunden, Minuten und Stunden. Dafür eignen sich am besten Zählerobjekte aus der Klasse *TModCounter*.

Die Analyse der für eine Uhrenklasse relevanten Eigenschaften ergibt nun z. B. folgendes Bild:

- Die Uhr besitzt drei Zählerobjekte als Stunden-, Minuten- und Sekundenzähler.
- Stunden, Minuten und Sekunden können gesetzt werden.
- Stunden, Minuten und Sekunden können ausgelesen werden.
- Die Uhrzeit kann schrittweise erhöht werden.

Auf die letztgenannte Methode könnte (zu diesem Zeitpunkt) allerdings durchaus auch verzichtet werden.

Beim Entwurf der Klasse *TModCounter* haben die Schüler die Vererbung als einen Beziehungstyp zwischen Klassen kennen gelernt. An dieser Stelle nun geht es um eine andere Form der Beziehung: Die Klasse *TClock* (Uhrenklasse) „besitzt“ drei Zählerobjekte, hier handelt es sich um eine Nutzungsbeziehung. Die Aggregation (eine Beziehung, bei der ein Objekt Teil eines anderen ist) konnten die Schüler bereits im ersten Abschnitt kennen lernen, nämlich als Beziehung zwischen der Fachklasse Zähler und der GUI-Klasse der Anwendungsoberfläche. Spätestens an dieser Stelle sollte jedoch dieser Beziehungstyp thematisiert werden.

Die Zählerobjekte sind hier existenzabhängig vom Objekt Uhr. Das Erzeugen der Uhr leitet automatisch das Erzeugen der Zählerobjekte ein, wird die Uhr gelöscht, so werden auch die Zählerobjekte gelöscht. Anders ausgedrückt: Die Uhr ist erst mit Stunden-, Minuten- und Sekundenzähler komplett, sie müssen also mit Erzeugen der Uhr automatisch miterzeugt werden. Wird die Uhr vernichtet, so gilt das entsprechende.

Diese Form der Aggregation, in der die Existenz der Teilobjekte untrennbar mit der Existenz des Aggregats verbunden ist, wird als Komposition bezeichnet.

Die Modellierung der Uhrenklasse und ihre Beziehung zur Klasse *TModCounter* wird im folgenden Diagramm verdeutlicht.

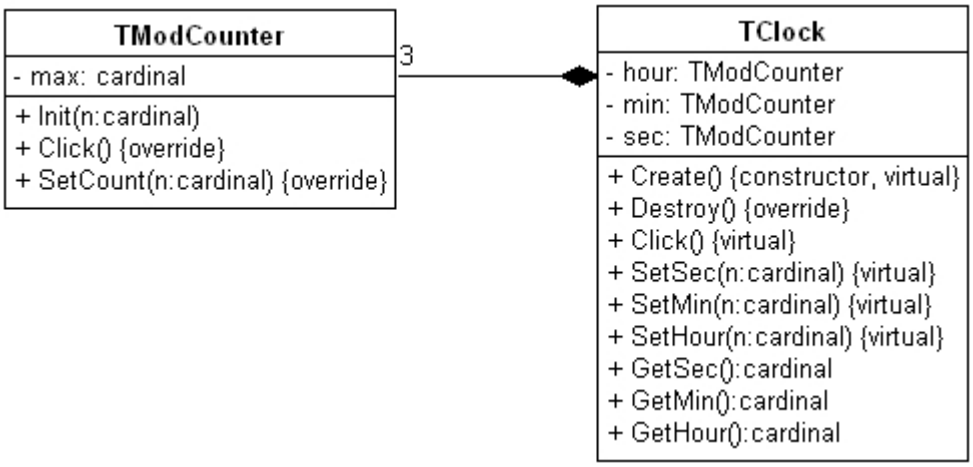


Abb. 3.5: Beziehung zwischen Uhrenklasse und Zählerklasse

Der Konstruktor der Klasse *TClock* sorgt für die Erzeugung der drei Zähler, indem er die entsprechende Konstruktormethode der Klasse *TModCounter* aufruft, außerdem wird jeweils die Zählergrenze festgelegt. Im Destruktor wird sichergestellt, dass die Zählerobjekte gelöscht werden, bevor das Uhrenobjekt selbst gelöscht wird.

Der Konstruktor *Create*, die Prozedur *Click* und die jeweiligen Methoden zum Setzen der Zeit werden als virtuell definiert, da nicht auszuschließen ist, dass sie in weiteren Unterklassen überschrieben werden müssen. Bei den Methoden zum Auslesen der Zeit wird dies sicher nicht der Fall sein (gleichwohl schadet es nichts, auch diese Methoden als virtuell zu definieren).

Die Deklaration der Uhrenklasse sieht entsprechend wie folgt aus:

```
TClock = class
private
  sec: TModCounter;
  min: TModCounter;
  hour: TModCounter;
public
  constructor Create; virtual;
  destructor Destroy; override;
  procedure Click; virtual;
  procedure SetSec(n: cardinal); virtual;
  procedure SetMin(n: cardinal); virtual;
  procedure SetHour(n: cardinal); virtual;
  function GetSec: cardinal;
  function GetMin: cardinal;
  function GetHour: cardinal;
end;
```

Es folgt die Implementation der Uhrenklasse:

```
constructor TClock.Create;
begin
  hour:= TModCounter.Create;
  hour.Init(24);
  min:= TModCounter.Create;
  min.Init(60);
  sec:= TModCounter.Create;
  sec.Init(60);
end;
```

```
destructor TClock.Destroy;
begin
  hour.Free; min.Free; sec.Free; //die Zählerobjekte werden zuerst vernichtet
  inherited Destroy;
end;

procedure TClock.Click;
begin
  sec.Click;                     //die Uhrzeit wird sekundenweise erhöht
  if sec.GetCount = 0 then       //Zählergrenze wurde überschritten, daher...
  begin
    min.Click;                   //... eine weitere Minute ist abgelaufen
    if min.GetCount = 0 then     //Zählergrenze wurde überschritten, daher...
      hour.Click;                //... eine weitere Stunde ist abgelaufen
  end;
end;

procedure TClock.SetSec(n: cardinal);
begin
  sec.SetCount(n);               //Aufruf der entsprechenden Methode der
end;                             //Zählerklasse

... //SetMin, SetHour entsprechend

function TClock.GetSec: cardinal;
begin
  GetSec:= sec.GetCount;         //Aufruf der entsprechenden Methode der
end;                             //Zählerklasse

... //GetMin, GetHour entsprechend
```

In der Destruktormethode wurde für das Löschen der Zählerobjekte die Methode *Free* eingesetzt. Es handelt sich dabei um eine von *TObject* geerbte Methode. Der Vorteil dieser Methode liegt darin, dass sie zunächst überprüft, ob das Objekt überhaupt instanziiert wurde. Wenn die Objektreferenz nicht *nil* ist, wird *Destroy* (der eigentliche Destruktor) automatisch aufgerufen.

Eine mögliche Anwendungsoberfläche zum Testen der Uhr könnte so aussehen:

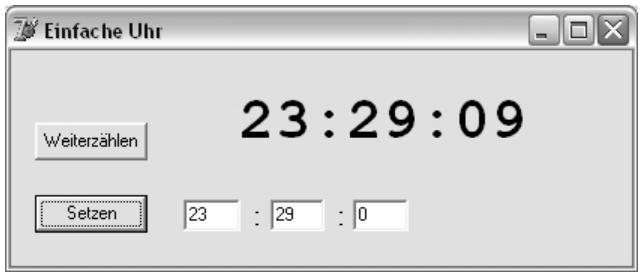


Abb. 3.6: Testanwendung Uhr

Zur Darstellung der Uhrzeit in der Anwendung dient hier ein einfaches Label, das bei jeder Änderung der Uhrzeit – durch Aufruf der Prozedur *Write* – aktualisiert wird.

```
procedure TForm1.Write;
var hour, min, sec: string;
begin
  hour:= IntToStr(Uhr.GetHour);
  min:= IntToStr(Uhr.GetMin);
  sec:= IntToStr(Uhr.GetSec);
  if length(hour) = 1 then hour:= '0' + hour;
  if length(min) = 1 then min:= '0' + min;
  if length(sec) = 1 then sec:= '0' + sec;
  LZaehler.Caption:= hour + ':' + min + ':' + sec;
end;
```

3.3.5 Die Uhr lernt laufen: Ein Timer kommt hinzu

Im nächsten Schritt soll die Uhr selbständig laufen. Dazu wird die Timerkomponente von Delphi eingesetzt. Die „Uhr mit Timer“ erhält ein Objektattribut *cTime*. Im überschriebenen Konstruktor wird der Timer erzeugt und das Timerintervall auf 1000 (Millisekunden) gesetzt. Zusätzlich soll es zwei neue Methoden *Start* und *Stop* geben, mit deren Hilfe der Timer aktiviert bzw. inaktiviert wird. Wenn der Timer aktiviert ist, so soll er die Uhrzeit im Sekundentakt erhöhen, dem *OnTimer*-Ereignis wird daher die private Methode *MoveTime* zugeordnet, die zwecks Typkompatibilität den Parameter *Sender* von *TObject* enthalten muss und die in der Vaterklasse definierte Methode *Click* aufruft.

Die Klasse „Uhr mit Timer“ bekommt den Namen *TClockWithTimer*, das zugehörige Klassendiagramm sieht wie folgt aus:

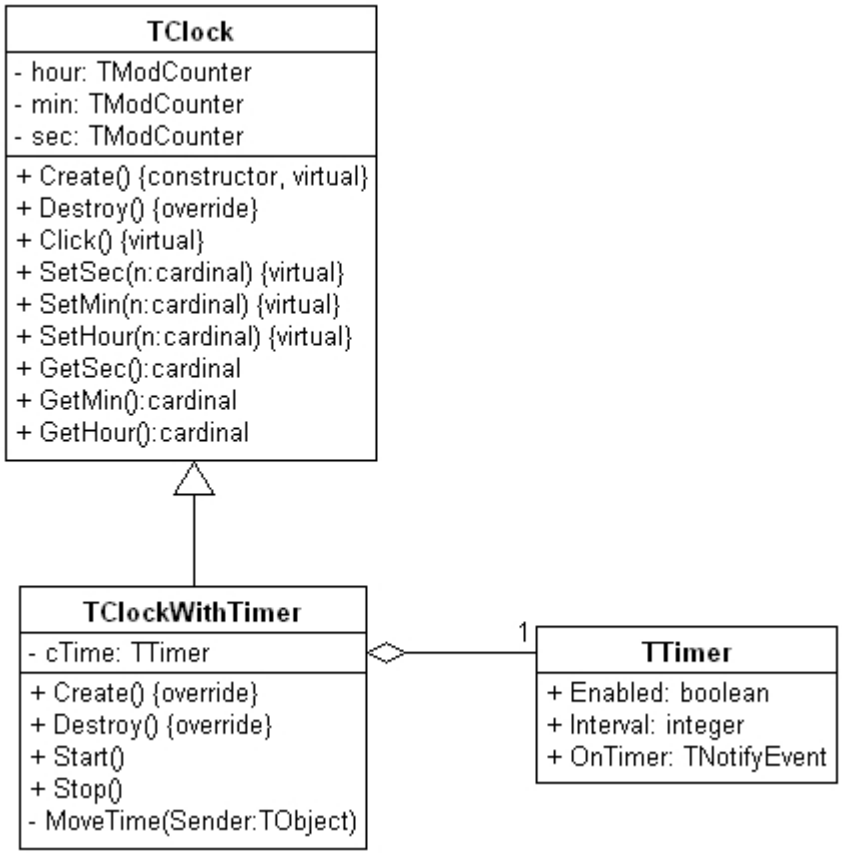


Abb. 3.7: Die Klasse „Uhr mit Timer“

Damit die Timerkomponente in der Uhrenklasse zur Verfügung steht, muss zuvor über die *Uses*-Klausel die Delphi-Unit *ExtCtrls* eingebunden werden.

