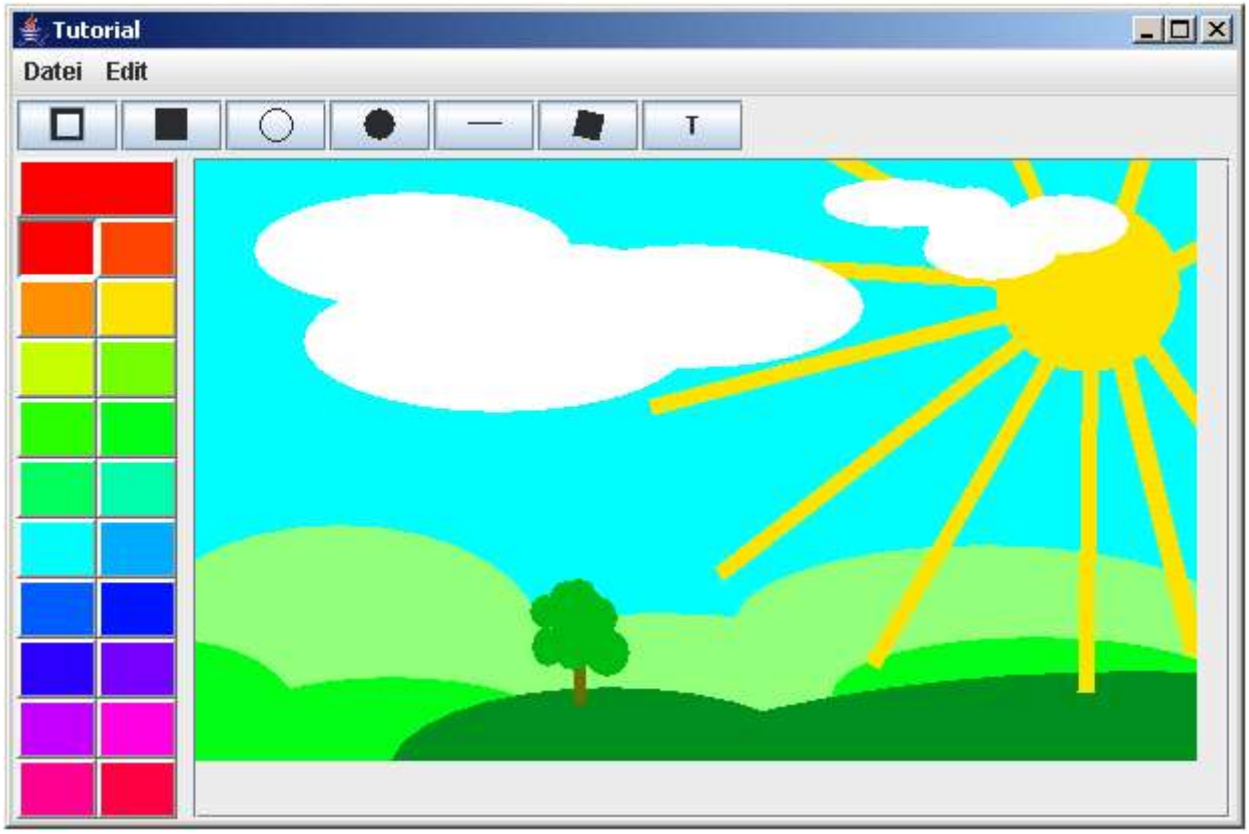
Ein Zeichenprogramm



Ein Java-Tutorial von Beni, Roar und L-ectron-X

©März 2005

Inhaltsverzeichnis

Vorwort....4
Vorwissen....4
Java 5.0....4
  Generics....4
  foreach....5
Was wollen wir?....6
Ein Wort zur Dokumentation....6
Die Spezifikation....7
Das Design....8
  Eine grobe Aufzählung....8
  Feinentwurf für ausgewählte Klassen und Interfaces....9
    FigureFactory....9
    FactoryListener....11
    Picture....11
    URManager....12
Implementierung....12
  Der Kern....12
    Figure....12
    FactoryListener....13
    FigureFactory....13
    PictureListener....15
    Picture....15
    URManager....17
  Eine Component um das Bild zu zeichnen....19
  Weitere graphische Elemente....22
    Wahl der Factory....22
    Wahl der Farbe....24
    Das Hauptmenü....35
    Das Hauptfenster....38
  public static void main....41
  Die Figuren....42
    AbstractFigureFactory....43
    AbstractFigure....45
    Rectangle....46
    FilledRectangle....48
    Aufgabe:....49
  Zwischenstop....50
Zusätzliche Features....51
  Speichern und Laden....51
    Factories und das Speichern/Laden....51
    PictureIO....52
    SimpleFileFilter....53
    FileManager....55
    Das Ganze einbauen....61
    It's not a bug, it's a feature....63
  Programm schließen....63
  Neues Bild herstellen....64
    Veränderungen in MainMenu....67
  Exportieren....67

Veränderungen in FileManager....68
Veränderungen in MainMenu....69
Eine JAR-Datei herstellen....70
Herstellen einer Dokumentation....71
Hacks und schmutzige Tricks....72
Neue Figuren....72
Das gedrehte Rechteck....73
RotatingRectangle....73
RotatingRectangleFactory....75
Text....79
Wie geht es weiter?....85
Anhang:....86
Was ist....86
... ein Interface?....86
... eine abstrakte Klasse?....87
... Swing?....88
... ein Input-/ OutputStream?....91
... so böse an static?....92
Copyright....93

Vorwort

Programmieren kann man nicht in 21 Tagen lernen, man benötigt Jahre dazu.

Sicher, es ist nicht weiter schwer ein Programm zu schreiben, welches „Hello World“ auf die Konsole schreibt, und danach 1+1 berechnet.
Das tiefere Verständnis, wie man Methoden schreibt, welche auch bei unerwarteten Ereignissen noch funktionieren, wie Klassen aufgebaut sind, die man für verschiedene Aufgaben verwenden kann, ..., kann man nur durch Erfahrung lernen.

Dieses Tutorial soll eine Erfahrung sein.

Wir werden gemeinsam ein kleines Zeichenprogramm schreiben. Mein Co-Autor Roar und ich werden dir jedes Mal erklären, wieso eine Klasse nach diesem oder jenem Schema aufgebaut ist, und Du wirst die Klasse dann schreiben (keine Angst, eine mögliche Lösung ist immer angegeben).

