AWT,
das "Abstract Window Toolkit"

Kapitel 9, aus
"Einführung in die Informatik"
Küchlin , Weber

Java AWT

Das abstract window toolkit ist eine Klassenbibliothek, zur Programmierung graphischer Benutzeroberflächen (graphical user interfaces -- GUIs).

- Wichtigste Bestandteile des AWT sind
  - Klassen zur Darstellung graphischer Komponenten in einem Fenstersystem und
  - Mechanismen, die eine Interaktion mit dem Benutzer ermöglichen, indem sie es erlauben, auf Ereignisse (events) wie z. B. Mausklicks zu reagieren.

Java AWT

Die Klassenbibliothek AWT ist ein objektorientiertes Rahmenwerk (object-oriented framework).

Wir können relativ leicht eine graphische Benutzeroberfläche programmieren,

- indem man eigene Klassen von AWT-Klassen ableitet
- und wenige Methoden, die in AWT definiert sind, durch eigenen Code überschreibt.
  - Solche Methoden sind **Haken** (hooks) im Rahmenwerk, an denen benutzerspezifischer Code „eingehängt" wird.

Über Vererbung und dynamisches Binden wird der eigene Code nahtlos in den durch AWT gegebenen Rahmen integriert.

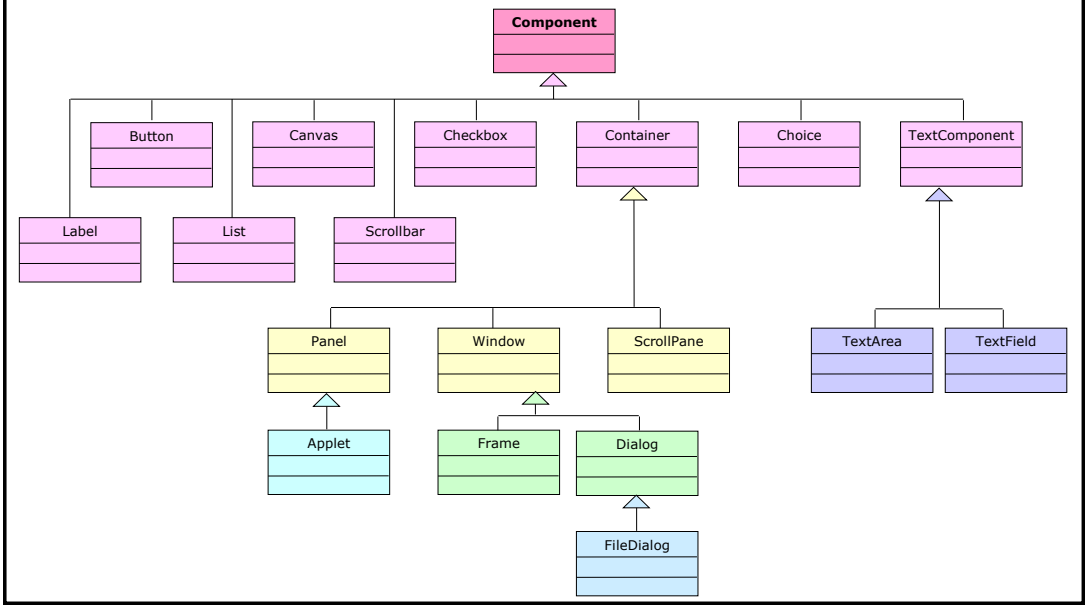
- Man kann somit den Code wieder verwenden, den das Rahmenwerk des AWT liefert.
- Die Kontrolle liegt ganz beim Framework, das den speziellen Darstellungscode des Benutzers aufruft.

Java AWT

Bemerkung:

- Die nachfolgenenden Beispiele verwenden AWT, obwohl es inzwischen neuere Klassenbibliotheken für die GUI-Programmierung in Java gibt.
  - Insbesondere die mit **Swing** bezeichnete Bibliothek.
  - Diese ist eine Erweiterung des AWT und bietet gewisse technische Vorteile, wie etwa ein systemunabhängiges aber benutzerdefinierbares „look and feel" von graphischen Elementen, auf die an dieser Stelle nicht näher eingegangen werden soll.
  - Aufgrund seiner etwas einfacheren Struktur ist das AWT aber für unsere Zwecke geeigneter als Swing.
- Die Prinzipien des Entwurfs graphischer Komponenten und der Ereignisbehandlung sind in beiden Bibliotheken die gleichen.

Vererbungshierarchie für Component in awt



Funktionalität von Component

Die von der Klasse Component zur Verfügung gestellte Funktionalität, d. h. sie besitzt folgende (abstrakte) Methoden; in einer vom Benutzer definierten Ableitung können diese entsprechend überschrieben werden.

Grundlegende Zeichenfunktionen:

- void paint(Graphics g) ist die Hauptschnittstelle („Haken") zum Anzeigen eines Objekts.
  - Diese Methode wird in einer abgeleiteten Klasse überschrieben, wobei die Methoden, die das Graphics-Objekt g zur Verfügung stellt, benutzt werden.
- void update()
- void repaint()

Funktionen zur Darstellung:

- void setFont(Font f) setzt die Schriftart (font) innerhalb der Graphik.
- void setForeground(Color c)
- void setBackground(Color c)

Funktionalität von Component

Funktionen zur Größen- und Positionskontrolle

- Dimension getMinimumSize()
- Dimension preferredSize()
- void setSize(int width, int height) setzt die Größe der Komponente (in Pixeleinheiten)
- void setSize(Dimension d)
- Dimension getSize()

Die Klasse Graphics

Die paint-Methode von AWT-Komponenten besitzt einen Parameter vom Typ Graphics.

Die Klasse Graphics ist die abstrakte Basisklasse für alle Klassen, die graphische Ausgabeobjekte realisieren.

- Wie z. B. Treiberklassen für verschiedene Bildschirme, Drucker, . . .
- Sie stellt einen sogenannten Graphik-Kontext (graphics context) zur Verfügung.

Die Klasse Graphics

Ein Graphik-Kontext wird vom Rahmenwerk des AWT erzeugt, nicht unmittelbar im Anwendungsprogramm.