Die Deklaration der Klasse *TClockWithTimer* sieht anschließend wie folgt aus:

```
TClockWithTimer = class(TClock)
private
  cTime: TTimer;
  procedure MoveTime(Sender: TObject); //ruft die geerbte Methode Click auf
public
  constructor Create; override;
  destructor Destroy; override;
  procedure Start;                     //aktiviert den Timer
  procedure Stop;                      //deaktiviert den Timer
end;
```

Für die Erzeugung eines Exemplars der Timerklasse erwartet der Konstruktor die Angabe eines Eigentümers des Timerobjekts, dieser wird dem Konstruktor als Parameter mitgegeben und muss vom Typ *TComponent* sein. Die Klasse *TClockWithTimer* ist jedoch (über die Zwischenstufe *TClock*) ein Abkömmling von *TObject*, so dass sie nicht Besitzer einer Komponente sein kann. Ein Ausweg ist hier daher, dem Parameter *Owner* den Wert *nil* zuzuweisen.

```
constructor TClockWithTimer.Create;
begin
  inherited Create;
  cTime:= TTimer.Create(nil);        //Referenz auf den Besitzer ist nil
  cTime.Interval:= 1000;
  cTime.Enabled:= false;
  cTime.OnTimer:= MoveTime;
end;

destructor TClockWithTimer.Destroy;
begin
  cTime.Free;
  inherited Destroy;
end;

procedure TClockWithTimer.Start;
begin
  cTime.Enabled:= true;
end;

procedure TClockWithTimer.Stop;
begin
  cTime.Enabled:= false;
end;

procedure TClockWithTimer.MoveTime(Sender: TObject);
begin
  Click;
end;
```

Testet man die neue Uhr in einer Anwendung – die im Wesentlichen der Testanwendung des vorherigen Beispiels entspricht – ergibt sich folgendes Problem: Die neue Uhr läuft zwar selbständig, allerdings wird die Anzeige auf der Oberfläche nicht automatisch aktualisiert. Über einen entsprechenden Button (Auslesen der momentanen Uhrzeit) kann zwar die jeweilige Uhrzeit ermittelt werden, aber in einem weiteren Entwicklungsschritt der Uhr wäre es natürlich wünschenswert, dass eine Methode zur Verfügung gestellt wird, welche die automatische Aktualisierung der Anzeige ermöglicht, falls sich die Uhrzeit geändert hat. Dies soll im nächsten Schritt realisiert werden.

3.3.6 Das Problem der Kommunikation zwischen Fachklasse und GUI-Klasse

Im hier verfolgten Entwurfsprinzip sind das Fachkonzept und die GUI-Klasse vollständig voneinander getrennte Einheiten. Die GUI-Klasse kennt zwar die Fachklasse (sie enthält ein Objekt der Fachklasse) umgekehrt weiß die Fachklasse jedoch nichts über die Anwendung, in der ein entsprechendes Objektexemplar erzeugt wird. Es ergibt sich also die Frage, wie das Uhrenobjekt die Anwendung über Änderungen informieren kann (sprich: Wie erfährt die Anwendung, dass sich die Uhrzeit geändert hat, so dass sie ihre Darstellung auf der Oberfläche entsprechend anpassen kann?).

Eine neue Uhrenklasse *TInfoClock* soll dieses Problem lösen: Wenn das Ereignis „Uhrzeit hat sich geändert“ eintritt, so soll eine Methode der Anwendung aufgerufen werden, die für die Aktualisierung der Ansicht sorgt. Zunächst muss ein Methodenzeigertyp vereinbart werden, er enthält eine Prozedur ohne Parameterliste. *TInfoClock* erhält eine Referenzvariable vom

Typ des Methodenzeigers. In der Anwendung wird später der Methodenzeiger mit einer passenden Prozedur der GUI-Klasse verknüpft.

Eine Änderung der Uhrzeit tritt bei Aufruf der Prozedur *Click* bzw. beim Setzen der Uhrzeit (*SetHour*, *SetMin* und *SetSec*) ein. Diese vier Methoden müssen daher in *TInfoClock* überschrieben werden und dafür sorgen, dass das entsprechende Ereignis ausgelöst wird.

Die nachstehende Grafik zeigt die Vererbungsbeziehungen der bisherigen Uhrenklassen. Die Attribute und Methoden der Vorfahrklassen von *TInfoClock* werden dabei nur ausschnittsweise dargestellt. Es sei an dieser Stelle noch angemerkt, dass es sicher nicht notwendig ist, die Klasse *TInfoClock* als eigenständige Unterklasse zu entwickeln. Wem die so erzeugte Klassenhierarchie ein wenig zu künstlich erscheint (man mag sich z. B. fragen, ob die Klasse „Uhr mit Timer“, wie sie zuvor entwickelt wurde, eine sinnvolle eigenständige Existenzberechtigung hat…), der kann an dieser Stelle natürlich auch darauf verzichten, eine zusätzliche Hierarchieebene einzufügen und stattdessen die ursprüngliche „Uhr mit Timer“ einfach um die hier beschriebenen Erweiterungen ergänzen.

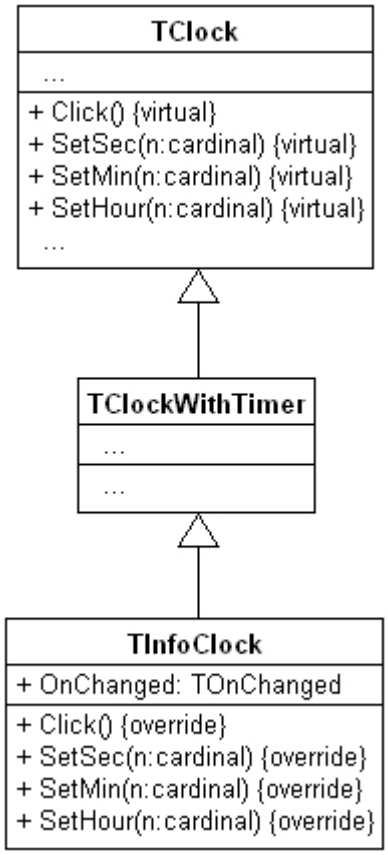


Abb. 3.8: Klassenhierarchie

Die folgende Auflistung zeigt die Vereinbarung des Methodenzeigertyps und der Klasse *TInfoClock*.

```
TOnChanged = procedure of object;     //Methodenzeigertyp
TInfoClock = class(TClockWithTimer)
public
  OnChanged: TOnChanged;              //Variable vom Typ des Methodenzeigers
  procedure Click; override;
  procedure SetSec(n: cardinal); override;
  procedure SetMin(n: cardinal); override;
  procedure SetHour(n: cardinal); override;
end;
```

Es folgt die Implementation der Methoden:

```
procedure TInfoClock.Click;
begin
  inherited Click;
  if assigned(OnChanged) then OnChanged;      //Ereignis auslösen
end;
procedure TInfoClock.SetSec(n: cardinal);
begin
  inherited SetSec(n);
  if assigned(OnChanged) then OnChanged;      //Ereignis auslösen
end;

... //SetMin, SetHour entsprechend
```

In der GUI-Klasse der Anwendung muss nun der Methodenzeiger mit einer Ereignisbehandlungsroutine verknüpft werden. Die entsprechende Prozedur verfügt über dieselbe Anzahl und dieselben Typen von Parametern, wie im Methodenzeigertyp *TOnChanged* definiert. In diesem Fall heißt das also, dass es sich um eine parameterlose Prozedur handeln muss. Die Prozedur *Write* (siehe S. 21) erfüllt diese Voraussetzung, sie wird *OnChanged* zugewiesen, so dass das Uhrenobjekt nun nach jeder Änderung seiner Daten (also der Uhrzeit) die Methode *Write* aufruft und die Formularansicht wird aktualisiert.

Die folgende Auflistung zeigt die Vereinbarungen in der GUI-Klasse:

```
type
  TForm1 = class(TForm)
    ...
  private
    { Private-Deklarationen }
    Uhr: TInfoClock;
    procedure Write;             //Ereignismethode
  public
    ...
  end;
```

… und (ausschnittsweise) die Implementierung:

```
procedure TForm1. FormCreate(Sender: TObject);
begin
  Uhr:= TInfoClock.Create;
  Write;
  Uhr.OnChanged:= Write;         //Verknüpfung mit der Ereignismethode
  Uhr.Start;
end;

procedure TForm1.Write;          //wird vom Uhrenobjekt aufgerufen
begin
  ...
end;
```

3.3.7 Erweiterung der Uhrenklasse: Spezielle Uhren

Die letzte Uhrenklasse bildet den Ausgangspunkt für die Entwicklung weiterer, spezialisierter Uhrentypen. Während bis zu diesem Zeitpunkt im Unterricht sicherlich vieles in Gemeinschaftsarbeit entwickelt werden muss, bietet es sich spätestens an dieser Stelle an, die Schüler nunmehr arbeitsteilig in Gruppen weitere Unterklassen für Uhren mit besonderer Funktionalität ausarbeiten zu lassen. Denkbare Erweiterungen, die von den Schülern selbständig ausgearbeitet werden können, sollen nachfolgend vorgestellt werden. Dabei muss jedoch berücksichtigt werden, dass die Schwierigkeitsgrade der Aufgabenstellungen durchaus sehr verschieden

sind. In einer heterogenen Lerngruppe bietet dies die Möglichkeit, differenziert zu arbeiten und die Schüler Aufträge gemäß ihrer Leistungsfähigkeit bearbeiten zu lassen. Aber es ist natürlich auch interessant, eine einheitliche Aufgabenstellung zu wählen – die in Kleingruppen bearbeitet wird – und im Anschluss die sicher unterschiedlichen Ergebnisse zu vergleichen.

In der hier vorliegenden Arbeit werden von den möglichen Erweiterungen zwei Beispiele vorgestellt: Das erste Beispiel ist ein Wecker, der sich durch zusätzliche Eigenschaften und Funktionen wie das Stellen der Weckzeit und die Aktivierung bzw. Deaktivierung der Alarmfunktion auszeichnet. Bei dem zweiten Beispiel handelt es sich nicht um eine Erweiterung der Klassenhierarchie der Uhrenklasse: Es wird die Entwicklung einer (eigenständigen) Klasse *TAnalog* vorgestellt, die später eine analoge Darstellung der Zeitanzeige in einer Anwendung ermöglichen soll.

Für den Unterricht bieten sich z. B. noch die folgenden Uhrentypen an:

- Eine Eieruhr: Dabei handelt es sich eigentlich um einen Kurzzeitwecker. Wie der Wecker auch, verfügt die Eieruhr über eine Einstellung der Alarmzeit und eine Funktion zum Auslösen des Alarms. Die Eieruhr stoppt jedoch nach Ablauf der gewählten Alarmzeit, die zudem begrenzt ist (i. d. R. im Minutenbereich).

- Eine Stoppuhr: Neben Minuten und Sekunden werden an der Stoppuhr auch Zehntel- und evtl. Hundertstelsekunden angezeigt. Neben einer Start-, Stop- und Rückstellfunktion kann die Stoppuhr noch über zusätzliche Möglichkeiten verfügen, bspw. die Anzeige einer Zwischenzeit. Bei der Umsetzung der Stoppuhr in Delphi wird allerdings die Unzulänglichkeit des Delphi-Timers deutlich: Schon im Zehntelsekundenbereich macht sich schnell eine deutliche Differenz zwischen der tatsächlich abgelaufenen Zeit und der mit Hilfe des Timers gemessenen Zeit bemerkbar.