Was du dabei lernst? Ein bisschen abstraktes Denken, aber vor allem, wie man Klassen und Methoden aufbauen kann, so dass man fast keine Fehler macht, und so dass man sich nicht im Quellcode verirrt.

Vorwissen

Dieses Tutorial richtet sich an Anfänger, die aber ihr erstes Hello-World schon hinter sich haben.

Du solltest in der Lage sein *.java-Dateien kompilieren zu lassen, Schleifen zu schreiben, Methoden aufzurufen, ... mit anderen Worten: die Syntax und die Tools beherrschen.

Auch erste Erfahrungen mit dem Programmieren von graphischen Oberflächen, insbesondere mit Swing, wären recht wertvoll. Das ist keine Voraussetzung, allerdings wirst du bei einigen Teilen etwas Mühe haben, wenn du diese Dinge noch nie betrachtet hast.
Im Anhang gibt es ein sehr kurzes Kapitel über Swing.

Java 5.0

Dieses Tutorial baut auf Java 5.0 auf. Wir werden die Features von 5.0 aber nicht groß betrachten, nur zwei Sprachelemente werden benutzt.

Generics

Generics[1] geben uns die Möglichkeit *typsicher* zu arbeiten. Wir können z.B. einen *Vector* herstellen, der nur noch *Strings* aufnimmt.

Ein Beispiel wird das deutlich machen:

Ohne Generics

In einem *java.util.Vector* sollen nur *Strings* gespeichert werden. Ohne Generics ist es aber einfach einen *Integer* zu speichern, das Programm wird in der letzten Zeile abstürzen, weil ein *Integer* nicht in ein *String* gecastet werden kann.

```
Vector vector = new Vector();

vector.add( "Hallo" );
vector.add( new Integer( 2 ));
```

1 Tatsächlich eröffnen Generics noch weitere, neue Arten der Programmierung. Wir können ganze Algorithmen nur aus Generics schreiben, und uns das implementieren von Subklassen für bestimmte Probleme ersparen.

```
String string = (String)vector.get( 0 );
string = (String)vector.get( 1 );
 // ClassCastException während der Laufzeit
```

Mit Generics.

Wieder wollen wir nur *Strings* in dem *Vector* haben. Mit Hilfe der spitzen Klammern geben wir dies an.
Wenn wir nun versuchen, dem *Vector<String>* einen *Integer* hinzuzufügen, wird das nicht funktionieren. Der Compiler wird diesen Teil des Codes ganz einfach nicht kompilieren.
Einen anderen praktischen Effekt sehen wir in der letzten Zeile: um einen *String* aus dem *Vector* abzurufen, brauchen wir nicht mehr zu casten.

```
Vector<String> vector = new Vector<String>();

vector.add( "Hallo" );
vector.add( new Integer( 2 )); // Compilerfehler

String string = vector.get( 0 );
```

Das ist nicht der einzige Anwendungszweck von Generics, es gibt wesentlich komplexere Konstrukte... du musst nur mal die API-Doc ein bisschen durchsuchen.

Wie auch immer, wir werden das Thema Generics hier nicht vertiefen, uns soll der typsichere Vector reichen.

foreach

Ein syntaktisches Zückerlein von Java 5.0 ist die foreach-Schleife. Wenn wir gleich den *String-Vector* von eben betrachten: wie können wir alle *Strings* abrufen und ausgeben?
Die beiden Beispiel-Codes zeigen es (und ich muss wohl auch nichts mehr dazu schreiben).

Alte Variante

```
Vector<String> vector = new Vector<String>();

vector.add( "Hallo" );
vector.add( "Blupp" );

for( int i = 0, n = vector.size(); i<n; i++ ){
    System.out.println( vector.get( i ));
}
```

Neue Foreach-Variante

```
Vector<String> vector = new Vector<String>();

vector.add( "Hallo" );
vector.add( "Blupp" );

for( String s : vector ){
    System.out.println( s );
}
```

Was wollen wir?

Wir wollen ein kleines Programm schreiben, mit dem wir Bilder zeichnen können!

Die Frage scheint schon beantwortet zu sein, aber könntest du jetzt am PC sitzen, und den Code schreiben? Falls ja: dann ist dieses Tutorial nicht für dich geeignet, aber ich danke für die Aufmerksamkeit.

So, für diejenigen, die noch dabei sind: zuerst wollen wir mal ein bisschen genauer spezifizieren, was denn dieses Programm *genau* machen soll (und z.B. auch, was wir unter einem „Bild“ verstehen).

Ein Wort zur Dokumentation

Es gibt Leute, die nicht gerne Dokumentationen schreiben, weil sie das für verschwendete Zeit halten.

Diese Leute schreiben entweder immer sauberen und korrekten Code (maximal 0.1% aller Programmierer können dies), oder sie schreiben Programme, die besser nicht veröffentlicht werden.

Wir hingegen werden zu (fast) jeder Klasse, Methode und Variable einen Kommentar schreiben und so immer in der Lage sein, vor längerer Zeit geschriebenen Code später auch noch zu verstehen. Wenn ich für mich etwas programmiere, schreibe ich sogar zuerst die Dokumentation, und erst danach den Code, so kann ich meine Gedanken bündeln, und entdecke Unstimmigkeiten schon frühzeitig.

Das JDK (Java Development Kit) hat ein praktisches Tool mitgeliefert. Javadoc. Javadoc generiert aus speziell formatierten Kommentaren eine Dokumentation in HTML. Wir werden nicht alle Befehle benötigen, aber anhand eines Beispieles kurz die wichtigsten kennen lernen.

```
/**
 * Berechnet die Summe zweier Integer-Zahlen.<br>
 * Dabei wird folgende Formel angewandt: <code>a + b</code>.
 * @param a Der erste Summand
 * @param b Der zweite Summand
 * @return Die Summe von a und b
 */
public int plus( int a, int b ){
      return a + b;
}
```

Die einzelnen Sonderzeichen hier bedeuten:

- **
** Das ist die HTML-Notierung für „neue Zeile“. Alle HTML-Notierungen können in dem Kommentar verwendet werden.
- **<code></code>** Text der innerhalb von *code* steht, bekommt eine spezielle Formatierung.
- **@param** Wird der Name eines Argumentes übergeben (hier a oder b), und eine kleine Beschreibung, wozu das Argument dient.
- **@return** Beschreibt den Rückgabewert einer Methode.