- Über die „Haken", die ein Graphics-Objekt als Parameter besitzen, kann im Anwendungsprogramm aber auf den vom AWT erzeugten Graphik-Kontext zugegriffen werden.
  - Ein wichtiger Haken ist etwa die Methode paint(Graphics g) der Basisklasse Component.

Die Klasse Graphics spezifiziert eine Vielzahl von Methoden, mit denen in dem Graphik-Kontext „gezeichnet" werden kann.

- Viele der Argumente, die vom Typ *int* sind, bezeichnen Koordinaten, die in Einheiten von Bildpunkten (picture element, pixel) gegeben sind, an denen ein spezielles Objekt gezeichnet werden soll
  - Dabei hat die *linke obere* Ecke die Koordinate (0,0).
  - Die x-Koordinate wächst nach *rechts*, die y-Koordinate nach *unten*.

Die Klasse Graphics

Einige der wichtigsten Methoden sind die Folgenden:

- void setColor(Color c) legt die in den folgenden Operationen verwendete Farbe fest.
- void setFont(Font f) legt die in den folgenden Text-Operationen verwendete Schriftart (font) fest.
- void drawString(String str, int x, int y) schreibt den String *str* in dem gerade gültigen Font (und in der gerade gültigen Farbe) an den Punkt (x, y).
- void drawLine(int x1, int y1, int x2, int y2) zeichnet eine Strecke (in der gerade gültigen Farbe) vom Punkt (x1, y1) zum Punkt (x2, y2) im Koordinatensystem des Graphik-Kontexts.

Die Klasse Graphics

Einige der wichtigsten Methoden (Forts.)

- void drawRect(int x, int y, int width, int height) zeichnet den Umriss eines Rechtecks (in der gerade gültigen Farbe), das durch die Parameter spezifiziert ist.
  - Der linke und rechte Rand des Rechtecks sind bei x und x+width .
  - Der obere und untere Rand sind bei y und y+height .
- void fillRect(int x, int y, int width, int height) zeichnet ein Rechteck, das mit der gerade gültigen Farbe gefüllt ist.
  - Der linke und rechte Rand des Rechtecks sind bei x und x+width-1 .
  - Der obere und untere Rand sind bei y und y+height-1 .
- void drawOval(int x, int y, int width, int height) zeichnet den Umriss einer Ellipse (in der gerade gültigen Farbe), die in das Rechteck eingepasst ist, das durch die Parameter gegeben ist.
- void fillOval(int x, int y, int width, int height) zeichnet eine Ellipse, die mit der gerade gültigen Farbe ausgefüllt ist.
  - Die Ellipse ist in das Rechteck eingepasst, das durch die Parameter gegeben ist.

Frames

Die von Component abgeleitete Klasse Frame aus java.awt dient zum Zeichnen von Fenstern mit Rahmen.

- In einem Frame-Objekt kann ein Menü-Balken (menu bar) verankert sein, über den Dialog-Menüs verankert sein können.

AWT-Frames haben folgende wichtige spezifische Funktionalität:

- void setTitle(String title) schreibt einen Titel in die obere Leiste des Fensters,
- void setMenuBar(MenuBar mb),
- MenuBar getMenuBar().

Um im Fenster, das durch ein Frame-Objekt erzeugt wird, etwas anzeigen zu können, muss - wie bei allen AWT-Komponenten - die paint-Methode überschrieben werden.

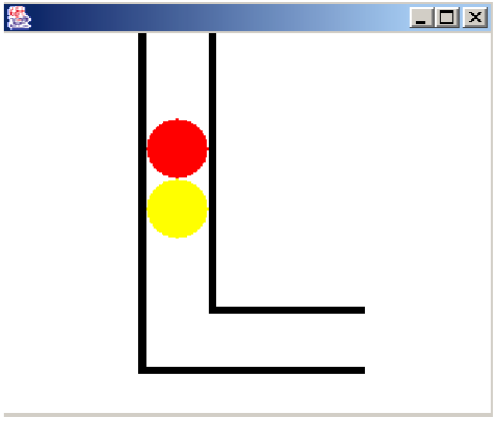
Frames: Beispiel

Das folgende Programm zeichnet in einem Fenster Kreise und Linien

```
import java.awt.*;

public class FrameBeispiel extends Frame {
 /**
 *  Erzeugt ein Fenster der Groesse 320 x 280
 */
 FrameBeispiel() {
            setSize(320,280);
            setVisible (true);
 }

  .....

 public static void main (String args[]) {

            new FrameBeispiel();
 }
}
```

```
/**
 * Zeichnet einige einfache graphische Objekte
*/
public  void paint (Graphics g) {
            // Röhre
            g.setColor( Color.black);
            g.fillRect(90,5,5,240);
            g.fillRect(135,5,5,200);
            g.fillRect(90,245,145,5);
            g.fillRect(135,205,100,5);
            // Bälle
            g.setColor(Color.red);
            g.fillOval(95,80,40,40);
            g.setColor(Color.yellow);
            g.fillOval(95,120,40,40);
}
```

Ausgabe des Hauptprogramms von FrameBeispiel



Container

Container dienen zur Gruppierung von AWT-Komponenten.

- Sind selber eine AWT-Komponente.
  - „Rekursion" daher möglich: Gruppierungen von gruppierten Komponenten.

Jeder Container besitzt einen LayoutManager, der für die Anordnung der AWT-Komponenten verantwortlich ist.

- Der Programmierer spezifiziert die relativen Positionen der Komponenten.
- Ihre absolute Positionierung und Dimensionierung bleibt dem Layout-Manager überlassen.
  - Dieser versucht, eine „günstige" Lösung in Abhängigkeit von der Fenstergröße und den abstrakten Vorgaben des Programmierers zu finden.

Container

Folgende grundlegende Methoden werden zur Verfügung gestellt:

- void setLayout(LayoutManager m) setzt den für den Container verantwortlichen LayoutManager.
- void add(Component c, . . .) fügt in Abhängigkeit vom LayoutManager die Komponente in den Container ein.
- void remove(Component) entfernt die Komponente aus dem Container.