- Eine Glasenuhr: Dieser Uhrentyp ist Schülern vermutlich nicht bekannt, er kommt aus der Schifffahrt und gibt durch Glockenschläge die Zeit an Bord bekannt. Die 24 Stunden eines Tages werden in gleichmäßige Abschnitte unterteilt (bspw. je vier Stunden, dies entspricht dem 3-Wachensystem, welches unter anderem auf deutschen Schiffen „traditionell“ gefahren wird). „Geglast“ wird im Halbstundentakt, dies bedeutet: Nach einer halben Stunde ertönt ein Glockenschlag, nach einer Stunde ein Doppelschlag, usw. bis zum Ablauf des Zeitabschnittes.

- Eine interessante Möglichkeit ist eine Schachuhr: Dabei handelt es sich eigentlich um zwei Uhren, die sich gegenseitig an- und abschalten. Das ganze funktioniert nach folgendem Prinzip: Wenn die Partie gestartet ist, läuft die Uhr desjenigen, der am Zug ist, los. Nachdem der Spieler seinen Zug ausgeführt hat stoppt er seine Uhr, gleichzeitig startet automatisch die Uhr des Gegners. Dabei hat jeder Spieler nur eine bestimmte Zeitspanne für die Partie zur Verfügung, die auf der Schachuhr zuvor eingestellt wird – je nach Schachform (z. B. Blitzschach oder Turnierschach) zwischen 5 Minuten und zweieinhalb Stunden. Das Ende der Bedenkzeit muss auf der Uhr natürlich angezeigt werden.

- Ein weiteres Beispiel ist die Entwicklung einer Weltzeituhr: Die Basisuhrzeit wird festgelegt (die Zeiten in den verschiedenen Zeitzonen sind heutzutage an die koordinierte Weltzeit – UTC – gekoppelt, die nach dem nullten Längengrad ausgerichtet ist), die Uhrzeiten der anderen Zeitzonen errechnen sich durch hinzufügen oder abziehen einer ganzzahligen Anzahl von Stunden. Die Modellierung einer Weltzeituhr ist sicher recht anspruchsvoll, es ist schwieriger als bei den anderen Uhrentypen eine Entscheidung darüber zu treffen, wie

das Uhrenmodell am besten repräsentiert wird: Zeigt die Weltzeituhr gleichzeitig die Uhrzeit in mehreren (allen) Zeitzonen an oder soll über eine Vorauswahl eine Zeitzone gewählt werden, deren Zeit dann ausgegeben wird? Wodurch wird die Auswahl repräsentiert (bspw. durch die Angabe von Längengraden, Ländernamen, ausgewählten Städtenamen)? Hier kann sicherlich viel kreatives Entwicklungspotenzial freigesetzt werden…

3.3.8 Ein Wecker mit 7-Segment-Anzeige

Bei dem hier zu entwickelnden Uhrentyp erfüllt der Zeitgeber einen speziellen zusätzlichen Zweck: Zu einer voreingestellten Uhrzeit gibt der Wecker ein Alarmzeichen. Grundsätzlich hat natürlich auch der Wecker alle Eigenschaften, die für eine Uhr typisch sind, ist also ein Vertreter der Klasse Uhr. Neben den typischen Eigenschaften einer Uhr besitzt der Wecker jedoch weitere Funktionalitäten, welche die Entwicklung einer eigenen Weckerklasse sinnvoll erscheinen lassen.

Die Weckerklasse zeichnet sich durch die folgenden zusätzlichen Eigenschaften aus:

- Die Weckzeit: Sie wird durch zwei Attribute für Stunde und Minute repräsentiert.
- Es gibt eine Methode zum Festlegen der Weckzeit.
- Es gibt eine Methode zum Auslesen der Weckzeit.
- In einer weiteren Methode wird überprüft, ob die Weckzeit erreicht wurde.

Eine weitere Eigenschaft „Alarmfunktion aktiviert“ wäre denkbar, in dem hier vorgestellten Beispiel wird jedoch darauf verzichtet. Die Anwendung fragt ggf. im entsprechenden Zustand regelmäßig das Erreichen der Weckzeit ab. Die Form der Weckbenachrichtigung soll ebenfalls durch die Anwendung bestimmt werden (im Beispiel wird dies ein einfacher Standard-Signalton sein).

Im folgenden Bild wird das Klassendiagramm für die Weckerklasse gezeigt. Dabei wird zusätzlich nur der direkte Vorfahre der Weckerklasse (*TInfoClock*) mit angegeben.

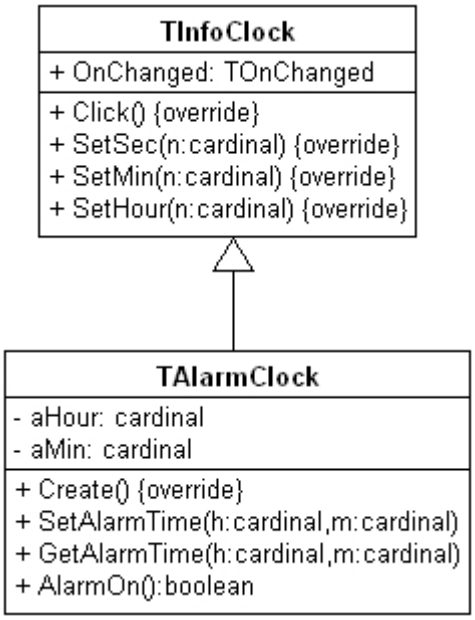


Abb. 3.9: Weckerklasse

Neben den drei hinzugekommenen Methoden wird auch die Konstruktormethode neu definiert. Dabei werden die Attribute für die Weckzeit vorbelegt. Für das Setzen und Auslesen der

Weckzeit wird diesmal darauf verzichtet, für jede Zeiteinheit eine eigene Methode zu definieren. Selbstverständlich kann man hier natürlich auch einen anderen Weg gehen.

Die Deklaration der Weckerklasse *TAlarmClock* sieht anschließend wie folgt aus:

```
TAlarmClock = class(TInfoClock)
private
  aHour, aMin: cardinal;
public
  constructor Create; override;
  procedure SetAlarmTime(h, m: cardinal);
  procedure GetAlarmTime(var h, m: cardinal);
  function AlarmOn: boolean;
end;
```

Es folgt die Implementation der Weckerklasse:

```
constructor TAlarmClock.Create;
begin
  inherited Create;
  aHour:= 0; aMin:= 0;
end;

procedure TAlarmClock.SetAlarmTime(h, m: cardinal);
begin
  aHour:= h; aMin:= m;
end;

procedure TAlarmClock.GetAlarmTime(var h, m: cardinal);
begin
  h:= aHour; m:= aMin;
end;

function TAlarmClock.AlarmOn: boolean;
begin
  if (aHour = GetHour) and (aMin = GetMin) then AlarmOn:= true
  else AlarmOn:= false;
end;
```

Die Überschrift dieses Kapitels verspricht einen Wecker mit 7-Segment-Anzeige. Die Anzeigeart ist jedoch keine Eigenschaft der Weckerklasse, vielmehr soll im Folgenden eine etwas komplexere Anwendung vorgestellt werden, in der unter anderem auf dem Display die Uhrzeit des Weckers durch eine 7-Segment-Anzeige dargestellt wird.

Eine Anwendung für den Wecker soll hier etwa wie in folgendem Bild aussehen:

Abb. 3.10: Wecker

Delphi stellt standardmäßig keine Komponente zur Verfügung, die eine Darstellung der Uhrzeit als 7-Segment-Anzeige, wie hier vorgestellt ermöglicht. Dazu ist es notwendig, zusätzliche Komponenten zu installieren. Im Internet findet man ein reichliches Angebot an Delphi-Komponenten, die von begeisterten und engagierten Delphi-Programmierern stetig erweitert

werden, so dass auch die Suche nach einer frei verfügbaren Komponente für die 7-Segment-Anzeige problemlos zu einem Treffer führt[6].

Der Button „Weckzeit anzeigen“ bewirkt (solange die Maustaste gedrückt ist), dass im Display die eingestellte Weckzeit erscheint. Wird der Button losgelassen, so ändert sich die Anzeige wieder auf die aktuelle Uhrzeit. Das Setzen der Zeit bzw. die Einstellung der Weckzeit soll in der Anwendung über ein gesondertes Formular erfolgen. Das Formular wird bei Betätigen eines der beiden entsprechenden Button neu erzeugt und natürlich anschließend auch wieder vernichtet. Ein Diagramm, das die Beziehungen der in der Anwendung verwendeten Klassen untereinander beschreibt, sieht wie folgt aus:

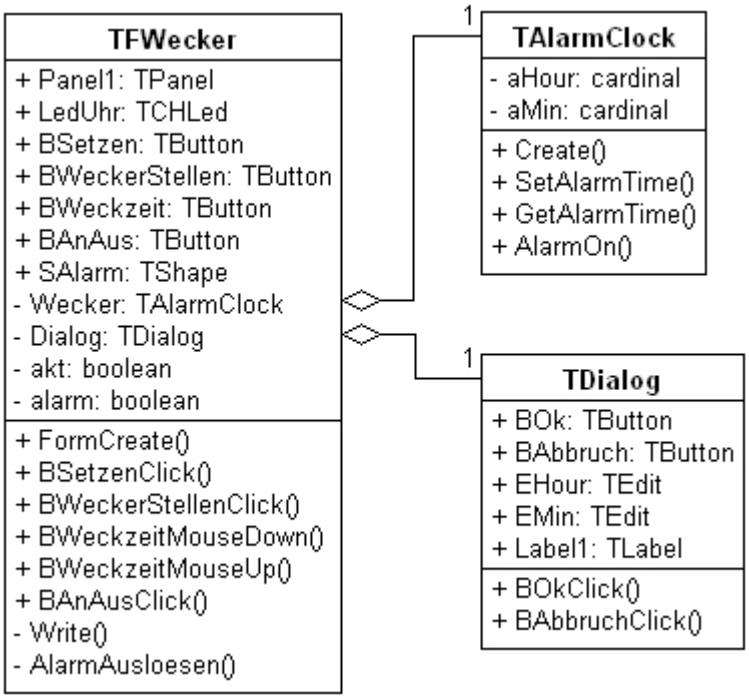


Abb. 3.11: Beziehungsdiagramm (ohne Komponentenklassen)

Genau wie *TFWecker* ist auch die Klasse *TDialog* ein Nachkomme der Klasse *TForm*. Um das Dialogfenster zu entwerfen, wird in Delphi zunächst ein neues Formular zum Projekt hinzugefügt, Delphi erzeugt automatisch die zugehörige Formular-Unit. Das Formular könnte bspw. wie in der folgenden Abbildung gestaltet werden.

Abb. 3.12: Dialogfenster

Wenn „Ok“ oder „Abbruch“ angeklickt wird, so wird das Formular wieder geschlossen. Zusätzlich wird in der Ereignisbehandlungsroutine für den Abbruch-Button die Button-Eigenschaft *Cancel* auf *true* gesetzt, darüber kann später im Hauptformular einfach überprüft werden, ob das Dialogformular über den Abbruch-Button verlassen wurde.