Wenn wir nun Javadoc ausführen, wird dieser Kommentar umgewandelt in:

plus

```
public int plus( int a, int b)
```

Berechnet die Summe zweier Integer-Zahlen.
Dabei wird folgende Formel angewant: `a + b`

Parameters:
a - Der erste Summand
b - Der zweite Summand
Returns:
Die Summe von a und b

Wie Javadoc ausgeführt wird, werden wir später im Kapitel „Herstellen einer Dokumentation“ genauer betrachten.

Die Spezifikation

Was ist denn ein Bild? Für uns soll ein Bild eine Sammlung von Rechtecken, Kreisen, evtl. auch Buchstaben, und anderen Figuren sein, welche übereinander liegen. Unser Bild wird also nicht aus Pixeln, wie eine (Digital-) Fotografie, sondern aus Figuren, wie farbiges Papier das ausgeschnitten und übereinander geklebt wurde, bestehen.

Farbiges Papier? Ja klar: diese Figuren sollten natürlich verschiedene Farben haben! Also müssen wir sicher dem Benutzer eine Möglichkeit bieten, Farben auszuwählen.

Und natürlich brauchen wir irgendwas, was das Bild darstellt.

Was noch? Die Figuren müssen irgendwie vom Benutzer eingegeben werden können. Ein erster Gedanke ist: nachdem der Benutzer den Typ der Figur ausgewählt hat, drückt er mit der Maus an eine bestimmte Stelle auf dem Bildschirm, fährt mit gedrückter Maustaste herum, und lässt irgendwann wieder los. Dann erhalten wir zwei Punkte, und können z.B. ganz einfach ein Rechteck zeichnen.

Und wie soll das mit Text funktionieren (mit der Maus kann man nur schwer schreiben)? Tut es nicht! Wenn wir das so implementieren bauen wir uns eine Sackgasse.
Der Fehler den wir hier gemacht haben ist, dass wir bereits zu viel gedacht haben. Wieso sollen wir bereits jetzt bis ins kleinste Detail beschreiben, wie der Benutzer etwas eingibt? Wir müssen das Ganze abstrakter betrachten, und kommen zum einfachen Schluss: „Der Benutzer muss die Möglichkeit haben, Eingaben zu machen.“
Wie dies genau geschehen soll, ist für jede Figur unterschiedlich. Wir werden das also für jede Figur einzeln besprechen.

Sehen wir mal was noch fehlt: Man sollte diese Bilder natürlich speichern, und auch wieder laden können. Dazu werden wir ein eigenes Dateiformat definieren (was nicht weiter schwierig sein wird).

Nett wäre es auch, wenn wir die Bilder in „echte“ Bilder, also z.B. in ein JPEG umwandeln könnten.

Und dann kommt noch ein dicker Brocken, der unser Programm allerdings wesentlich aufwerten wird: die Möglichkeit Aktionen rückgängig zu machen (und sie später zu wiederholen).

Das Design

Eine grobe Aufzählung

Jetzt haben wir schon eine Ahnung, was unser Programm alles leisten wird.
Nun ist es an der Zeit zu überlegen, was wir alles implementieren müssen, um das Ganze auch auf den Bildschirm zu zaubern.

Wir designen jetzt unser Programm, d.h. wir schreiben die Klassen (und Interfaces) auf, welche wir später benötigen werden.

Würden wir ein großes Programm schreiben, wären die nächsten 200 Seiten für dieses Kapitel reserviert, und wirklich jede einzelne Variable würde aufgezählt werden.
Aber wir schreiben etwas kleines, was man sich auch im Kopf vorstellen kann (vielleicht braucht man dazu etwas Übung), deshalb werden wir nur die wichtigsten Dinge aufschreiben. Sozusagen das Grundgerüst, welches wir während dem Implementieren erweitern, verfeinern und fertig stellen.

Wir werden alle Klassen, Interfaces, Methoden, ... mit englischen Namen bezeichnen. Das hat gleich mehrere Vorteile: zum einen hat auch ein Engländer eine Chance unseren Quellcode zu verstehen, zum anderen ist dies ein deutsches Tutorial, und es wird Dir immer klar sein, wann ich von Variablen oder etc. spreche, und wann nicht.

Zählen wir doch einmal auf, was wir brauchen werden: Etwas, das unser Bild darstellt, das *Picture*.

Auf dem *Picture* muss etwas gezeichnet werden, und zwar unsere Figuren, welche wir unter einem Interface namens *Figure* zusammenfassen werden (was ein Interface ist, werde ich später noch erklären. Falls dir der Begriff nichts sagt: ersetze ihn erstmal durch „Klasse“).

Woher kommen eigentlich die Figuren? Jemand muss sich um die Eingabe, und um die Herstellung der verschiedenen Figuren kümmern. Das wird eine *FigureFactory* machen.

Wenn wir schon bei der Eingabe sind, wohin mit dem Rückgängig/Wiederholen-Feature? Dieses Feature wird sich auf das Entfernen, und auf das Wieder-Hinzufügen von Figuren beschränken. Es wäre nun möglich direkt in das *Picture* entsprechende Methoden einzubauen, allerdings ist das keine gute Idee.
Erinnerst du dich noch an die Beschreibung des Bildes? Bunte Papierfiguren werden auf ein Hintergrund-Papier geklebt.
Wenn man nun eine Figur wieder entfernen will, sagt man nicht zum Hintergrund-Papier „entferne bitte mal kurz die letzte Figur“, sondern man greift selbst zu.
Deshalb werden wir etwas ähnliches wie Hände bauen, nur dass diese Hände außer Figuren wegnehmen und hinzufügen nichts machen können. Dieses Konstrukt nennen wir den „Undo-Redo-Manager“, oder eine Kurzform: *URManager*.

Das führt uns sofort zum nächsten Problem: wie bemerkt der *URManager*, dass dem *Picture* eine *Figure* hinzugefügt wurde? Wir benötigen einen Mechanismus, der dem Bild die Fähigkeit gibt, andere Objekte zu unterrichten, sollte jemand eine Figur hinzufügen (oder auch entfernen).
Du hast es sicher schon bemerkt, ich habe geschrieben, dass ein „Objekt“, und nicht der *URManager*, benachrichtigt werden soll.
Das hat verschiedene Gründe: zum einen kann es passieren, dass sonst noch jemand vom Bild und seinen Figuren unterrichtet werden muss, zum anderen ist es für das Bild **absolut unwichtig**, wer unterrichtet wird.
Wenn man nicht gezwungen wird eine Abhängigkeit (hier *Picture* unterrichtet *URManager*) einzubauen, sollte man es lassen! Das Programm wird dadurch flexibler, da wir nun den *URManager* auch durch etwas anderes ersetzen können.