Container

Unterstützt werden folgende Layout-Typen:

- BorderLayout kann zur Gruppierung von Komponenten an den Rändern des Containers benutzt werden.
  - Die vier Ränder werden mit NORTH, EAST, SOUTH bzw. WEST bezeichnet, wobei diese Himmelsrichtungen der Anordnung auf Landkarten entsprechen.
  - Das Innere eines solchen Containers wird mit CENTER bezeichnet.
- CardLayout ergibt eine spielkartenförmige Anordnung der Komponenten.
- FlowLayout ergibt eine „fließende" Anordnung, die linksbündig (FlowLayout.LEFT), zentriert (FlowLayout.CENTER) oder rechtsbündig (FlowLayout.RIGHT) sein kann.
- GridLayout richtet die Komponenten an einem Gitter aus.
- GridBagLayout dient zur einer flexiblen horizontalen und vertikalen Anordnung von Komponenten, die nicht alle von der gleichen Größe zu sein brauchen.

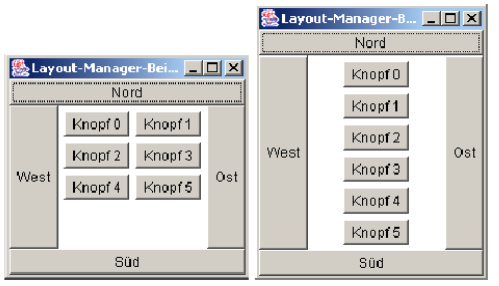
Container: Beispiel

```
/**
 * Beispielprogramm zum Layoutmanger
 */
import java.awt.*;

public class LayoutBeispiel extends Frame {

    public static void main (String args[]) {

            Frame f = new Frame("Layout-Manager-Beispiel");
            f.setSize(250, 200);
            // Fenstergrösse für das 2. Beispiel
            // f.setSize(200,245)
            // Vier Buttons
            // f.setLayout (new BorderLayout)
            // überflüssig da default
            f.add (new Button ("Nord"), BorderLayout.NORTH);
            f.add (new Button ("Ost"),  BorderLayout.EAST);
            f.add (new Button ("South"),BorderLayout.SOUTH);
            f.add (new Button ("West"), BorderLayout.WEST);


            ......................

            f.setVisible(true);
    }
}
```

```
// Im Zentrum ein Container im FlowLayout
Container c = new Container();
c.setLayout (new FlowLayout() );

for (int i = 0; i < 6; i++) {
            c.add(new Button("Knopf " + i ));
}

c.setVisible(true);
f.add(c, BorderLayout.CENTER);
```

Beispiel zu Layout-Managern

Ausgaben von LayoutBeispiel



`f.setSize(250,200);` `f.setSize(200,245);`

Ereignisse (events)

Für die Interaktion mit einem Benutzer ist die Behandlung äußerer Ereignisse (events) wesentlich.

- Dies kann z. B. eine Mausbewegung, ein Mausklick oder das Drücken einer Taste sein usw.

AWT stellt einen entsprechenden Rahmen zur Verfügung, den wir im folgenden skizzieren wollen.

Verschiedene Klassen von Ereignissen sind in Java durch verschiedene Java-Klassen repräsentiert.

- Jede Ereignisklasse ist eine Unterklasse von java.util.EventObject.
  - Ereignisobjekte tragen gegebenenfalls weitere nützliche Informationen in sich, die das Ereignis weiter charakterisiert

    z. B. die X- und Y-Koordinate bei MouseEvent, das zu einem Mausklick gehört.

Ereignisse (events)

AWT-Events

- AWT-Ereignisse sind Unterklassen von java.awt.AWTEvent
  - Sie sind im Paket java.awt.event zusammengefasst.
- AWT-Komponenten können folgende Ereignisse erzeugen:

```
ActionEvent       AdjustmentEvent
ComponentEvent    ContainerEvent
FocusEvent        ItemEvent
KeyEvent          MouseEvent
TextEvent         WindowEvent
```

Ereignisquellen und Ereignisempfänger

Jedes Ereignis wird von einer Ereignisquelle (event source) generiert

- dies ist ein anderes Objekt, das man mit getSource() erhält.

Die Ereignisquelle liefert ihre Ereignisse an interessierte Parteien, die Ereignisempfänger (event listener) aus, die dann selbst eine geeignete Behandlung vornehmen.

Die Zuordnung zwischen Quelle und Empfängern darf nicht im Programmcode statisch fixiert werden, sondern sie muss sich zur Laufzeit dynamisch ändern können.

- Dazu verwendet man ein elegantes Prinzip, das wir (in ähnlicher Form) beim generischen Programmieren kennen lernen.

Ereignisquellen und Ereignisempfänger

Die Ereignisempfänger müssen sich bei der Quelle an- und abmelden.

- Ereignisquelle implementiert hierzu eine geeignete Schnittstelle.
- Die Ereignisquelle unterhält eine Liste von angemeldeten Ereignisempfängern.
- Hat die Ereignisquelle ein Ereignis generiert, dann sucht sie die Liste der Empfänger ab und ruft auf jedem Empfänger eine Methode auf, der sie das Ereignisobjekt (per Referenz) übergibt.

  Hierzu ist für jede Ereignisklasse eine entsprechende Schnittstelle für Empfänger (event listener interface) spezifiziert.

Ereignisquellen und Ereignisempfänger

Beispiel: Bei einer AWT-Komponente können ein oder mehrere event listener mittels einer Methode add*Type*Listener() registriert werden.

- Im folgenden Programmfragment wird ein ActionListener bei einer Button-Komponente registriert.

```
import java.awt.*;
import java.awt.event.*;
// ...
Button button;
// ...
button.addActionListener(this);
```

Ein event listener interface definiert Methoden, die beim Auftreten eines Ereignisses automatisch aufgerufen werden und die man so implementieren kann, dass sie das Ereignis behandeln.

Jeder Empfänger muss das zum Ereignis gehörige Interface implementieren, damit die Quelle ihn aufrufen kann.

Jeder Empfänger implementiert dies individuell so, dass die für ihn typische Bearbeitung des übergebenen Ereignisses stattfindet.

Events : WindowListener

Als Beispiel wollen wir das zum Paket **java.awt.event** gehörige Interface WindowListener beschreiben.