[6] Eine geeignete, frei verfügbare Komponentensammlung inklusive 7-Segment-Anzeige findet man im Internet bspw. unter http://www.blue-xplosion.de

Der Delphi-Automatismus sieht vor, dass bei Programmstart stets alle Formulare des Projekts im Speicher erzeugt werden, wobei zunächst nur das Hauptformular sichtbar ist. Um diesen Automatismus aufzuheben – d. h. das Dialogfenster wird erst dann im Speicher erzeugt und sichtbar gemacht, wenn es benötigt wird – sind folgende Schritte notwendig:

- Im Hauptformular wird eine Objektvariable des Typs *TDialog* definiert.
- Die in der Unit für das Dialogfenster automatisch erzeugte Objektvariable wird gelöscht.
- Über den Menübefehl *Projekt/Optionen* wird die automatische Formularerstellung abgeschaltet, das Dialogformular wird in das rechte Listenfeld übertragen, lediglich das Hauptformular verbleibt im linken Listenfeld.

Im Folgenden werden die privaten Vereinbarungen in der GUI-Klasse gezeigt:

```
type
  TFWecker = class(TForm)
    ...
  private
    Wecker: TAlarmClock;
    Dialog: TDialog;
    akt: boolean;   //wenn Weckzeit-Button gedrückt ist akt=false, sonst true
    alarm: boolean; //Alarmfunktion einschalten
    procedure Write;
    procedure AlarmAusloesen;
  end;
```

Es folgt (ausschnittsweise) die Implementierung:

```
procedure TFWecker.Write;               //wird vom Uhrenobjekt aufgerufen
var hour, min: string;
begin
  if akt = true then                    //Weckzeit-Button ist nicht gedrückt
  begin
    hour:= Format('%2.2d', [Wecker.GetHour]);  //2-ziffrige Anzeige
    min:= Format('%2.2d', [Wecker.GetMin]);
    LedUhr.Value:= hour + min;
  end;
  if alarm = true then AlarmAusloesen;
end;

procedure TFWecker.AlarmAusloesen;
begin
  if Wecker.AlarmOn = true then beep;
end;

procedure TFWecker.BSetzenClick(Sender: TObject);
var hour, min: cardinal;
begin
  Dialog:= TDialog.Create(FWecker);     //Dialogformular erzeugen
  Dialog.Caption:= 'Uhrzeit stellen';
  Dialog.ShowModal;                     //modales Fenster
  if not Dialog.BAbbruch.Cancel then    //Abbruch-Button wurde nicht gedrückt
  begin
    hour:= StrToInt(Dialog.EHour.Text); //Zeit setzen
    min:= StrToInt(Dialog.EMin.Text);
    with Wecker do begin
      SetHour(hour); SetMin(min)
    end;
  end;
  Dialog.Free;                          //Dialogformular löschen
end;
```

Das Dialogfenster ist ein modales Fenster, d. h. es muss zunächst geschlossen werden, bevor mit dem Hauptformular weitergearbeitet werden kann. Allerdings wird das Dialogformular dabei nicht vernichtet, es verbleibt weiterhin im Speicher, daher kann das Hauptformular

problemlos die benötigten Werte auslesen. Erst anschließend wird das Dialogfenster auch aus dem Speicher entfernt.

3.3.9 Eine Analoguhr

Im letzten Beispiel soll eine Klasse entwickelt werden, die es ermöglicht, Uhren mit analoger Zeitanzeige darzustellen. Dabei handelt es sich nicht um eine Uhrenklasse sondern um eine eigenständige Klasse. In der Anwendung muss später sowohl eine Uhreninstanz als auch eine Instanz der Analogklasse erzeugt werden.

Die Klasse *TAnalog* soll ein Nachkomme der Klasse *TImage* sein. Die von *TImage* geerbte Konstruktormethode wird ergänzt, dabei soll zusätzlich die Position und die Breite des Objektes festgelegt werden. Die Analog-Klasse stellt sowohl Methoden zur Ermittlung der Position (Winkelausrichtung) der verschiedenen Zeiger, als auch Zeichenmethoden für das Zifferblatt und die Zeiger bereit.

Selbstverständlich könnte man sich auch vorstellen, die Darstellung des Zifferblattes und der Zeiger in unterschiedlichen Klassen zu realisieren und mit verschiedenen Gestaltungsmöglichkeiten zu versehen. In der hier vorgestellten Modellierung wird die Analoganzeige als Einheit aus Zifferblatt und Zeigern verwirklicht und enthält nur begrenzte Möglichkeiten zur Variation. Wie in den vorangegangenen Beispielen ist das hier vorgestellte Modell daher nur als mögliche Variante zu begreifen, ausgefeiltere Versionen sind ohne weiteres denkbar.

Die Klasse *TAnalog* soll im Einzelnen über folgende Eigenschaften verfügen:

- Position (Mittelpunkt) und Radius des Ziffernblattes.
- Methoden zur Bestimmung der (Winkel-)Position der Zeiger.
- Methoden zum Zeichnen des Zifferblattes und der Zeiger.

Als Zeiger werden Sekunden-, Minuten-, Stundenzeiger und ein Zeiger für die Alarmzeit zur Verfügung gestellt. Das Klassendiagramm für die Klasse *TAnalog* sieht wie folgt aus:

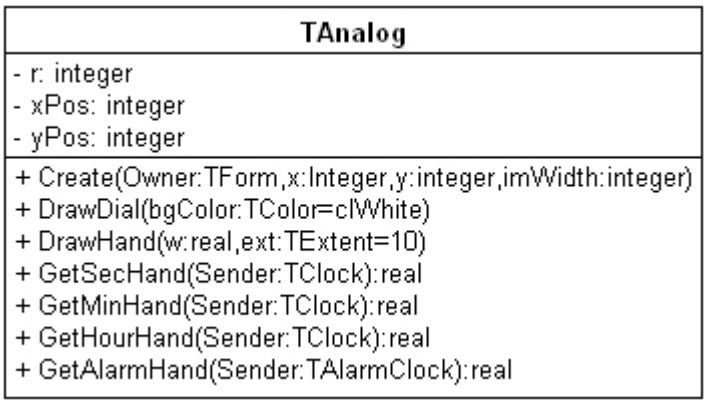


Abb. 3.13 Klasse TAnalog

Die *Get*-Methoden liefern die Winkelstellung der verschiedenen Zeiger. Dazu erhalten sie als Parameter einen Verweis auf das Uhrenobjekt. In der Funktion kann nun die aktuelle Uhrzeit ausgelesen und der passende Zeigerwinkel bestimmt werden. Die Funktion *GetAlarmHand* muss konsequenterweise natürlich als Parameter einen Verweis auf ein Weckerobjekt erhalten. Die Methode *DrawDial* zeichnet das Zifferblatt. Dabei kann zusätzlich eine Farbe angegeben werden, wird darauf verzichtet, so erhält das Zifferblatt einen weißen Hintergrund. Mit Hilfe der Methode *DrawHand* werden die Zeiger gezeichnet. Sie erhält zwei Parameter, der erste steht für den Zeigerwinkel (mit Hilfe der *Get*-Methoden ermittelt), über den zweiten

Parameter kann die Zeigerlänge beeinflusst werden: Der Datentyp *TExtent* ist ein – zuvor definierter – Unterbereichs- oder Teilbereichstyp, der *Integer*-Werte nur von 1 bis 10 zulässt. Die Zahl 1 repräsentiert die kürzestmögliche Länge, 10 entsprechend die Maximallänge.

In der Unit der Analogklasse müssen über die *Uses*-Klausel zusätzlich zur Unit mit den Uhrenklassen auch die Delphi-Units *Forms* (Übergabe des *Owner* vom Typ *TForm* im Konstruktor), *ExtCtrls* (wegen *TImage*) und *Graphics* (wegen *TColor*) angegeben werden.

Es folgt – ausschnittsweise – die Implementation der Klasse *TAnalog*:

```
type
  TExtent = 1..10;
  TAnalog = class(TImage)
  ...                       //siehe Klassendiagramm
  end;

implementation

constructor TAnalog.Create(Owner: TForm; x, y, imWidth: integer);
begin
  inherited Create(Owner);
  Parent:= Owner;
  Left:= x; Top:= y; Width:= imWidth; Height:= imWidth;
  r:= Width div 2;                      //Radius des Zifferblattes
  xPos:= r;                             //x-Koordinate Mittelpunkt
  yPos:= r;                             //y-Koordinate Mittelpunkt
end;

procedure TAnalog.DrawDial(bgColor: TColor = clWhite);
var i: integer; mult: real;
begin
  with Canvas do begin
    Brush.Color:= bgColor;
    Pen.Width:= 3;
    Ellipse(xPos-r, yPos-r, xPos+r, yPos+r);
    for i:=1 to 60 do                   //Einteilung des Zifferblattes
    begin
      if (i mod 5)=0 then begin         //jeder 5. Zifferstrich
        Pen.Width:=2; mult:=0.9; end    //soll etwas breiter und länger werden
      else begin
        Pen.Width:=1; mult:=0.93; end;
      MoveTo(round(xPos+cos(i*Pi/30)*mult*r),
             round(yPos+sin(i*Pi/30)*mult*r));
      LineTo(round(xPos+cos(i*Pi/30)*r), round(yPos+sin(i*Pi/30)*r));
    end;
  end;
end;

procedure TAnalog.DrawHand(w: real; ext: TExtent = 10);
var xEPos, yEPos: integer;
begin
  xEPos:= xPos + round(cos(w)*r*ext/10*0.9);  //x-Koordinate Zeigerende
  yEPos:= yPos + round(sin(w)*r*ext/10*0.9);  //y-Koordinate Zeigerende
  Canvas.MoveTo(xPos, yPos);
  Canvas.LineTo(xEPos, yEPos);
end;

function TAnalog.GetSecHand(Sender: TClock): real;
var wSec: real;
begin
  wSec:= 3/2*Pi + Sender.GetSec*(Pi/30); //Winkelberechnung sekundengenau
  if wSec >= (2*Pi) then
    wSec:= wSec - 2*Pi;
  GetSecHand:= wSec;
end;

... //GetMinHand, GetHourHand, GetAlarmHand vergleichbar
```

Die folgende Abbildung zeigt die Darstellung der Analoguhr in einer Anwendung.

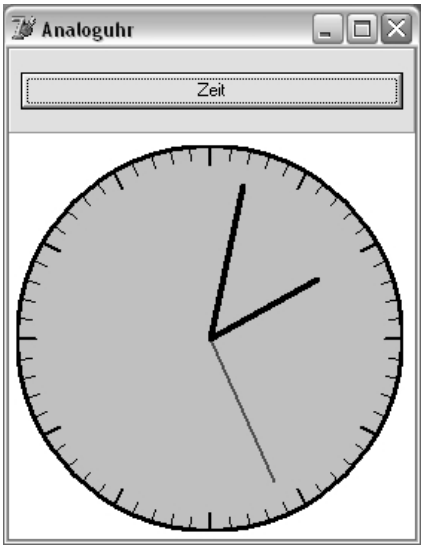


Abb. 3.14: Analoguhr

In der Anwendung wird in der Methode *FormCreate* sowohl eine Instanz der Uhrenklasse, als auch eine Instanz der Analogklasse erzeugt. Damit in der Darstellung die Zeiger korrekt aktualisiert werden, müssen bei jedem Zeichenvorgang zunächst die alten Zeiger gelöscht werden (Prozedur *Loeschen*), anschließend werden die Zeiger entsprechend der aktuellen Uhrzeit neu gezeichnet (Prozedur *NeuZeichnen*). Bei jedem Neuzeichnen werden die zuvor abgefragten Winkelstellungen der Zeiger in Hilfsvariablen gespeichert um für das Löschen zur Verfügung zu stehen. Beim Löschen werden die Zeiger einfach in der Farbe des Zifferblattes übermalt.