Dann benötigen wir noch eine ganze Menge graphische Oberflächen (wir werden hierzu Swing

verwenden), aber wenn wir erst einmal den Kern des Programmes geschrieben haben, wird die GUI (graphical user interface) kein Problem mehr sein. Deshalb verzichten wir auf ein detailliertes Design der Oberfläche.

Feinentwurf für ausgewählte Klassen und Interfaces

Nun werden wir den Feinentwurf für 4 Interfaces, und 2 Klassen machen. Diese 6 Elemente werden den Kern unseres Programmes bilden.

Vielleicht musst du allerdings zuerst im Anhang nachlesen, was ein Interface ist!

Dann beginnen wir doch endlich mit dem Feinentwurf!

Wir werden wohl oft mit Figuren arbeiten müssen. Deshalb kommt dieses Interface zuerst an die Reihe.

Figure

- Wir benötigen eine Möglichkeit die Figur zu zeichnen. Wir werden *nicht* eine endlose if-else-Kette hinschreiben, sondern das Zeichnen an die Figuren delegieren.
  Dabei werden wir ein Objekt des Typs *Graphics* verwenden. *Graphics* befindet sich im Package *java.awt* und bietet die Möglichkeit verschiedenste Figuren auf den Bildschirm zu malen.
  public void paint(Graphics g)
- Um Bilder zu speichern und zu laden werden wir gewiss die Farbe der einzelnen Figuren benötigen, also muss die Farbe abrufbar sein.
  Für Farben gibt es die Klasse *java.awt.Color*, welche wir verwenden werden.
  public void setColor(Color color)
  public Color getColor(Color color)
- Für das Laden und Speichern des Bildes wird es auch nützlich sein, die Herkunft einer Figur zu kennen. Wieso? Angenommen wir Laden eine Figur. Nun haben wir also die Bits welche die Figur beschreiben. Wenn wir außerdem wissen, wer die Figur das letzte Mal hergestellt hat, können wir die Bits diesem Jemand übergeben, und er wird die Figur für uns korrekt zusammenbauen (wiederum ohne irgendwelche gigantischen if-else-Blöcke).
  public FigureFactory getOrigin()

Wenn wir schon bei Figuren sind, machen wir gleich mit der *FigurFactory* weiter.

FigureFactory

Die Factory wird die Figuren herstellen. Dies geschieht entweder durch Eingaben des Benutzers, oder durch das Lesen eines Teiles einer Datei.
Wie soll das mit den Eingaben funktionieren? Es ist dir wahrscheinlich bekannt, wie man Mausereignisse verarbeiten kann: man übergibt einer *Component* einen *MouseListener* (bzw. ein *MouseMotionListener*). Dann werden spezielle Methoden aufgerufen, in welchen man irgendetwas tun kann.

Falls du nicht weist, wie ein *MouseListener* funktioniert, im Anhang im Swing-Kapitelchen gibt es ein Beispiel.

Eine *FigureFactory* wird also folgende Eigenschaften haben müssen:

- Jede Factory muss auch ein *MouseListener* und *MouseMotionListener* sein. Die Eingaben des Benutzers werden wir so direkt an die Factory weiterleiten, welche die nächste Figur herstellen

soll.

- Wahrscheinlich möchte der Benutzer beobachten, wie er etwas Neues herstellt. Deshalb geben wir jeder Factory die Möglichkeit direkt auf unser Bild zu malen.
  public void paint(Graphics g)
- Wenn die Factory schon etwas zeichnet, wahrscheinlich eine Vorschau der Figur, welche sie herstellt, sollte die Factory auch wissen, welche Farbe die Figur schlussendlich erhalten wird.
  public void setColor(Color color)
- Dann muss die Factory ihre hergestellten Figuren irgendwie dem Bild übermitteln. Die wohl einfachste Variante ist hierzu: das Bild der Factory übergeben, und der Klasse *Picture* eine Methode einbauen, mit welcher man Figuren hinzufügen kann.
  Man kann argumentieren, dass dies eine zu große Abhängigkeit ist, und man der Factory nur ein Interface übergeben sollte (welches von *Picture* implementiert wird), bei der Größe unseres Programms wäre das allerdings übertrieben.
  public void setPicture(Picture picture)
- Sollte der Benutzer plötzlich auf die Idee kommen, eine neue Factory auszuwählen, noch während die alte Factory Eingaben erwartet, muss diese Factory irgendwie davon erfahren. Überhaupt wäre es sinnvoll, wenn eine Factory wüsste, ob der Benutzer derzeit mit ihr arbeitet. Deshalb kommt folgende Methode hinzu:
  public void setActive(boolean active)
- Wir wollen dem Benutzer die Möglichkeit bieten, Dateien zu schreiben und zu lesen.
  Das Speichern und Laden einzelner Figuren wird jeweils die Factory übernehmen, welche die Figur hergestellt hat.
  Wir werden den *DataInput*- und den *DataOutputStream* aus dem Package *java.io* verwenden, da wir mit dieser Klasse sehr einfach Variablen wie Integer oder Double speichern können.
  Im Anhang stehen noch ein paar Zusatzinformationen zu Streams.
  public void write(Figure figure, DataOutputStream out)
  public Figure read(DataInputStream in)
- Um die Dateien später wieder zu lesen, müssen wir die benutzte Factory kennen. Deshalb soll jede Factory eine ID besitzen; ein String, der für jede Factory anders ist, und der niemals verändert wird.
  public String getID()
- Wir werden dem Benutzer eine Reihe von Knöpfen anbieten, mit welchen er die Factory auswählen kann, welche die nächste Figur herstellen soll.
  Dazu benötigen wir ein Icon und einen Namen für jede Factory
  public Icon getIcon()
  public String getName()
- Die Factory zeichnet Dinge, die sich verändern. Deshalb muss es eine Möglichkeit geben, die Factory „abzuhören“, und herauszufinden, wann die Darstellung der Factory erneuert werden muss.
  Wir werden der Factory ein Objekt mit nur einer Methode übergeben. Dieses Objekt ist vom Typ *FactoryListener*, und die Methoden werden wir aufrufen, wenn die Factory in ihrer *paint*-Methode etwas anderes zeichnet als bisher.
  public void addFactoryListener(FactoryListener listener)
  public void removeFactoryListener(FactoryListener listener)

FactoryListener

Du kennst ja schon den *MouseListener*, nun werden wir selbst einen neuen Typ Listener definieren.

Das ist nicht weiter schwer, da Listener auch nur Interfaces sind.