In der Schnittstelle sind die folgenden Methoden spezifiziert:

- void windowOpened(WindowEvent e),
- void windowClosing(WindowEvent e) Diese Methode wird aufgerufen, wenn vom Benutzer ein sogenannter Close-Request abgesetzt wurde,
- void windowClosed(WindowEvent e),
- void windowIconified(WindowEvent e),
- void windowDeiconified(WindowEvent e),
- void windowActivated(WindowEvent e),
- void windowDeactivated(WindowEvent e).

WindowListener-Interface werden meist von einer Erweiterung der Frame-Klasse implementiert, um das Fenster, in dem gezeichnet wird, wieder schließen zu können.

Adapter-Klassen

Bei der Verwendung eines Listener-Interfaces müssen wie bei allen Interfaces immer alle Methoden implementiert werden.

- Wenn man sich nur für bestimmte Events interessiert, kann man die anderen Methoden als sog. „Attrappen"-Methoden (dummy method) mit leerem Rumpf implementieren,
  - da keine der in den Listener-Interfaces spezifzierten Methoden einen Rückgabewert besitzt.

Adapter-Klassen

Um dem Programmierer die Arbeit zu erleichtern, stellt das AWT für alle Interfaces, die mehr als eine Methode definieren, eine Adapter-Klasse zur Verfügung.

- Adapter-Klassen haben alle Interface-Methoden als leere Methoden implementiert.
- Bei der Verwendung von Adapter-Klassen müssen also nur die benötigten Methoden überschrieben werden.
- Die eigene Klasse muss von der Adapter-Klasse abgeleitet werden, so dass Adapter-Klassen nur dann verwendet werden können, wenn die eigene Klasse nicht von einer anderen Klasse erben soll.

Beispiel: Rahmen zum Zeichnen reeller Funktionen

Wir definieren eine abstrakte Klasse FunctionPlotter, mit deren Hilfe wir eine mathematische Funktion zeichnen können,

- genauer gesagt eine eindimensionale Funktion *f: R → R*

Diese Klasse kann insbesondere zur Veranschaulichung und Auswertung von Experimenten verwendet werden.

- Sie eignet sich auch sehr gut als Basis für eigene Erweiterungen.

Da wir beliebige ein-dimensionale reelle Funktionen zeichnen wollen, definieren wir in der Klasse FunctionPlotter eine abstrakte Funktion f: double → double .

Beispiel: Rahmen zum Zeichnen reeller Funktionen

Die Klasse FunctionPlotter wird von der Klasse java.awt.Frame abgeleitet.

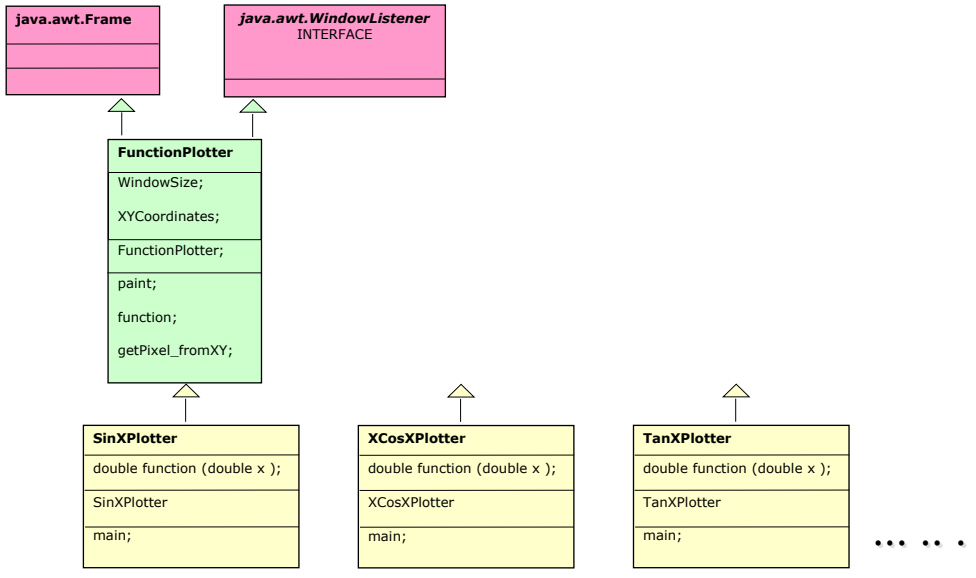
Die virtuelle Funktion paint ist diejenige Funktion in der Klasse Frame, die vom Windows-System aufgerufen wird, wenn ein Frame-Objekt gezeichnet wird .

- Wir überschreiben daher diese Methode in FunctionPlotter.

In unserer Implementierung der Methode paint benutzen wir die rein abstrakte Funktion f .

- Damit in FunctionPlotter diese abstrakte Methode durch verschiedene Realisierungen ersetzt werden kann, ohne dass paint() reimplementiert werden muss, hat paint() also die Funktion f als einen (impliziten) Parameter.
  - Dieser ist nicht in der Aufrufschnittstelle sichtbar.
- Die Methode paint ist somit eine **Funktion höherer Stufe** (higher-order function).

Beispiel: FunctionPlotter



Beispiel: Code der abstrakten Klasse FunctionPlotter

```
import java.awt.* ;

public abstract class FunctionPlotter
    extends java.awt.Frame {

   private int noSamples; // samples to be plotted
   private double minx;    // minimal plotted x-Value
      ...........
   private int winx;         // horiz. window size in pixel
   private int winy;         // vertical window size in pixel
    // Constructor  for the diplay domain
    public FunctionPlotter ( int _noSamples,
                             int _winx, int _winy,
                              double _minx,   double _maxx,
                              double _miny,   double _maxy) {
                .............
                setSize (winx,winy);
                setVisible(true);
   }
    // Transforanitons from World to Pixel Koordinates
    public int getPixelXfromWorldX( double x){
                return (int) ( (x - minx) / (maxx - minx)* winx)  ;
   }
   public int getPixelYfromWorldY( double y){ ..... }
    /**
    * Abstract method to be implemented in child classes
    * to evaluate the function to be plotted
    */
    public abstract double f ( double x);
```

```
/** This method is invoked automatically by the
*  runtime-environment whenever the contents
*  of the window is displayed on the screen and
*  (once everytime a redraw event occures)
*  plotting of the graph of f(x) is implemented here
*/
public void paint (Graphics g) {

   int i;
   int x1,x2,y1,y2;
   double step = (maxx - minx) / (noSamples -1);
   double x = minx;
   g.setColor( Color.red);

   x2 = getPixelXfromWorldX( x );
   y2 = getPixelYfromWorldY( f( x )  );

   for(i = 1; i< noSamples; i++){
              x+= step;
              x1 = x2; y1 = y2;
              x2 = getPixelXfromWorldX (  x     );
              y2 = getPixelYfromWorldY   ( f(x) );

              g.drawLine(x1,y1,x2,y2);
   }
 }
}
```