Die Implementation der drei Prozeduren wird im Folgenden ausschnittsweise vorgestellt.

```
procedure TFAnalogUhr.FormCreate(Sender: TObject);
begin
  Uhr:= TInfoClock.Create;
  Anzeige:= TAnalog.Create(FAnalogUhr, 5, 65, 260);
  Anzeige.DrawDial(clSilver);    //Zifferblatt zeichnen
  NeuZeichnen;                   //Zeiger (in Grundstellung) zeichnen
  Uhr.OnChanged:= Loeschen;      //bei Zeitänderung Loeschen aufrufen
  Uhr.Start;                     //Uhr starten
end;

procedure TFAnalogUhr.Loeschen;
begin
  with Anzeige do begin
    Canvas.Pen.Color:= clSilver; //Farbe des Zifferblattes
    Canvas.Pen.Width:= 2;
    DrawHand(SecZeiger, 9);      //DrawHand mit alter Zeigerstellung aufrufen
    Canvas.Pen.Width:= 4;
    DrawHand(MinZeiger, 9);
    DrawHand(StdZeiger, 7);
  end;
  NeuZeichnen;
end;

procedure TFAnalogUhr.NeuZeichnen;
begin
  with Anzeige do begin
    Canvas.Pen.Color:= clRed;    //Farbe für Sekundenzeiger
    Canvas.Pen.Width:= 2;        //Breite des Sekundenzeiger
    SecZeiger:= GetSecHand(Uhr); //Zeigerwinkel ermitteln
    DrawHand(SecZeiger, 9);      //Zeiger an (neuer) Position zeichnen
    ...      end; end;           //usw.
```

Auf dieselbe Art und Weise lässt sich nun auch ein Analogwecker realisieren.

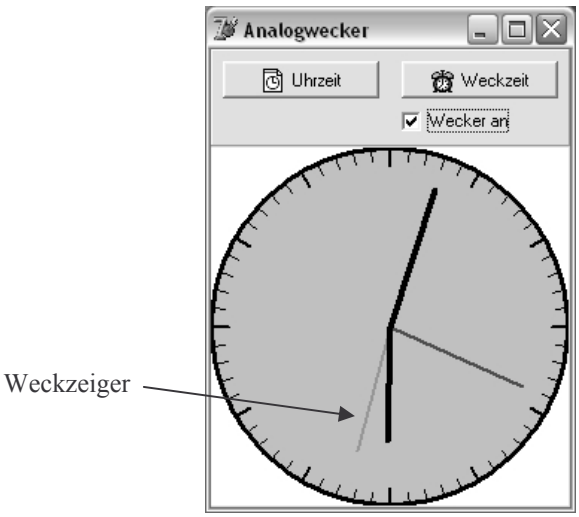


Abb. 3.15: Analogwecker

Ein Test ergibt allerdings, dass der Analogwecker noch verbesserungswürdig ist: Analoguhren haben eine 12-Stunden-Anzeige, insofern „klingelt“ ein Analogwecker normalerweise im Verlaufe eines Tages zweimal: Wenn die Weckzeit wie im Bild gezeigt eingestellt ist, müsste das um 6:30 Uhr und um 18:30 Uhr der Fall sein. Da die hier modellierten Uhrenklassen aber auf einer 24-Stunden-Anzeige (!) basieren, wird in der Simulation auch nur zur exakt eingegebenen Uhrzeit (also einmal innerhalb von 24 Stunden) der Alarm ausgelöst. Für eine realistischere Simulation müsste die Uhrenklasse angepasst werden, z. B. indem man eine Wahlmöglichkeit zwischen 12- und 24-Stunden-Anzeige einbaut. Dies soll hier jedoch nicht mehr thematisiert werden.

3.4 Abschließende Überlegungen zur unterrichtlichen Umsetzung

In den zurückliegenden Kapiteln wird eine Folge von logisch aufeinander aufbauenden Entwicklungsschritten für nach objektorientierten Gesichtspunkten modellierte und programmierte Anwendungen zum Thema Uhr vorgestellt. Dabei erfahren die Schüler, wie ein reales Problem für die Umsetzung in eine technische Anwendung zunächst in ein auf die wesentlichen Momente reduziertes Modell abgebildet wird, sie lernen mit der Modellierungssprache UML eine Möglichkeit kennen, das so gebildete Modell in einer übersichtlichen und standardisierten Weise darzustellen und beschäftigen sich nicht zuletzt mit den grundlegenden Prinzipien der objektorientierten Programmiertechnik.

Zum Abschluss der hier vorliegenden Arbeit sollen in diesem Kapitel zusammenfassend nun noch einige grundsätzliche Überlegungen zur unterrichtlichen Umsetzung folgen.

3.4.1 Ein möglicher Unterrichtsverlauf

Im bisherigen Informatikunterricht haben die Schüler sich bislang vermutlich noch nicht konkret mit der objektorientierten Denkweise auseinandergesetzt. Um eine solche Sichtweise zu Beginn einer Unterrichtseinheit über objektorientierte Modellierung zu motivieren, ist es

bspw. möglich, den Begriff des Objekts und die Möglichkeit, Objekte zu klassifizieren anhand bekannter und alltäglicher Beispiele nachzuvollziehen. Die objektorientierte Sichtweise ist keine „Erfindung“ der Informatik, sie entspricht im Prinzip der natürlichen menschlichen Wahrnehmung der Umwelt. Dies sollte – möglicherweise direkt als Einstieg – den Schülern vor Augen geführt werden.

Als Beispiel könnten hier Fahrzeuge betrachtet werden, die Schüler lernen, zwischen einem ganz konkreten Objekt (der grüne Ford von Frau W.) und der Klasse (Auto), dem Sammelbegriff für gleichartige Objekte unterscheiden. Die Fahrzeugklasse lässt sich in verschiedene Unterkategorien (Unterklassen) wie Autos, Motorräder, Fahrräder gliedern, Autos können wiederum in LKWs und PKWs unterteilt werden und so weiter. Ein Auto besteht aus ganz bestimmten Bauteilen, also wiederum aus Objekten, die zu entsprechenden Klassen, wie bspw. Motor, Reifen, Sitz gehören.

Die Biologie liefert ein schönes Beispiel für die Bildung einer Klassenhierarchie (das die Schüler bereits im Anfangsunterricht Biologie kennen lernen): Sie teilt die Lebewesen in Stamm, Klasse, Ordnung, usw. dadurch entsteht ein weit verzweigter Baum, in dem die Spezialisierung von oben nach unten immer weiter zunimmt.

Im Übrigen sollte man bereits in einer solchen Phase die UML-Notation für Klassen einführen und bspw. die Fahrzeugklasse und ihre Unterklassen mit Hilfe ausgewählter Attribute und Methoden beschreiben und entsprechend in einem UML-Klassendiagramm darstellen.

Was bedeutet es aber, eine objektorientierte Denkweise in die Informatik zu übertragen? Was kennzeichnet ein Programmsystem, das objektorientiert entwickelt wird? Sicherlich wird für die Schüler die Beantwortung dieser Frage erst durch die Beschäftigung mit einem konkreten informatorischen Problem ersichtlich: Nicht das *WIE…* (kann ich dieses und jenes Problem algorithmisch lösen…) soll für die folgenden Problemstellungen im Mittelpunkt stehen, sondern das *WAS…* (soll das gesuchte Modell eigentlich leisten) wird zunächst im Zentrum der Überlegungen stehen. Dieser Auftrag bildet den Ausgangspunkt für die nun folgende Unterrichtseinheit: Die objektorientierte Modellierung (und Programmierung) einer Uhr.

Die Unterrichtseinheit folgt dem in den Kapiteln *3.3.1* bis *3.3.7* beschriebenen Konzept. Die Umsetzung der aufeinander aufbauenden Entwicklungsschritte kann methodisch natürlich unterschiedlich umgesetzt werden:

- Gemeinsame Entwicklung im Unterrichtsgespräch, anschließende Realisierung durch die Schüler.
- Diskussion und Umsetzung (im Unterrichtsgespräch getroffener Vorhaben) in Gruppen, anschließender Austausch der Ergebnisse (z. B. könnten die Schülergruppen im Austausch die entwickelten Fachklassen in einer Anwendung umsetzen).
- Arbeitsblätter mit (mehr oder weniger) detaillierten Vorgaben, die durch die Schüler realisiert werden.

Grundsätzlich denke ich, dass ein gemischtes Vorgehen am besten geeignet ist. Insbesondere bis zur Klasse *TInfoClock* (die sozusagen das Endstadium der Entwicklung einer voll funktionsfähigen „allgemeinen“ Uhr darstellt) halte ich den vorzugsweise im Klassenteam gestalteten Unterricht für am besten geeignet, schließlich werden in diesen Phasen die Begrifflichkeiten der objektorientierten Modellierung und Programmierung erarbeitet. Arbeitsblätter können sinnvoll eingesetzt werden, wenn es um die Erarbeitung spezieller Problemkreise geht, z. B. bei der Beobachtung der Aktivität von Konstruktoren und Destruktoren (S. 16) oder einfach als Informationsblätter, z. B. für die Notation in der UML.

Im Anschluss an die Entwicklung der Uhrenklasse „*TInfoClock*“ folgt im Unterricht die Modellierung spezieller Uhrenklassen. Hier würde ich als Vorgehensweise die Entwicklung in

Gruppenarbeit vorschlagen. Interessant ist dieser Abschnitt insbesondere deswegen, weil hier die Bedeutung von Kapselung und Schnittstellen durch gegenseitigen Austausch der Gruppenergebnisse in einer Anwendung „erprobt“ werden kann: Eine Gruppe erarbeitet bspw. die Fachklasse für einen Wecker, eine andere Gruppe erstellt eine Anwendung, die den Wecker simuliert und muss dabei mit den zur Verfügung gestellten Schnittstellen arbeiten.

Da Uhren und Zeitmessung hier eine zentrale Rolle spielen, bietet es sich an, diese Unterrichtseinheit auch mit Informationen zur Geschichte und Bedeutung der Zeitmessung zu ergänzen, z. B. in Form von Referaten. Als Inhalt für ein Referat eignet sich z. B. die Geschichte der Zeitrechnung, speziell die Geschichte und Technik von Uhren oder das Thema koordinierte Weltzeit (UTC) und Zeitzonen.

3.4.2 Ausblick

In dem hier vorgestellten Konzept beschränkt sich die Modellierung auf sehr wenige Objekte, nämlich im Wesentlichen auf das Objekt Uhr (und die Zählerobjekte, aus denen die Uhr zusammengesetzt ist) sowie die Objekte der Anwendungsoberfläche. Der Vorteil des hier gewählten Beispiels liegt sicherlich darin, dass es sich sehr gut in für die Schüler leicht überschaubare und logisch aufeinander folgende Teilschritte zerlegen lässt, die allmählich (und durch immer wiederkehrende Anwendung vertiefend) in die Konzepte der objektorientierten Programmierung einführen.

Die Effizienz der objektorientierten Denkweise wird jedoch erst an komplexeren Aufgabenstellungen wirklich offensichtlich, d. h. in solchen Anwendungen, deren zu Grunde liegende Problemstellung sich durch ein Modell – mehr oder weniger – zahlreicher (miteinander kommunizierender und interagierender) Objekte repräsentieren lässt.

Weitere Aufgabenstellungen, die von den Schülern im Anschluss an diese Einführung bearbeitet werden, sollten so gewählt werden, dass sie sich durch eine umfangreichere Struktur der beteiligten Objekte auszeichnen. In der didaktischen Literatur werden dazu bspw. Spiele oder Verwaltungssysteme als für den Unterricht geeignete Problemstellungen beschrieben.