- Dieser Listener soll benachrichtigt werden, wenn eine Factory anders gezeichnet wird (weil der Benutzer z.B. eine Eingabe getätigt hat). Dazu reicht eine einzige Methode.
  public void shouldRepaint(FigureFactory factory)

Wir haben bereits früher gesagt, dass das Picture seiner Aussenwelt irgendwie mitteilen muss, wenn etwas mit ihm geschieht.
Deshalb werden wir auch einen Listener für das *Picture* schreiben.

PictureListener

- Wir benötigen eine Methode um dem Listener mitzuteilen, dass dem Bild eine Figur hinzugefügt wurde.
  public void figureAdded(Figure figure, Picture picture)
- Und dann schreiben wir auch gleich das Gegenteil hin: eine Methode die wir aufrufen, wenn ein Bild entfernt wird.
  public void figureRemoved(Figure figure, Picture picture)

Nun kommen wir zu dem Bild selbst.

Picture

- Zuerst einmal muss eine Methode daher, das Bild zeichnen.
  public void paint(Graphics g)
- Wir müssen dem Bild die Fähigkeiten geben, Figuren aufzunehmen und auch wieder zu entfernen.
  public void add(Figure figure)
  public void remove(Figure figure)
- Um das Bild auf einfachere Weise zu kontrollieren, schreiben wir auch Methoden welche Zugriffe auf die Figuren mittels Indizes erlauben.
  public Figure get(int index)
  public void remove(int index)
- Wenn wir irgendwann einmal mit Indizes arbeiten, müssen wir unbedingt die Möglichkeit haben, herauszufinden, wie viele Figuren in dem Bild sind.
  public int getSize()
- Und dann muss da noch eine Möglichkeit existieren, unseren Listener zu registrieren, und auch wieder zu entfernen.
  public void addPictureListener(PictureListener listener)
  public void removePictureListener(PictureListener listener)
- Wir sollten dem Bild auch noch eine Größe geben, das ist besonders für eine spätere Umwandlung in ein echtes Bild praktisch.
  public int getWidth()
  public int getHeight()

Nun fehlt noch die Klasse, welche es uns ermöglichen wird, Aktionen rückgängig zu machen: der *URManager*.

URManager

- Um auf die wichtigsten zwei Methoden zu kommen sind wohl keine langen Erklärungen notwendig.
  public void undo()
  public void redo()
- Die graphische Oberfläche wird wohl zwei Knöpfe anbieten, auf die der Benutzer drücken kann, um ein Undo- oder ein Redo-Schritt durchzuführen. Es gibt aber durchaus Situationen, in denen dies nicht möglich ist. Darum muss der Manager Angaben machen können, ob undo/redo ausgeführt werden kann.
  public boolean isUndoPossible()
  public boolean isRedoPossible()
- Und dann dürfte der Manager sehr interessiert sein, welches Bild denn gerade bearbeitet wird:
  public void setPicture(Picture picture)
- Wir sollten auch daran denken, dass der Manager von Veränderungen im *Picture* unterrichtet werden möchte. Deshalb muss er das Interface *PictureListener* implementieren.

Implementierung

Der Kern

Nun ist es an der Zeit, die soeben beschriebenen Interfaces und Klassen zu implementieren.

Diese 5 Elemente bilden den Kern unseres Programmes, und deshalb werden wir sie in einem Package Namens *tutorial.core* ablegen. Packages sind die Java-Darstellung von Verzeichnissen. Das bedeutet also nichts anderes, als dass wir die 5 Dateien in einem Verzeichnis *tutorial/core* speichern werden.

Das Beste ist natürlich, wenn Du gleich selbst versuchst, die 5 Dateien mit sinnvollem Code zu füllen. Als Hilfe schreibe ich Dir aber gleich hier den Code für die Elemente hin.

Figure

```
package tutorial.core;

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

/**
 * Eine Figur stellt ein gezeichnetes Element auf einem Bild dar
 */
public interface Figure{
    /**
     * Zeichnet diese Figure.
     * @param g Das Graphics-Object, mit dem gezeichnet werden soll
     */
    public void paint( Graphics g );

    /**
     * Gibt die Farbe zurück, mit welcher diese Figure gezeichnet wird.
     * @return Die Farbe dieser Figure
     */
    public Color getColor();
```

```
    /**
     * Setzt die Farbe, mit der diese Figure gezeichnet werden soll.
     * @param color Die Farbe der Figure
     */
    public void setColor( Color color );

    /**
     * Gibt die FigureFactory zurück, welche diese Figure hergestellt hatte.
     * @return Die Herkunft dieser Figure
     */
    public FigureFactory getOrigin();
}
```

FactoryListener

```
package tutorial.core;

/**
 * Dieser Listener wird bei einer FigureFactory registriert.
 */
public interface FactoryListener{
    /**
     * Teilt dem Listener mit, dass die graphische Darstellung der übergebenen
     * Factory
     * verändert wurde und dass die Factory neu gezeichnet werden sollte.
     * @param factory Die veränderte Factory
     */
    public void shouldRepaint( FigureFactory factory );
}
```