Beispiel: Rahmen zum Zeichnen reeller Funktionen

Um eine Funktion, wie z. B. x → x*cos(x) plotten zu können, müssen wir nur eine Klasse definieren,

- die FunctionPlotter erweitert,
- und in der f durch die gewünschte Funktion implementiert ist.

Außer einem entsprechenden Konstruktor muss in dieser Klasse keine weitere Methode definiert sein.

Beispiel: Rahmen zum Zeichnen reeller Funktionen

Beispiele zweier erweiternder Klassen

```
/**
 * Concrete Class which implements function f()
 * to calculate x times cos(x)
 */

public class XCosXPlotter
   extends FunctionPlotter {

   public XCosXPlotter( int nSamples,
                        int _winx, int _winy,
                        double  _minx,  double  _maxx,
                        double  _miny,  double  _maxy )   {
         super( nSamples, _winx,_winy, _minx, _maxx, _miny,_maxy);
   }

   public double f(double x) { return    x* Math.cos(x);   }

   public static void main( String args[]) {
          new XCosXPlotter( 50, 300,200, 0.0, 10.0, -1.5, 1.5);
   }
}
```

Beispiel: Rahmen zum Zeichnen reeller Funktionen

Damit das Graphik-Fenster, der Klasse FunctionPlotter, wieder geschlossem werden kann, muss die Methode windowClosing des Interfaces WindowListener implementiert werden.

- Es müssen alle in WindowListener deklarierten Methoden implementiert werden, egal ob wir auf die entsprechenden Ereignisse reagieren wollen oder nicht.
- Die nicht benötigten Methoden implementieren wir mit leerem Rumpf (dummy method).

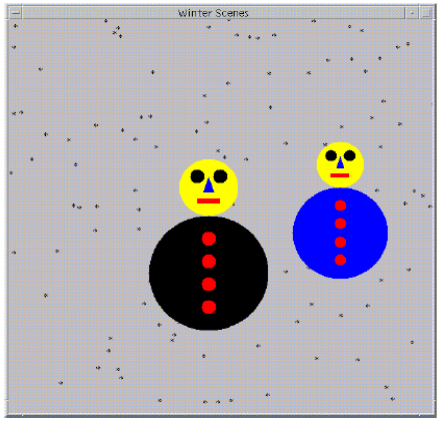
Beispiel: Code der abstrakten Klasse FunctionPlotter

```
import java.awt.* ;
import java.awt.event.WindowListener;
import java.awt.event.WindowEvent;

public abstract class FunctionPlotter
    extends java.awt.Frame
              implements WindowListener{

  private int noSamples; // samples to be plotted
  private double minx;    // minimal plotted x-Value
    ...........
  // Constructor  for the display domain
  public FunctionPlotter ( int _noSamples,
                           ..........) {
  //          .............
              setSize (winx,winy);
              setVisible(true);
              addWindowListener(this);

   }
  // Attribute and Methods for Function Plotter
  // .......
public abstract double f( double x);
```

```
public void paint (Graphics g) {
      .............
   // implementation of paint
}

// The implemention of WindowListener
/** Event handler  to process
  *  WINDOW_DESTROY events, which sould
   * terminate the program
  */

public void windowClosing(WindowEvent e) {
              setVisible( false) ;
              dispose();            //  destroy the window
              System.exit(0);      //  terminate execution
       }
 public void windowActivated (WindowEvent e){ }
 public void windowClosed (WindowEvent e){ }
 public void windowDeiconified (WindowEvent e){ }
 public void windowIconified (WindowEvent e){ }
 public void windowOpened(WindowEvent e){  }
 public void windowDeactivated (WindowEvent e){}
}
```

Beispiel2: Darstellung einer Winterlandschaft

Wir wollen eine Klassenbibliothek schreiben, die es uns ermöglicht, eine Winterlandschaft, wie sie rechts dargestellt ist, in einem Fenster zu zeichnen.



Beispiel: Darstellung einer Winterlandschaft

Anforderungsanalyse

- Die Klassenbibliothek soll verschiedene Objekte, die zu einer Winterlandschaft gehören - wie etwa Schneemänner und Schneeflocken - in einem Fenster darstellen.
- Ein Schneemann besteht aus einem Gesicht und einem Bauch.
  - Der Bauch eines Schneemanns ist rund und enthält einige runde Knöpfe.
  - Das runde Gesicht besteht aus zwei kreisförmigen Augen, einem rechteckigen Mund und einer dreieckigen Nase.

Beispiel: Darstellung einer Winterlandschaft

Objektorientierte Analyse und Design:

- Analyse relativ einfach.
- Konzentrieren uns hier auf die Verfeinerung in der Design-Phase,
  - Implementierung folgt später.
- Die Landschaft als Objekt einer Klasse SnowScene,
  - enthält Graphik-Fenster, in das gezeichnet wird,
  - und eine Liste von geometrischen Objekten.
- Wir benötigen ein Graphik-Fenster, die Klasse DrawWindow, welche die Klasse Frame aus java.awt erweitert.
- Die abstrakte Funktion paint von Frame müssen wir in DrawWindow implementieren.
- Damit das Fenster, per Mausklick geschlossen werden kann, realisiert DrawWindow auch noch das Interface WindowListener aus java.awt.event.