4 Literaturverzeichnis

Baumann, Rüdeger (1996), Didaktik der Informatik, Klett: Stuttgart

Damann/Wemßen (2001), Objektorientierte Programmierung mit Delphi – Ein Unterrichtswerk – Band 1, Klett: Stuttgart, Düsseldorf, Leipzig

Damann/Wemßen (2003), Objektorientierte Programmierung mit Delphi – Ein Unterrichtswerk – Band 2, Klett: Stuttgart, Düsseldorf, Leipzig

Erler, Dr. Thomas (2000 – 2002), UML – Das Einsteigerseminar, bhv: Bonn

Hubwieser, Peter (2001), Didaktik der Informatik, Springer: Berlin, Heidelberg, New York

Hessisches Kulturministerium (2003), Lehrplan Informatik, Gymnasialer Bildungsgang, Jahrgangsstufe 11 bis 13

Mittendorfer, Josef (1991), Einführung in objektorientierte Programmierung mit Turbo Pascal, Addison-Wesley

Schubert, Sigrid & Schwill, Andreas (2004), Didaktik der Informatik, Spektrum: Heidelberg, Berlin

Wehrheim, Otto (1994), Objektorientiertes Programmieren im Informatikunterricht, Lehrgangsmanuskript, HILF

Wehrheim, Otto (2003), Objektorientiertes Modellieren mit Delphi, LOG IN 125, S. 30-35

Internetquellen:

http://hsg.region-kaiserslautern.de/faecher/inf/material/se/swep/beispiele/uhren (2003), mk, Museums- und andere Uhren

http://www.oszhdl.be.schule.de/gymnasium/faecher/informatik/index.htm (2004), Siegfried Spolwig, Unterrichtsmaterialien u. a. zu Modellierung, OOP

http://de.wikipedia.org/wiki/Hauptseite (ohne Datum), Informationen u. a. zu Uhren, Zeitzonen, Geschichte der Zeitmessung

http://www.hnbk.de/pdf/umaterial/Vorlesung_Schoppe/IT_AW_HALBJAHR2_Block4_Skript_Version2.pdf (2004), Einführung in die Objektorientierung

5 Anhang

Vollständige Klassenhierarchie der erarbeiteten Uhrenklassen **S. 39**

Arbeitsblatt: Klassendiagramme in UML **S. 40**

Arbeitsblatt: Methodenbindung **S. 41**

Vollständige Auflistung des Programmcodes der Analoguhr **S. 43**

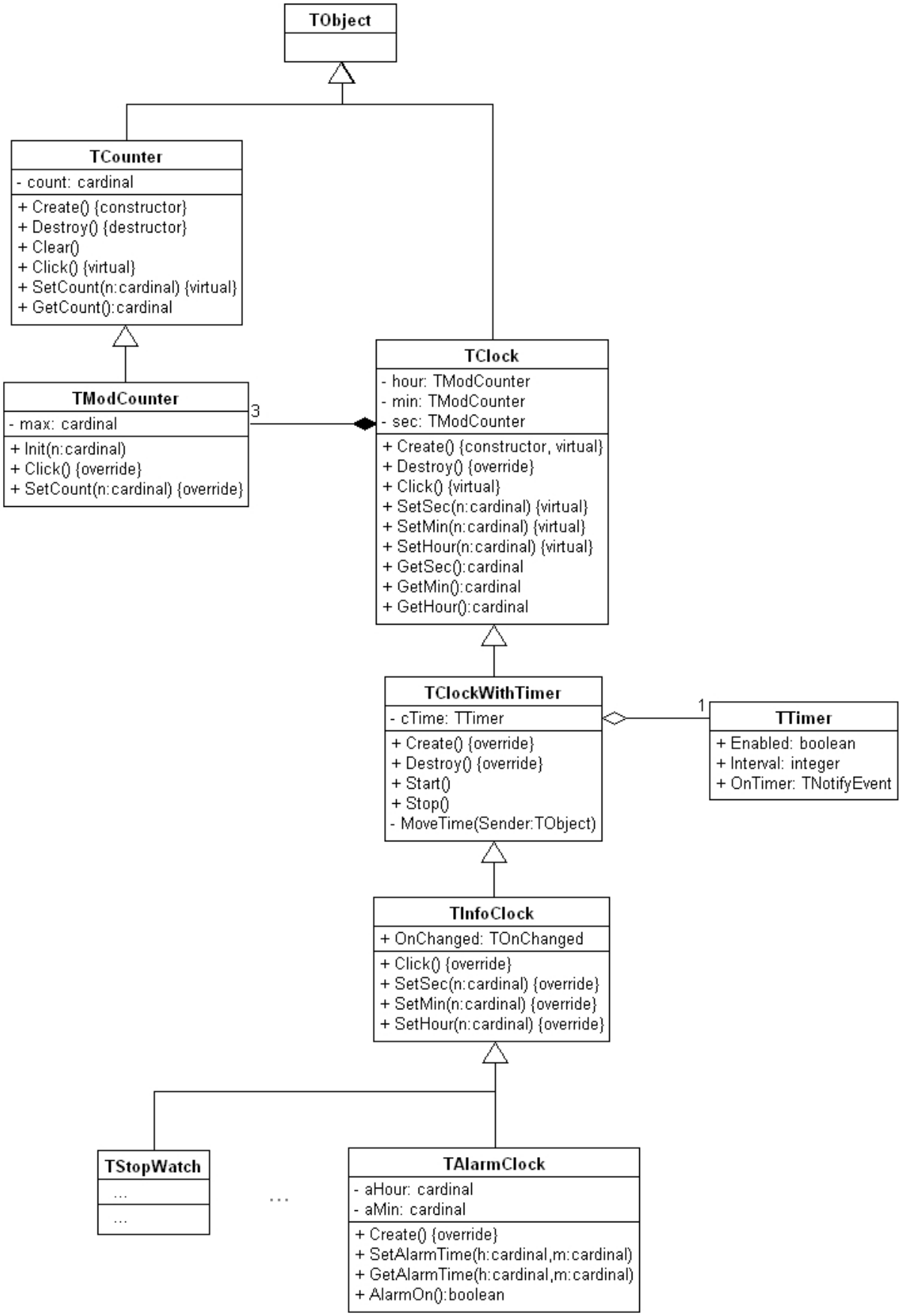
Anwendungsklasse für die Analoguhr S. 43

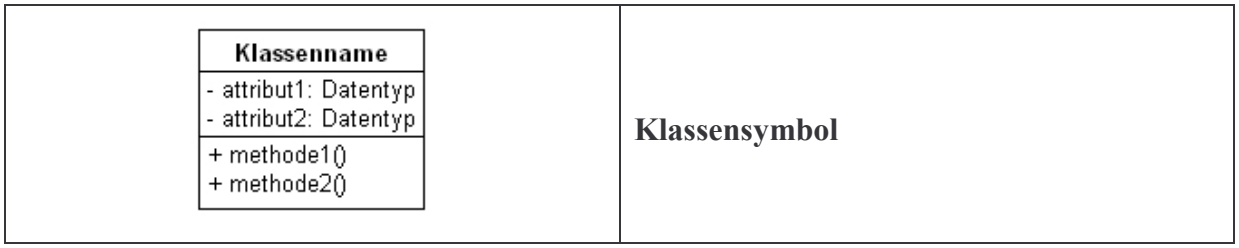
Zähler- und Uhrenklassen S. 44

Analogklasse S. 48

Dialogklasse S. 50

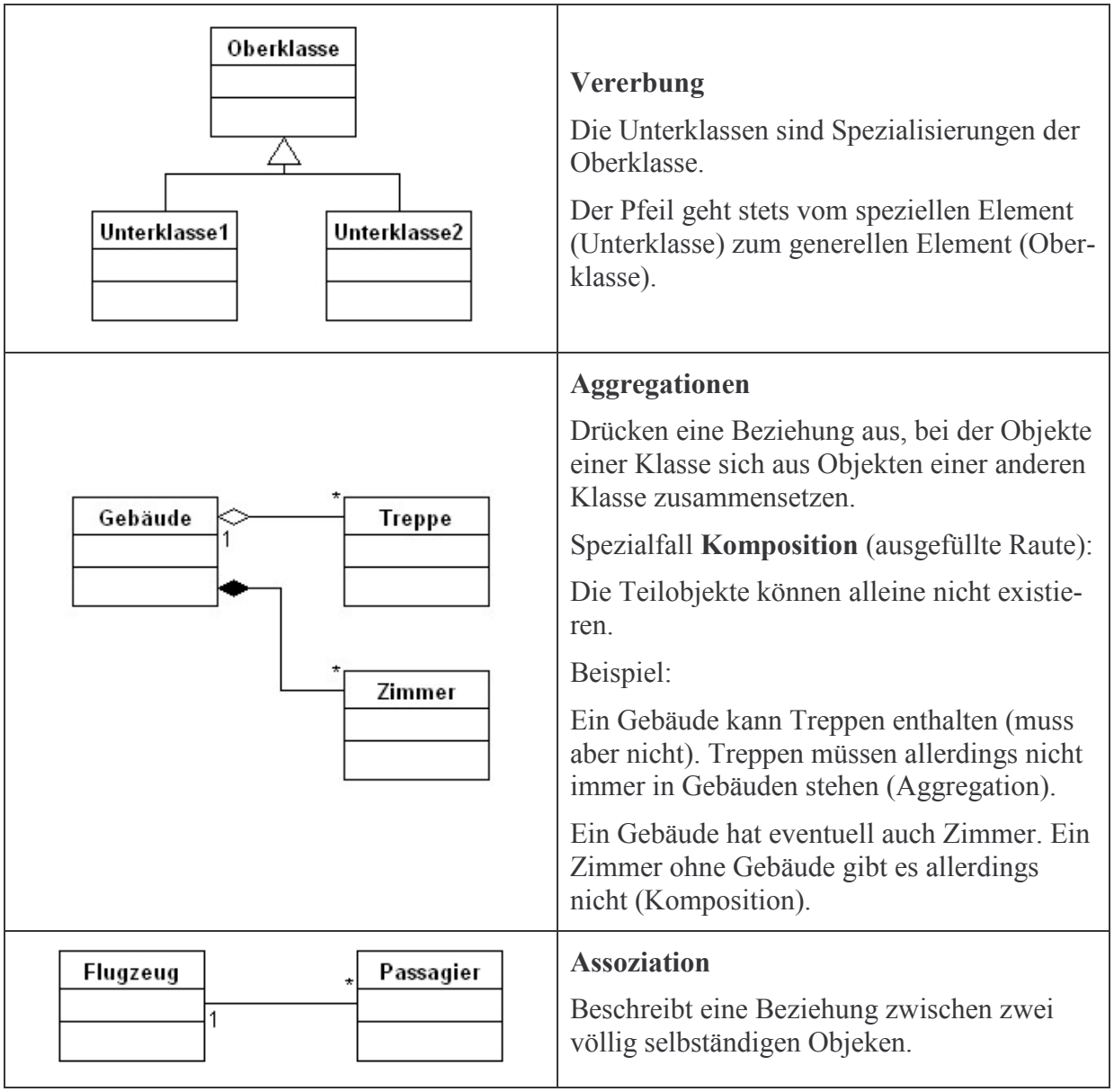
Klassenhierarchie





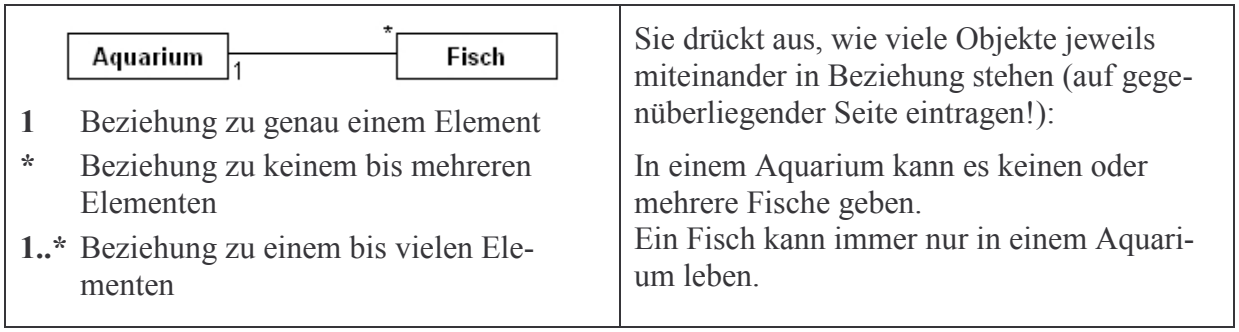
Klassenname - attribut1: Datentyp - attribut2: Datentyp + methode1() + methode2()	**Klassensymbol**

Beziehungen zwischen Klassen



Oberklasse Unterklasse1 Unterklasse2	**Vererbung** Die Unterklassen sind Spezialisierungen der Oberklasse. Der Pfeil geht stets vom speziellen Element (Unterklasse) zum generellen Element (Oberklasse).
Gebäude 1 * Treppe * Zimmer	**Aggregationen** Drücken eine Beziehung aus, bei der Objekte einer Klasse sich aus Objekten einer anderen Klasse zusammensetzen. Spezialfall **Komposition** (ausgefüllte Raute): Die Teilobjekte können alleine nicht existieren. Beispiel: Ein Gebäude kann Treppen enthalten (muss aber nicht). Treppen müssen allerdings nicht immer in Gebäuden stehen (Aggregation). Ein Gebäude hat eventuell auch Zimmer. Ein Zimmer ohne Gebäude gibt es allerdings nicht (Komposition).
Flugzeug 1 * Passagier	**Assoziation** Beschreibt eine Beziehung zwischen zwei völlig selbständigen Objeken.