FigureFactory

```
package tutorial.core;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

import javax.swing.Icon;

/**
 * Erstellt einen Typ von Figuren<br>
 * Die Factory implementiert den MouseListener und den MouseMotionListener. Die
 * Methoden
 * dieser Listener wird aufgerufen, wenn der Benutzer die Factory verwendet,
 * dann sollte
 * die Factory auf ihre Weise auf die Eingaben reagieren.
 */
public interface FigureFactory extends MouseListener, MouseMotionListener{
    /**
     * Diese Methode wird nur dann aufgerufen, wenn diese Factory aktiv ist
     * (dies ist abhängig vom letzten Übergabewert von
     * {@link #setActive(boolean)}).<br>
     * Die Factory kann Informationen zeichnen, welche dem Benutzer helfen,
     * seine
     * Eingaben zu machen.
     * @param g Das Graphics-Object das benutzt werden soll
     */
    public void paint( Graphics g );
```

```
    /**
     * Unterrichtet die Factory, dass der Benutzer eine neue Farbe ausgewählt
     * hat.<br>
     * Die Factory sollte nur Figuren herstellen, welche die übergebene Farbe
     * benutzen.
     * @param color Die Farbe, welche der Benutzer ausgewählt hat.
     */
    public void setColor( Color color );

    /**
     * Setzt das Bild, welches verwendet wird. Diesem Bild muss die Factory neue
     * Figuren hinzufügen.
     * @param picture Das zu verwendende Bild oder null
     * @see Picture#add(Figure)
     */
    public void setPicture( Picture picture );

    /**
     * Unterrichtet die Factory, ob sie vom Benutzer verwendet wird, oder
     * nicht.<br>
     * Sollte der Wert <code>true</code> übergeben werden, muss die Factory so
     * justiert werden, dass der Benutzer mit der Eingabe neuer Figuren
     * beginnen kann.
     * @param active true, wenn der Benutzer die Factory verwenden kann,
     * andernfalls false.
     */
    public void setActive( boolean active );

    /**
     * Schreibt die Eigenschaften der übergebenen Figur in den OutputStream.
     * Nicht gespeichert werden muss die Farbe.
     * @param figure Die zu schreibende Figur
     * @param out Der zu benutzende OutputStream
     * @throws IOException Bei Problemen
     */
    public void write( Figure figure, DataOutputStream out ) throws IOException;

    /**
     * Liest die Eigenschaften einer zuvor geschriebenen Figur aus dem
     * InputStream aus.
     * @param in Der InputStream mit den Informationen über die Figur
     * @return Eine neue Instanz einer Figure
     * @throws IOException Bei Problemen
     */
    public Figure read( DataInputStream in ) throws IOException;

    /**
     * Gibt ein Icon zurück, welches diese Factory charakterisiert.
     * @return Ein Icon der Größe 16x16 Pixel
     */
    public Icon getIcon();

    /**
     * Gibt den Namen dieser Factory zurück. Unter diesem Namen wird die Factory
     * dem Benutzer präsentiert.
     * @return Der Name der Factory
     */
    public String getName();

    /**
     * Gibt einen String zurück, welcher diese Factory identifiziert. Keine
     * andere Factory sollte dieselbe ID besitzen.
     * @return Die ID
     */
    public String getID();
```

```
    /**
     * Fügt der Factory einen Listener hinzu.
     * @param listener Der neue Listener
     */
    public void addFactoryListener( FactoryListener listener );

    /**
     * Entfernt ein früher registrierter Listener der Factory.
     * @param listener Der zu entfernende Listener
     */
    public void removeFactoryListener( FactoryListener listener );
}
```

PictureListener

```
package tutorial.core;

/**
 * Der PictureListener wird unterrichtet, wenn der Inhalt eines Pictures
 * verändert wurde
 */
public interface PictureListener{
    /**
     * Wird aufgerufen, wenn einem Picture eine Figur hinzugefügt wurde.
     * @param figure Die neue Figur
     * @param picture Das veränderte Bild
     */
    public void figureAdded( Figure figure, Picture picture );

    /**
     * Wird aufgerufen, wenn einem Picture eine Figur entfernt wurde.
     * @param figure Die entfernte Figur
     * @param picture Das veränderte Bild
     */
    public void figureRemoved( Figure figure, Picture picture );
}
```

Picture

```
package tutorial.core;

import java.awt.Color;
import java.awt.Graphics;
import java.util.Vector;

/**
 * Stellt ein Bild mit verschiedenen Figuren dar.
 */
public class Picture{
    /** Eine Liste aller Figuren dieses Bildes */
    private Vector<Figure> figures = new Vector<Figure>();

    /** Eine Liste aller Listener dieses Bildes */
    private Vector<PictureListener> listeners = new Vector<PictureListener>();

    /** Die Breite dieses Bildes*/
    private int width;

    /** Die Höhe dieses Bildes */
    private int height;

    /**
     * Standardkonstruktor.
     * @param width Die Breite dieses Bildes, mindestens 1
     * @param height Die Höhe dieses Bildes, mindestens 1
```

```
     */
    public Picture( int width, int height ){
        if( width < 1 || height < 1 )
            throw new IllegalArgumentException( "width und height müssen" +
                  "wenigstens die Größe 1 haben" );

        this.width = width;
        this.height = height;
    }

    /**
     * Zeichnet dieses Bild.
     * @param g Das Graphics-Objekt, das benutzt wird
     */
    public void paint( Graphics g ){
        g.setColor( Color.WHITE );
        g.fillRect( 0, 0, width, height );

        for( Figure f : figures )
            f.paint( g );
    }

    /**
     * Gibt die Breite dieses Bildes zurück.
     * @return die Breite
     */
    public int getWidth() {
        return width;
    }

    /**
     * Gibt die Höhe dieses Bildes zurück.
     * @return Die Höhe
     */
    public int getHeight() {
        return height;
    }

    /**
     * Fügt diesem Bild eine Figur hinzu. Diese Figur wird zu oberst gezeichnet.
     * @param figure Die neue Figur
     */
    public void add( Figure figure ){
        figures.add( figure );
        fireFigureAdded( figure );
    }

    /**
     * Entfernt eine Figur von diesem Bild.
     * @param figure Die zu entfernende Figur
     */
    public void remove( Figure figure ){
        if( figures.remove( figure ) )
            fireFigureRemoved( figure );
    }

    /**
     * Entfernt eine Figur von diesem Bild.
     * @param index Der Index der zu entfernenden Figur
     */
    public void remove( int index ){
        Figure figure = figures.remove( index );
        fireFigureRemoved( figure );
    }
```

```
    /**
     * Gibt die Anzahl Figuren zurück, welche in diesem Bild
     * dargestellt werden.
     * @return Die Anzahl Figuren
     */
    public int getSize(){
        return figures.size();
    }

    /**
     * Gibt die Figur mit dem angegebenen Index zurück.
     * @param index Der Index
     * @return Die Figur zum Index
     */
    public Figure get( int index ){
        return figures.get( index );
    }

    /**
     * Unterrichtet die Listener, dass eine Figur hinzugefügt wurde.
     * @param figure Die neue Figur
     */
    protected void fireFigureAdded( Figure figure ){
        for( PictureListener p : listeners )
            p.figureAdded( figure, this );
    }

    /**
     * Unterrichtet die Listener, dass eine Figur entfernt wurde.
     * @param figure Die entfernte Figur
     */
    protected void fireFigureRemoved( Figure figure ){
        for( PictureListener p : listeners )
            p.figureRemoved( figure, this );
    }

    /**
     * Fügt dem Bild einen Listener hinzu.
     * @param listener Der Listener
     */
    public void addPictureListener( PictureListener listener ){
        listeners.add( listener );
    }

    /**
     * Entfernt einen Listener von diesem Bild.
     * @param listener Der zu entfernende Listener
     */
    public void removePictureListener( PictureListener listener ){
        listeners.remove( listener );
    }
}
```