Beispiel: Darstellung einer Winterlandschaftv

Objektorientierte Analyse und Design (Forts.)

- Um die Szenen einfach erweitern zu können, wird sie als beliebige Liste von geometrischen Objekten gezeichnet.
- Von einer abstrakten Basisklasse Shape, werden alle geometrischen Objekte in der Landschaft abgeleitet.
  - Diese Klasse besitzt die (abstrakte) Methode draw, die in den Klassen, Shape entsprechend implementiert werden.

    Da diese Methode eine virtuelle Funktion darstellt, können wir die paint Methode in DrawWindow implementieren, ohne alle möglichen Erweiterungen der Klasse Shape zu kennen!

Beispiel: Darstellung einer Winterlandschaft

Objektorientierte Analyse und Design (Forts.)

- Die Liste der geometrischen Objekte vom Typ Shape realisieren wir als einfach verkettete lineare Liste.
- Entsprechend früher gewählten Namenskonvention hat diese Klasse den Namen ShapeList, und jede solche Liste besteht aus Knoten vom Typ ShapeNode .
- In unserer Winterlandschaft benötigen wir bislang Objekte vom Typ Snowman und vom Typ Snowflake.
- Ein Objekt vom Typ Snowman aggregiert zwei Felder,
  - eines für Body
  - und eines für Face .
- Diese beiden Klassen sind auch geometrische Objekte, die wir zeichnen wollen, sie sind also auch Erweiterungen der abstrakten Basisklasse Shape .

Beispiel: Darstellung einer Winterlandschaft

Objektorientierte Analyse und Design (Forts.)

- Ein Objekt vom Typ Body aggregiert einen Kreis für den eigentlichen Körper und einige (kleinere) Kreise für die Knöpfe.
- Wir benötigen daher auch eine Klasse Circle.
  - Da die Anzahl der Knöpfe nicht fest bestimmt ist, notieren wir eine allgemeine 1:*n*-Beziehung zwischen Body und Circle.
- Das Gesicht - ein Objekt vom Typ Face - wird durch einen umfassenden Kreis dargestellt, der zwei kleinere Kreise als Augen enthält.
  - Die Aggregation zwischen Face und Circle ist also vom Typ 1:3.
  - Das Gesicht enthält ferner einen rechteckigen Mund und eine dreieckige Nase.
  - Wir benötigen daher noch Erweiterungen von Shape zu einer Klasse Rectangle und zu einer Klasse Triangle.
    - Da ein Gesicht genau einen Mund und genau eine Nase enthält, sind diese Aggregationen vom Typ 1:1 .

Beispiel: Darstellung einer Winterlandschaft

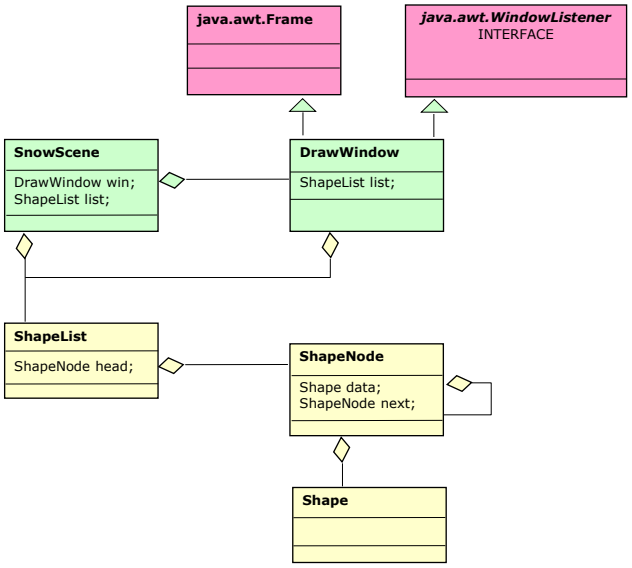
```
import java.awt.Point;
import java.awt.Color;

public class SnowScene {

  DrawWindow win;            // the window to paint
  ShapeList list;            // Shapes to be painted

 public SnowScene() {
    list = new ShapeList();     // create empty list
    // Adding two Snowman objects to the list
    Snowman sm1 = new Snowman(new Point(200, 475), 250);
    list.insertFirst(sm1);

   Snowman sm2 = new Snowman(new Point(400, 400), 200);
   sm2.body.setBodyColor(Color.blue);
   list.insertFirst(sm2);
  // Adding Snowflake objects to the list
   for(int i = 0; i < 120; i++) {
    list.insertFirst(new Snowflake((int)(600*Math.random()),
                                   (int)(600*Math.random())));
   }
    // Create a DrawWindow to paint the objects in the list
   win = new DrawWindow(600, 600, list);
 }
```

↓

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

        SnowScene sc = new SnowScene();
    }
}
```

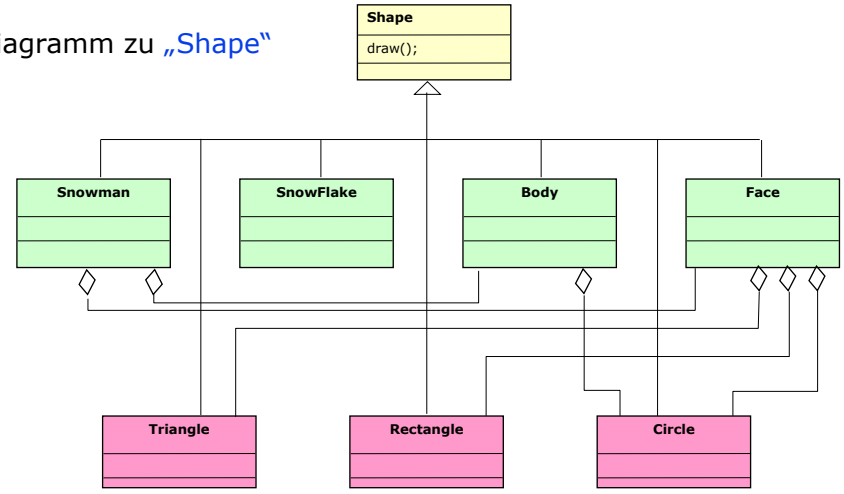
Beispiel: Darstellung einer Winterlandschaft

SnowScene



Beispiel: Darstellung einer Winterlandschaft

Klassendiagramm zu „Shape"



Beispiel: Darstellung einer Winterlandschaft

Bei der Implementierung der Shape-Klassen beginnen wir bei den abstrakten Basisklassen.

- Die Shape-Klassen, die keine anderen Shape-Klassen aggregieren, können danach unabhängig voneinander implementiert werden.
- Die Shape-Klassen, die andere aggregieren (wie etwa Face) sind sinnvollerweise erst danach zu implementieren.

Beispiel: Darstellung einer Winterlandschaft

Code für abstrakte Basisklasse Shape