Kardinalität

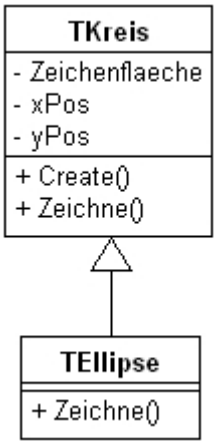


Aquarium 1 * Fisch **1** Beziehung zu genau einem Element ***** Beziehung zu keinem bis mehreren Elementen **1..*** Beziehung zu einem bis vielen Elementen	Sie drückt aus, wie viele Objekte jeweils miteinander in Beziehung stehen (auf gegenüberliegender Seite eintragen!): In einem Aquarium kann es keinen oder mehrere Fische geben. Ein Fisch kann immer nur in einem Aquarium leben.

Statische und dynamische Bindung

1. Statische Bindung

Im folgenden Klassendiagramm wird die Vererbungsbeziehung zwischen zwei Klassen *TKreis* und *TEllipse* beschrieben.
Die Klasse *TEllipse* ist dabei eine Unterklasse der Klasse *TKreis*:




```
type
  TKreis = class
  private
    Zeichenflaeche: TImage;
    xPos, yPos: integer;      //Mittelpunkt
  public
    constructor Create(Sender: TImage);
    procedure Zeichne;
  end;
  TEllipse = class(TKreis)
  public
    procedure Zeichne;
  end;
```

Die Prozedur *Zeichne*() muss für eine Ellipse etwas anders aussehen, als für einen Kreis, daher taucht sie in der Deklaration der Unterklasse *TEllipse* erneut auf: *Zeichne*() wird in der Unterklasse *TEllipse* neu definiert.
Wonach richtet sich die Entscheidung, welche der beiden Methoden *Zeichne*() schließlich in einer Anwendung aufgerufen wird? Das nachfolgende Programm soll dies testen:

Auf einer Zeichenfläche sollen Kreise und Ellipsen dargestellt werden, je nachdem auf welchen Button man drückt. In der Anwendung wird dazu zunächst eine private Variable *Figur* vom Typ *TKreis* vereinbart:

```
type
  TForm1 = class(TForm)
    ...
  private
    Figur: TKreis;
  end;
```

Laut Klassendiagramm ist eine Ellipse einfach nur ein spezieller Kreis, daher sollte es kein Fehler sein, in der Vereinbarung der Objektvariablen zunächst den allgemeineren Datentyp anzugeben, schließlich wird erst während des Programmablaufs entschieden, welche Figur gezeichnet wird.

Die beiden *OnClick*-Prozeduren für die Erzeugung der gewünschten Figur sehen ganz ähnlich aus, im ersten Fall wird jedoch der Konstruktor der Klasse *TKreis* aufgerufen, im zweiten Fall der Konstruktor der Klasse *TEllipse*, anschließend in beiden Fällen die Methode *Zeichne*().

```
procedure TForm1.BKreisClick(Sender: TObject);
begin
  Figur:= TKreis.Create(Image);     //Instanz der Klasse Kreis erzeugen
  Figur.Zeichne;                    //Zeichnen
  Figur.Destroy;
end;
```

```
procedure TForm1.BEllipseClick(Sender: TObject);
begin
  Figur:= TEllipse.Create(Image);   //Instanz der Klasse Ellipse erzeugen
  Figur.Zeichne;                    //Zeichnen
  Figur.Destroy;
end;
```

Aufgabe

- ☞ Testen Sie das Programm (die entsprechenden Units finden Sie im Kursordner). Was stellen Sie fest?
- ☞ Fügen Sie in der Anwendung folgende Änderung ein:
  Deklarieren Sie eine zweite Objektvariable *Figur2* vom Typ *TEllipse*. Ändern Sie die Bezeichner in der Methode *BEllipseClick* entsprechend in *Figur2*. Testen Sie das Programm erneut.

2. Dynamische Bindung

In den Klassendeklarationen von *TKreis* und *TEllipse* wurde die Methode *Zeichne*() **statisch** definiert. Dies bedeutet: Der während der Compilierzeit *deklarierte* Typ bestimmt, welche der *Zeichne*()-Methoden im Programm aufgerufen wird.

In Programmsystemen, in denen während der Entwurfszeit nicht endgültig festgelegt werden kann, von welchem Typ die Objektvariable letztlich sein soll, ist dies nicht sehr vorteilhaft (für jeden möglichen Typ muss eine eigene Objektvariable deklariert werden, das ist ineffektiv!).

Es gibt jedoch auch die Möglichkeit, Methoden **virtuell** zu definieren: In dieser Form definierte Methoden können in abgeleiteten Klassen *überschrieben* werden. Dies hat den Vorteil, dass in einer Anwendung erst während der Laufzeit entschieden wird, welche der vorhandenen (gleichnamigen) Methoden zur Ausführung kommt. Anders gesagt: Die Entscheidung darüber, welche Methode aufgerufen wird, hängt vom *aktuellen* Typ der Objektvariablen ab!

Ändern Sie die Deklaration der Klassenmethoden von *TKreis* und *TEllipse* wie nachfolgend beschrieben ab, um das Verhalten virtueller Methoden zu testen!

Aufgabe

- ☞ Stellen Sie in der Anwendungsklasse zunächst wieder den ursprünglichen Zustand her (löschen Sie also die Objektvariable *Figur2* und ändern Sie die Bezeichner in *BEllipseClick* entsprechend).
- ☞ Ändern Sie nun die Klassendeklaration von *TKreis* und *TEllipse* wie folgt: Fügen Sie der Methode *Zeichne*() in der Deklaration der Klasse *TKreis* die Direktive **virtual** hinzu:

  ```
  TKreis = class
    ...
    public
      procedure Zeichne; virtual;
  ```

  Fügen Sie der Methode *Zeichne*() in der Klasse *TEllipse* die Direktive **override** hinzu:

  ```
  TEllipse = class(TKreis)
  public
    procedure Zeichne; override;
  ```

  Führen Sie erneut das Anwendungsprogramm aus!

Im folgenden findet sich die komplette Auflistung des Quellcodes für die Realisierung der Analoguhr in Kapitel 3.3.9. Der Quellcode ist auf vier Units verteilt: *UAnaloguhr* ist die Quelldatei der eigentlichen Anwendung, *UClock* enthält die Realisierung sämtlicher Zähler- und Uhrenklassen (aus Gründen der Vereinfachung hier in einer Unit zusammengefasst), *UAnalog* enthält den Quelltext für die Realisierung der Analogdarstellung und *UDialog* den entsprechenden Quelltext für die Dialogfunktion.

Anwendungsklasse für die Analoguhr

```
unit UAnalogUhr;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, UClock, UAnalog, UDialog;

type
  TFAnalogUhr = class(TForm)
    Panel1: TPanel;
    BZeit: TButton;
    procedure FormCreate(Sender: TObject);
    procedure BZeitClick(Sender: TObject);
  private
    { Private-Deklarationen }
    Uhr: TInfoClock;
    Anzeige: TAnalog;
    Dialog: TDialog;
    SecZeiger, MinZeiger, StdZeiger: real;
    procedure Loeschen;
    procedure NeuZeichnen;
  public
    { Public-Deklarationen }
  end;

var
  FAnalogUhr: TFAnalogUhr;

implementation

{$R *.dfm}

procedure TFAnalogUhr.FormCreate(Sender: TObject);
begin
  Uhr:= TInfoClock.Create;
  Anzeige:= TAnalog.Create(FAnalogUhr, 5, 65, 260);
  Anzeige.DrawDial(clSilver);
  NeuZeichnen;
  Uhr.OnChanged:= Loeschen;
  Uhr.Start;
end;

procedure TFAnalogUhr.Loeschen;
begin
  with Anzeige do begin
    Canvas.Pen.Color:= clSilver;
    Canvas.Pen.Width:= 2;
    DrawHand(SecZeiger, 9);
    Canvas.Pen.Width:= 4;
    DrawHand(MinZeiger, 9);
    DrawHand(StdZeiger, 7);
  end;
  NeuZeichnen;
end;
```

```
procedure TFAnalogUhr.NeuZeichnen;
begin
  with Anzeige do begin
    Canvas.Pen.Color:= clRed;
    Canvas.Pen.Width:= 2;
    SecZeiger:= GetSecHand(Uhr);
    DrawHand(SecZeiger, 9);
    Canvas.Pen.Color:= clBlack;
    Canvas.Pen.Width:= 4;
    MinZeiger:= GetMinHand(Uhr);
    DrawHand(MinZeiger, 9);
    StdZeiger:= GetHourHand(Uhr);
    DrawHand(StdZeiger, 7);
  end;
end;

procedure TFAnalogUhr.BZeitClick(Sender: TObject);
var hour, min: cardinal;
begin
  Dialog:= TDialog.Create(FAnalogUhr);
  Dialog.Caption:= 'Uhrzeit stellen';
  Dialog.ShowModal;
  if not Dialog.BAbbruch.Cancel then
  begin
    hour:= StrToInt(Dialog.EHour.Text);
    min:= StrToInt(Dialog.EMin.Text);
    with Uhr do begin
      SetHour(hour); SetMin(min)
    end;
  end;
  Dialog.Free;
end;

end.
```