URManager

```
package tutorial.core;

import java.util.Stack;

/**
 * Bietet die Möglichkeit Figuren die einem Bild hinzugefügt wurden wieder
 * zu entfernen, und später neu hinzuzufügen.
 */
public class URManager implements PictureListener{
    /** Die Figuren welche aufgrund von undo-Aktionen entfernt wurden */
```

```
    private Stack<Figure> removedFigures = new Stack<Figure>();

    /** Das Bild, auf das sich der URManager bezieht */
    private Picture picture = null;

    /** Eine Flag welche auf true gesetzt wird, wenn der URManager etwas an dem
        Bild verändert */
    private boolean urCall = false;

    /**
     * Defaultkonstruktor
     */
    public URManager(){
    }

    /**
     * Setzt das Bild, welches der Manager überwachen soll.
     * @param picture Das Bild oder null
     */
    public void setPicture( Picture picture ){
        if( this.picture != picture ){
            if( this.picture != null ){
                this.picture.removePictureListener( this );
                removedFigures.clear();
            }

            this.picture = picture;

            if( picture != null ){
                picture.addPictureListener( this );
            }
        }
    }

    /**
     * Entfernt das oberste Element des Bildes, sofern ein Bild
     * vorhanden ist
     */
    public void undo(){
        if( isUndoPossible() ){
            try{
                urCall = true;
                int index = picture.getSize()-1;
                Figure figure = picture.get( index );
                removedFigures.push( figure );
                picture.remove( index );
            }
            finally{
                urCall = false;
            }
        }
    }

    /**
     * Fügt die zuletzt entfernte Figur wieder dem Bild hinzu.
     */
    public void redo(){
        if( isRedoPossible() ){
            try{
                urCall = true;
                Figure figure = removedFigures.pop();
                picture.add( figure );
            }
            finally{
                urCall = false;
```

```
            }
        }
    }

    /**
     * Gibt an, ob ein Aufruf von {@link #undo()} einen Effekt ergeben wird.
     * @return true, wenn ein Aufruf von <code>undo</code> sinnvoll ist.
     */
    public boolean isUndoPossible(){
        return picture != null && picture.getSize() > 0;
    }

    /**
     * Gibt an, ob ein Aufruf von {@link #redo()} einen Effekt ergeben wird.
     * @return true, wenn ein Aufruf von <code>redo</code> sinnvoll ist.
     */
    public boolean isRedoPossible(){
        return picture != null && removedFigures.size() > 0;
    }

    public void figureAdded( Figure figure, Picture picture ) {
        if( !urCall ){
            removedFigures.clear();
        }
    }

    public void figureRemoved( Figure figure, Picture picture ) {
        if( !urCall ){
            removedFigures.clear();
        }
    }
}
```

Eine Component um das Bild zu zeichnen

Unser nächstes Ziel ist die Implementierung einer *JComponent*, auf der wir das Bild darstellen können, und welche auch die Factories bei ihren Eingaben unterstützt.

Auf eine *JComponenten* zeichnet man am besten, indem man die Methode *paintComponent* überschreibt. Von dort aus werden wir die *paint*-Methoden sowohl eines *Pictures*, als auch einer *FigureFactory* aufrufen.

Wir werden diese Component, geben wir ihr den Namen *PictureComponent*, so aufbauen, dass man jederzeit das Bild und die benutzte Factory austauschen kann. Wir definieren hier auch gleich, dass eine aktive Factory eine Factory ist, welche der *PictureComponent* übergeben wurde.

Unsere neue *Component* werden wir in einem Package *tutorial.gui* abspeichern, um die logische Trennung zwischen graphisches Oberfläche und Datenstruktur zu verdeutlichen.

Noch einmal ein kurzer Überblick, was die PictureComponent benötigt:

- Eine Methode um der Component zu sagen, welches Bild sie darstellen soll.
  public void setPicture(Picture picture)
- Eine Methode um der Component zu sagen, welche Factory aktiv sein soll. Die Component wird den aktiv-Status der Factories steuern.
  public void setFactory(FigureFactory factory)
- Eine Möglichkeit zu bestimmen, mit welcher Farbe die aktuelle Factory arbeiten soll.
  public void setColor(Color color)
- Dann sollen die Eingaben, welche mit der Maus getätigt werden, an die Factory weitergeleitet

werden. Wir machen dies, indem wir ausnutzen, dass die *FigureFactory MouseListener* und *MouseMotionListener* implementiert hat, wir sie also einfach als entsprechende Listener der Component hinzufügen können.

- Natürlich soll das Bild und die Factory gezeichnet werden, die Methoden ***paintComponent*** muss überschrieben werden.
- Und nicht vergessen dürfen wir, dass die Component neu gezeichnet werden soll, wenn sich Bild oder Factory verändern, also müssen wir den *FactoryListener* und den *PictureListener* implementieren, und den entsprechenden Objekten hinzufügen.

Das war bereits alles, was es zu der *PictureComponent* zu sagen gibt. Versuch dein Glück, und wenn du nicht weiterkommst: eine mögliche Implementierung ist unten beschrieben.

```
package tutorial.gui;

import java.awt.Dimension;
import java.awt.Graphics;

import javax.swing.JComponent;

import tutorial.core.FactoryListener;
import tutorial.core.Figure;
import tutorial.core.FigureFactory;
import tutorial.core.Picture;
import tutorial.core.PictureListener;

/**
 * Diese Component stellt ein Bild dar, und lässt eine Factory
 * eine Darstellung zeichnen.
 */
public class PictureComponent extends JComponent implements PictureListener,
        FactoryListener{

    /** Das Bild, welches gezeichnet werden soll */
    private Picture picture;

    /** Die Factory, mit der neue Figuren erzeugt werden */
    private FigureFactory factory;

    /** Die Farbe, mit der die Factory arbeiten soll */
    private Color color;

    /**
     * Defaultkonstruktor
     */
    public PictureComponent(){
    }

    public void paintComponent( Graphics g ){
        if( picture != null ){
            // es soll nur der Teil vom Bild gezeichnet werden, der auch
            // sichtbar ist.
            g = g.create( 0, 0, picture.getWidth(), picture.getHeight() );

            picture.paint( g );


            if( factory != null )
                factory.paint( g );
```

```
        }
    }

    /**
     * Setzt das Bild, welches gezeichnet werden soll.
     * @param picture Das Bild oder null
     */
    public void setPicture( Picture picture ){
        if( this.picture != null )
            this.picture.removePictureListener( this );

        this.picture = picture;
        if( picture != null ){
            picture.addPictureListener( this );

            // Die bevorzugte Größe dieser Component sollte die Größe des
            // Bildes sein
            setPreferredSize( new Dimension( picture.getWidth(),
                  picture.getHeight() ) );
        }

        if( factory != null ){
            factory.setPicture( picture );
            factory.setActive( picture != null );
        }

        repaint();
    }

    /**
     * Setzt die Factory, welche aktiv ist, und neue Figuren erzeugen soll.
     * @param factory Die Factory oder null
     */
    public void setFactory( FigureFactory factory ){
        if( this.factory != null ){
            // Die alte Factory entfernen
            this.factory.setActive( false );
            this.factory.removeFactoryListener( this );
            ach
            removeMouseListener( this.factory );
            removeMouseMotionListener( this.factory );
        }

        this.factory = factory;

        if( factory != null ){
            // Die neue Factory installieren
            factory.setPicture( picture );
            factory.setActive( picture != null );
            factory.setColor( color );

            factory.addFactoryListener( this );
            addMouseListener( factory );
            addMouseMotionListener( factory );
        }

        repaint();
    }

    public void setColor( Color color ) {
        this.color = color;
        if( factory != null )
            factory.setColor( color );
    }
```

```
    public void figureAdded( Figure figure, Picture picture ) {
        repaint();
    }

    public void figureRemoved( Figure figure, Picture picture ) {
        repaint();
    }

    public void shouldRepaint( FigureFactory factory ) {
        repaint();
    }
}
```