```
import java.awt.Graphics;
import java.awt.Color;

/****
 * This is a class that defines an
 * abstract shape. It ensures that every subclass
 * implements a draw method.
 */
public abstract class Shape {
  protected Color color = Color.black; // color of the object

  /**
   * Gives back the color.
   * @return The color of the Shape.
   */
  public Color getColor() {
    return color;
  }
```

```
  /**
   * Sets the color of the Shape
   * @param col Nothing to say.
   */
  public void setColor(Color col) {
    color = col;
  }

  /**
   * An abstract method which ensures that every
   * subclass of Shape implements a draw method.
   * @param g Nothing to say.
   */
  public abstract void draw(Graphics g);


}
```

Beispiel Winterlandschaft: Face.java

```
import java.awt.Graphics;
import java.awt.Point;
import java.awt.Color;
/**
 * This extension of Shape defines the Face
 * of a Snowman with eyes, nose and mouth.
 */
public class Face extends Shape {
  private Circle face;     // the face itself
  private Circle leftEye;  // one eye
  private Circle rightEye; // the other eye
  private Triangle nose;   // the nose
  private Rectangle mouth; // the mouth
   /**
   * Creates a new Face.
   * @param midpoint Nothing to say.
   * @param radius Must be >= 0.
   */
  public Face(Point midpoint, int radius) {
   // The different components are placed relative
   // to the midpoints. Their sizes dependent only
   // on the radius.

   // Face
    face = new Circle(midpoint, radius);
   // Eyes
    Point leftEyePos = new Point(midpoint.x, midpoint.y);
    Point rightEyePos = new Point(midpoint.x, midpoint.y);

    leftEyePos.translate(-(int)(0.4*radius), -(int)(0.4*radius));
    rightEyePos.translate((int)(0.4*radius), -(int)(0.4*radius));
    leftEye = new Circle(leftEyePos, radius/4);
    rightEye = new Circle(rightEyePos, radius/4);

    leftEye.setColor(Color.yellow);
    rightEye.setColor(Color.yellow);

    // Nose
    Point p1 = new Point(midpoint.x, midpoint.y);
    Point p2 = new Point(midpoint.x, midpoint.y);
    Point p3 = new Point(midpoint.x, midpoint.y);

    p1.translate(-(int)(0.2*radius), (int)(0.2*radius));
    p2.translate((int)(0.2*radius), (int)(0.2*radius));
    p3.translate(0, -(int)(0.4*radius));

    nose = new Triangle(p1, p2, p3);
    nose.setColor(Color.blue);

    // Mouth
    Point min = new Point(midpoint.x, midpoint.y);
    Point max = new Point(midpoint.x, midpoint.y);

    min.translate(-(int)(0.4*radius), (int)(0.4*radius));
    .......
...
....
```

Beispiel Winterlandschaft: Circle.java

Kreise: Einfache Erweiterungen der Shape-Klasse

```
import java.awt.Graphics;
import java.awt.Point;
/**
 * This extension of Shape defines a Circle
 * by point and radius.
 */
public class Circle extends Shape {
  private Point m; // center
  private int r;   // radius
   /**
   * Creates a new Circle.
   * @param midpoint Nothing to say.
   * @param radius Must be >= 0.
   */
  public Circle(Point midpoint, int radius) {
    m = midpoint;
    r = radius;
  }
 /**
  * Draw method for Circle. Paints the Circle
  * as a filled oval.
  * @param g Nothing to say.
  */
  public void draw(Graphics g) {
    g.setColor(color);
    g.fillOval(m.x-r,m.y-r,2*r,2*r);
  }
}
```

Beispiel Winterlandschaft: Rectangle.java

Rechtecke

```
import java.awt.Graphics;
import java.awt.Point;
/**
 * This extension of Shape defines a Rectangle
 * by two corners.
 */
public class Rectangle extends Shape {
  private Point min;  // lower left corner
  private Point max;  // upper right corner
  /**
   * Creates a Rectangle through two points.
   * @param pmin The lower left corner.
   * @param pmax The upper right corner.
   */
  public Rectangle(Point pmin, Point pmax) {
    min = pmin;
    max = pmax;
  }
  /**
   * Draw method for Rectangle. The Rectangle is
   * painted and filled.
   * @param g Nothing to say.
   */
  public void draw(Graphics g) {
    g.setColor(color);
    g.fillRect(min.x,min.y,max.x-min.x,max.y-min.y);
  }
}
```

Beispiel Winterlandschaft: Face.java

Erweiterungen von Shape, die andere Shape-Objekte aggregieren

```
import java.awt.Graphics;
import java.awt.Point;
import java.awt.Color;
/**
 * This extension of Shape defines the Face
 * of a Snowman with eyes, nose and mouth.
 */
public class Face extends Shape {
  private Circle face;       // the face itself
  private Circle leftEye;    // one eye
  private Circle rightEye;   // the other eye
  private Triangle nose;     // the nose
  private Rectangle mouth; // the mouth
  /**
   * Creates a new Face.
   * @param midpoint Nothing to say.
   * @param radius Must be >= 0.
   */
  public Face(Point midpoint, int radius) {
    // The different components are placed relative
    // to the midpoints. Their sizes dependent only
    // on the radius.
```