Zähler- und Uhrenklassen

```
unit UClock;

interface

uses
  ExtCtrls;

type
  TCounter = class        //von TObject abgeleitet
  private
    count: cardinal;
  public
    constructor Create;
    destructor Destroy; override;
    procedure Clear;
    procedure Click; virtual;
    procedure SetCount(n: cardinal); virtual;
    function GetCount: cardinal;
  end;
  TModCounter = class(TCounter)
  private
    max: cardinal;
  public
    procedure Init(n: cardinal);
    procedure Click; override;
    procedure SetCount(n: cardinal); override;
  end;
```

```
  TClock = class
  private
    sec: TModCounter;
    min: TModCounter;
    hour: TModCounter;
  public
    constructor Create; virtual;
    destructor Destroy; override;
    procedure Click; virtual;
    procedure SetSec(n: cardinal); virtual;
    procedure SetMin(n: cardinal); virtual;
    procedure SetHour(n: cardinal); virtual;
    function GetSec: cardinal;
    function GetMin: cardinal;
    function GetHour: cardinal;
  end;
  TClockWithTimer = class(TClock)
  private
    cTime: TTimer;
    procedure MoveTime(Sender: TObject);
  public
    constructor Create; override;
    destructor Destroy; override;
    procedure Start;
    procedure Stop;
  end;
  TOnChanged = procedure of object;
  TInfoClock = class(TClockWithTimer)
  public
    OnChanged: TOnChanged;
    procedure Click; override;
    procedure SetSec(n: cardinal); override;
    procedure SetMin(n: cardinal); override;
    procedure SetHour(n: cardinal); override;
  end;
  TAlarmClock = class(TInfoClock)
  private
    aHour, aMin: cardinal;
  public
    constructor Create; override;
    procedure SetAlarmTime(h, m: cardinal);
    procedure GetAlarmTime(var h, m: cardinal);
    function AlarmOn: boolean;
  end;

implementation


{-------------------- TCounter ----------------------------}

constructor TCounter.Create;
begin
  count:= 0;
end;

destructor TCounter.Destroy;
begin
  inherited Destroy;
end;

procedure TCounter.Clear;
begin
  count:= 0;
end;

procedure TCounter.Click;
begin
  Inc(count);
end;
```

```
procedure TCounter.SetCount(n: cardinal);
begin
  count:= n;
end;

function TCounter.GetCount: cardinal;
begin
  GetCount:= count;
end;


{-------------------- TModCounter -------------------------}

procedure TModCounter.Init(n: cardinal);
begin
  max:= n;
end;

procedure TModCounter.Click;
begin
  inherited Click;
  if GetCount >= max then Clear;
end;

procedure TModCounter.SetCount(n: cardinal);
begin
  inherited SetCount(n);
  if GetCount >= max then Clear;
end;


{-------------------- TClock ------------------------------}

constructor TClock.Create;
begin
  hour:= TModCounter.Create;
  hour.Init(24);
  min:= TModCounter.Create;
  min.Init(60);
  sec:= TModCounter.Create;
  sec.Init(60);
end;

destructor TClock.Destroy;
begin
  hour.Free; min.Free; sec.Free;
  inherited Destroy;
end;

procedure TClock.Click;
begin
  sec.Click;
  if sec.GetCount = 0 then
  begin
    min.Click;
    if min.GetCount = 0 then
      hour.Click;
  end;
end;

procedure TClock.SetSec(n: cardinal);
begin
  sec.SetCount(n);
end;

procedure TClock.SetMin(n: cardinal);
begin
  min.SetCount(n);
end;
```

```
procedure TClock.SetHour(n: cardinal);
begin
  hour.SetCount(n);
end;

function TClock.GetSec: cardinal;
begin
  GetSec:= sec.GetCount;
end;

function TClock.GetMin: cardinal;
begin
  GetMin:= min.GetCount;
end;

function TClock.GetHour: cardinal;
begin
  GetHour:= hour.GetCount;
end;


{-------------------- TClockWithTimer ----------------------}

constructor TClockWithTimer.Create;
begin
  inherited Create;
  cTime:= TTimer.Create(nil);
  cTime.Interval:= 1000;
  cTime.Enabled:= false;
  cTime.OnTimer:= MoveTime;
end;

destructor TClockWithTimer.Destroy;
begin
  cTime.Free;
  inherited Destroy;
end;

procedure TClockWithTimer.Start;
begin
  cTime.Enabled:= true;
end;

procedure TClockWithTimer.Stop;
begin
  cTime.Enabled:= false;
end;

procedure TClockWithTimer.MoveTime(Sender: TObject);
begin
  Click;
end;


{-------------------- TInfoClock ---------------------------}

procedure TInfoClock.Click;
begin
  inherited Click;
  if assigned(OnChanged) then OnChanged;
end;

procedure TInfoClock.SetSec(n: cardinal);
begin
  inherited SetSec(n);
  if assigned(OnChanged) then OnChanged;
end;
```

```
procedure TInfoClock.SetMin(n: cardinal);
begin
  inherited SetMin(n);
  if assigned(OnChanged) then OnChanged;
end;

procedure TInfoClock.SetHour(n: cardinal);
begin
  inherited SetHour(n);
  if assigned(OnChanged) then OnChanged;
end;


{-------------------- TAlarmClock -------------------------}

constructor TAlarmClock.Create;
begin
  inherited Create;
  aHour:= 0; aMin:= 0;
end;

procedure TAlarmClock.SetAlarmTime(h, m: cardinal);
begin
  aHour:= h; aMin:= m;
end;

procedure TAlarmClock.GetAlarmTime(var h, m: cardinal);
begin
  h:= aHour; m:= aMin;
end;

function TAlarmClock.AlarmOn: boolean;
begin
  if (aHour = GetHour) and (aMin = GetMin) then
    AlarmOn:= true
  else
    AlarmOn:= false;
end;

end.
```

Analogklasse

```
unit UAnalog;

interface

uses
  Forms, Graphics, Extctrls, UClock;

type
  TExtent = 1..10;
  TAnalog = class(TImage)
  private
    r, xPos, yPos: integer;
  public
    constructor Create(Owner: TForm; x, y, imWidth: integer);
    procedure DrawDial(bgColor: TColor = clWhite);
    procedure DrawHand(w: real; ext: TExtent = 10);
    function GetSecHand(Sender: TClock): real;
    function GetMinHand(Sender: TClock): real;
    function GetHourHand(Sender: TClock): real;
    function GetAlarmHand(Sender: TAlarmClock): real;
  end;
```

```
implementation


{-------------------- TAnalog ------------------------------}

constructor TAnalog.Create(Owner: TForm; x, y, imWidth: integer);
begin
  inherited Create(Owner);
  Parent:= Owner;
  Left:= x; Top:= y; Width:= imWidth; Height:= imWidth;
  r:= Width div 2;
  xPos:= r;
  yPos:= r;
end;

procedure TAnalog.DrawDial(bgColor: TColor = clWhite);
var i: integer;
    mult: real;
begin
  with Canvas do begin
    Brush.Color:= bgColor;
    Pen.Width:= 3;
    Ellipse(xPos-r, yPos-r, xPos+r, yPos+r);
    for i:=1 to 60 do              //Einteilung des Ziffernblattes
    begin
      if (i mod 5)=0 then begin
        Pen.Width:=2; mult:=0.9; end
      else begin
        Pen.Width:=1; mult:=0.93; end;
      MoveTo(round(xPos+cos(i*Pi/30)*mult*r), round(yPos+sin(i*Pi/30)*mult*r));
      LineTo(round(xPos+cos(i*Pi/30)*r), round(yPos+sin(i*Pi/30)*r));
    end;
  end;
end;

procedure TAnalog.DrawHand(w: real; ext: TExtent = 10);
var xEPos, yEPos: integer;
begin
  xEPos:= xPos + round(cos(w)*r*ext/10*0.9);
  yEPos:= yPos + round(sin(w)*r*ext/10*0.9);
  Canvas.MoveTo(xPos, yPos);
  Canvas.LineTo(xEPos, yEPos);
end;

function TAnalog.GetSecHand(Sender: TClock): real;
var wSec: real;
begin
  wSec:= 3/2*Pi + Sender.GetSec*(Pi/30); //Winkelberechnung sekundengenau
  if wSec >= (2*Pi) then
    wSec:= wSec - 2*Pi;
  GetSecHand:= wSec;
end;

function TAnalog.GetMinHand(Sender: TClock): real;
var wMin: real;
begin
  wMin:= 3/2*Pi + Sender.GetMin*(Pi/30); //Winkelberechnung minutengenau
  if wMin >= (2*Pi) then
    wMin:= wMin - 2*Pi;
  GetMinHand:= wMin;
end;

function TAnalog.GetHourHand(Sender: TClock): real;
var wHour: real;
begin
  wHour:=3/2*Pi+Sender.GetHour*(Pi/6)+Sender.GetMin*(Pi/360); //Winkelberechnung
  if wHour >= (2*Pi) then                                     //minutengenau
    wHour:= wHour - 2*Pi;
  GetHourHand:= wHour;
end;
```

```
function TAnalog.GetAlarmHand(Sender: TAlarmClock): real;
var wAlarm: real;
    h, m: cardinal;
begin
  Sender.GetAlarmTime(h, m);
  wAlarm:= 3/2*Pi + h*(Pi/6) + m*(Pi/360);
  GetAlarmHand:= wAlarm;
end;

end.
```

Dialogklasse

```
unit UDialog;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TDialog = class(TForm)
    BOk: TButton;
    EHour: TEdit;
    EMin: TEdit;
    Label2: TLabel;
    BAbbruch: TButton;
    procedure BOkClick(Sender: TObject);
    procedure BAbbruchClick(Sender: TObject);
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

implementation

{$R *.dfm}

procedure TDialog.BOkClick(Sender: TObject);
begin
  Close;
end;

procedure TDialog.BAbbruchClick(Sender: TObject);
begin
  BAbbruch.Cancel:= true;
  Close;
end;

end.
```

6 Inhaltsverzeichnis der CD

Ordner **Programme**: Fachklassen und Anwendungsprogramme (nach Kapiteln geordnet).

Enthält die folgenden Unterordner:

- Ordner **Gemeinsam genutzte Klassen**: Zähler- und Uhrenklassen (UClock.pas), Analogklasse (UAnalog.pas), Dialogklasse (UDialog.pas).
- Ordner **Kap. 3.3.2 Einfacher Zähler**: Anwendungsbeispiel für einen einfachen Zähler.
- Ordner **Kap. 3.3.2 Einfacher Zähler V2**: Alternative Version, zeigt die Änderung der Speicherbelegung bei Aufruf des Konstruktors bzw. Destruktors.
- Ordner **Kap. 3.3.3 Methodenbindung**: Beispielprogramm zur Demonstration des Unterschieds zwischen statischer und dynamischer Bindung.
- Ordner **Kap. 3.3.3 Modulo-Zähler**: Anwendungsbeispiel für einen Zähler mit festgelegter Zählergrenze (hier: 24).
- Ordner **Kap. 3.3.4 Einfache Uhr**: Anwendungsbeispiel für eine einfache (nicht selbständig lauffähige) Uhr.
- Ordner **Kap. 3.3.5 Uhr mit Timer**: Anwendungsbeispiel für die Uhr mit Timer.
- Ordner **Kap. 3.3.6 Uhr Endversion**: Anwendungsbeispiel für die endgültige Uhrenversion.
- Ordner **Kap. 3.3.6 Uhr V2 mit 7Segment**: Alternative Anwendung für die Uhr, hier mit 7-Segment-Anzeige und Dialogfunktion.
- Ordner **Kap. 3.3.8 Wecker mit 7Segment**: Anwendungsbeispiel für einen Wecker mit 7-Segment-Anzeige.
- Ordner **Kap. 3.3.9 Analoguhr**: Anwendungsbeispiel für eine Uhr mit Analoganzeige.
- Ordner **Kap. 3.3.9 Analogwecker**: Anwendungsbeispiel für einen Wecker mit Analoganzeige.

Ordner **Prüfungsarbeit**: Prüfungsarbeit als PDF-Dokument.

7 Erklärung

Ich versichere, diese Arbeit selbständig verfasst, keine anderen als die angegebenen Hilfsmittel verwendet und die Stellen, die in anderen Werken im Wortlaut, als Graphik oder dem Sinne nach entnommen sind, mit Quellenangaben kenntlich gemacht zu haben.

Darmstadt, den 13.09.04

Katja Weishaupt