```
    // Face
    face = new Circle(midpoint, radius);
    // Eyes
    Point leftEyePos   = new Point(midpoint.x, midpoint.y);
    Point rightEyePos = new Point(midpoint.x, midpoint.y);

    leftEyePos.translate(-(int)(0.4*radius), -(int)(0.4*radius));
    rightEyePos.translate((int)(0.4*radius), -(int)(0.4*radius));

    leftEye   = new Circle(leftEyePos, radius/4);
    rightEye = new Circle(rightEyePos, radius/4);

    leftEye.setColor(Color.yellow);
    rightEye.setColor(Color.yellow);
```

Beispiel Winterlandschaft: Face.java (2)

Fortsetzung des Codes zu Face

```
// Nose
    Point p1 = new Point(midpoint.x, midpoint.y);
    Point p2 = new Point(midpoint.x, midpoint.y);
    Point p3 = new Point(midpoint.x, midpoint.y);

    p1.translate(-(int)(0.2*radius), (int)(0.2*radius));
    p2.translate((int)(0.2*radius), (int)(0.2*radius));
    p3.translate(0, -(int)(0.4*radius));

    nose = new Triangle(p1, p2, p3);
    nose.setColor(Color.blue);

    // Mouth
    Point min = new Point(midpoint.x, midpoint.y);
    Point max = new Point(midpoint.x, midpoint.y);

    min.translate(-(int)(0.4*radius), (int)(0.4*radius));
    max.translate((int)(0.4*radius), (int)(0.6*radius));

    mouth = new Rectangle(min, max);
    mouth.setColor(Color.red);
  }
```

```
  /**
* Sets the color of the whole face.
* @param col Nothing to say.
*/
public void setColor(Color col) {
  super.setColor(col);
  face.setColor(col);
}
/**
* Sets the color for both eyes together.
* @param col Nothing to say.
*/
public void setEyeColor(Color col) {
  leftEye.setColor(col);
  rightEye.setColor(col);
}
/**
* Sets the color of the nose.
* @param col Nothing to say.
*/
public void setNoseColor(Color col) {
  nose.setColor(col);
}
/**
* Sets the color of the mouth.
* @param col Nothing to say.
*/
public void setMouthColor(Color col) {
  mouth.setColor(col);
}
```

Beispiel: Winterlandschaft

Fortsetzung des Codes zu Face

```
/**
   * Draw method for Face. Paints all components
   * face, eyes, nose, mouth.
   * @param g Nothing to say.
   */

  public void draw(Graphics g) {
    face.draw(g);
    leftEye.draw(g);
    rightEye.draw(g);
    nose.draw(g);
    mouth.draw(g);
  }
}
```

Beispiel: Winterlandschaft

Schneemann

- Der Schneemann besteht aus einem Gesicht und einem Kreis als Bauch.
- Diese Shape-Klasse soll hier ebenfalls nicht wiedergegeben werden.

Beispiel Winterlandschaft: DrawWindow.java

Ein Fenster mit einer Winterlandschaft

- Dem Konstruktor der folgenden Klasse DrawWindow kann neben der Größe des zu zeichnenden Fensters eine Liste von Shape-Objekten mitgegeben werden, die in dem Fenster gezeichnet werden.

```
import java.awt.Frame;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Event;
import java.awt.AWTEvent;
import java.awt.event.WindowListener;
import java.awt.event.WindowEvent;

/**
 * This class is a window for a ShapeList.
 * The Shapes are painted in a Frame.
 */
public class DrawWindow extends Frame implements
WindowListener {

  private ShapeList list;    // list of Shape elements
```

```
/**
   * Creates a new Frame for the Winter Scene.
   * @param width Width of the Frame. Must be >= 0.
   * @param height Height of the Frame. Must be >= 0.
   */
  public DrawWindow( int width, int height,
                     ShapeList elements) {
    // Frame parameters
    setSize(width, height);
    setTitle("Winter Scenes");
    setVisible(true);

    // this class handles the window events
    addWindowListener(this);

    // the ShapeList to be drawn
    list = elements;
  }
;
```

Beispiel Winterlandschaft: DrawWindow.java (2)

Fortsetzung des Codes von DrawWindow

```
/**
   * Paints the given list of Shapes.
   * @param g Nothing to say.
   */
  public void paint(Graphics g) {
    // trivial case
    if (list==null)
      return;
    // go through the non-empty list
    ShapeNode x = list.getHead();
    while(x != null) {
      x.getData().draw(g);   // draw the Shape
      x=x.getNext();         // switch to the next
    }
    return;
  }
```

```
/**
   * Closes the window if desired and exits the whole program.
   * @param e Nothing to say.
   */
  public void windowClosing(WindowEvent e) {
    setVisible(false);
    dispose();
    System.exit(0);
  }
  /** Empty implementation for interface WindowListener */
  public void windowActivated(WindowEvent e) { }
  public void windowClosed(WindowEvent e) { }
  public void windowDeiconified(WindowEvent e) { }
  public void windowIconified(WindowEvent e) { }
  public void windowOpened(WindowEvent e) { }
  public void windowDeactivated(WindowEvent e) { }
}
```

Beispiel: Darstellung einer Winterlandschaft

Hauptprogramm SnowScene

- Die Objekte einer ganzen Winterlandschaft können z. B. mit der main-Methode der folgenden Klasse SnowScene gezeichnet werden.
  - Neben zwei Schneemännern werden 120 Schneeflocken gezeichnet.

Beispiel: Darstellung einer Winterlandschaft

```
import java.awt.Point;
import java.awt.Color;

public class SnowScene {

  DrawWindow win;              // the window to paint
  ShapeList list;              // Shapes to be painted

public SnowScene() {
    list = new ShapeList();     // create empty list
    // Adding two Snowman objects to the list
    Snowman sm1 = new Snowman(new Point(200, 475), 250);
    list.insertFirst(sm1);

   Snowman sm2 = new Snowman(new Point(400, 400), 200);
   sm2.body.setBodyColor(Color.blue);
   list.insertFirst(sm2);
  // Adding Snowflake objects to the list
   for(int i = 0; i < 120; i++) {
    list.insertFirst(new Snowflake((int)(600*Math.random()),
                                   (int)(600*Math.random())));
   }
    // Create a DrawWindow to paint the objects in the list
   win = new DrawWindow(600, 600, list);
}
```

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

      SnowScene sc = new SnowScene();
  }
}
```