Weitere graphische Elemente

Wir haben nun alle Elemente (zum Teil auf eine sehr abstrakte Weise) implementiert, welche wir benötigen um ein Bild zu zeichnen, und um das Bild herzustellen.

Unser Programm ist aber noch lange nicht fertig, die zweite große Arbeit ist das Schreiben einer Oberfläche, welche dem Benutzer die Wahl des Figuren-Types bietet, ihn die Farbe auswählen lässt, und bereit ist, weitere Steuerungsmöglichkeiten auf einem einfachen Weg einzubinden.

Wir werden das folgendermaßen lösen: wir schreiben ein Frame, auf dem es Toolbars zur Wahl der Factory und der Farbe gibt, außerdem werden wir diesem Frame eine Menübar geben, da man Menüs wunderbar einfach erweitern kann.

In einem ersten Durchgang werden wir nur diejenigen Methoden implementieren, welche wir unbedingt benötigen, um unsere Datenstruktur (das Bild) zu testen. Erst wenn wir sicher sind, dass das Bild funktioniert, werden wir uns dem Rest der graphischen Oberfläche widmen.

Wahl der Factory

Diese Component wird nicht weiter schwer zum implementieren sein.

Sie soll eine Liste der verfügbaren Factories in Form von *JToggleButtons* anbieten.
Der Benutzer kann dann eine Factory wählen, welche der *PictureComponent* übergeben wird.

Wir nennen diese Component *FigureList.*

- Um die *PictureComponent* in der *FigureList* verfügbar zu machen, übergeben wir sie einfach im Konstruktor.
- Um zu bestimmen ob die Buttons der Liste untereinander oder nebeneinander (vertikal oder horizontal) angeordnet sein sollen, übergeben wir dem Konstruktor ein weiteres Argument. So bauen wir etwas Flexibles zusammen, und können später die *FigureList* an verschiedenen Orten positionieren.
- Um die verschiedenen Figuren darzustellen benutzen wir die Methoden *getIcon* und *getName* der *FigureFactory*.

Versuch dein Glück! Eine mögliche Lösung sieht so aus:

```
package tutorial.gui;

import java.awt.GridLayout;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
```

```
import javax.swing.ButtonGroup;
import javax.swing.JComponent;
import javax.swing.JToggleButton;

import tutorial.core.FigureFactory;

/**
 * Zeigt dem Benutzer eine Liste der Figuren, welche gezeichnet werden können
 */
public class FigureList extends JComponent implements ItemListener{
    /** Angabe, dass die Buttons nebeneinander stehen sollen */
      public static final int HORIZONTAL = 1;

      /** Angabe, dass die Buttons untereinander liegen sollen */
      public static final int VERTICAL = 2;

      /** Die Buttons, welche angezeigt werden */
      private JToggleButton[] buttons;

      /** Die Factories, welche gewählt werden können */
      private FigureFactory[] factories;

      /** Die Component, welcher die selektierte Factory übergeben wird */
      private PictureComponent component;

      /**
       * Standardkonstruktor
       * @param factories Die Liste der Factories, welche angezeigt werden
       * sollen
       * @param component Die PictureComponent, der die selektierte Factory
       * übergeben wird
       * @param orientation Die Art, wie die Buttons positioniert werden, einer
       * von {@link #HORIZONTAL} oder {@link #VERTICAL}
       */
      public FigureList( FigureFactory[] factories, PictureComponent component,
                  int orientation ){
            this.component = component;
            this.factories = factories;

            if( orientation == HORIZONTAL )
                  setLayout( new GridLayout( 1, factories.length, 2, 2 ));
            else
                  setLayout( new GridLayout( factories.length, 1, 2, 2 ));


            ButtonGroup group = new ButtonGroup();
            buttons = new JToggleButton[ factories.length ];

            for( int i = 0, n = factories.length; i<n; i++ ){
                  JToggleButton button = new JToggleButton();
                  button.setIcon( factories[i].getIcon() );
                  button.setToolTipText( factories[i].getName() );
                  button.addItemListener( this );
                  add( button );
                  group.add( button );
                  buttons[i] = button;
            }
      }

      /**
       * Sucht die Factory, welche von dem übergebenen Button dargestellt wird.
       * @param button Der Button
       * @return Die Factory, welche von dem Button angezeigt wird
       */
      private FigureFactory factoryFor( JToggleButton button ){
```

```
            for( int i = 0, n = buttons.length; i<n; i++ )
                  if( buttons[i] == button )
                        return factories[i];

            return null;
      }

      public void itemStateChanged( ItemEvent e ){
            JToggleButton button = (JToggleButton)e.getSource();

            if( button.isSelected() ){
                  FigureFactory factory = factoryFor( button );
                  component.setFactory( factory );
            }
      }
}
```

Wahl der Farbe

Wir müssen uns wieder einmal Gedanken machen, wie der Benutzer etwas auswählen soll, diesmal die Farbe.

Wir müssen verschiedene Anforderungen berücksichtigen:

Zum einen soll der Benutzer die Möglichkeit haben, selbst eine neue Farbe herzustellen. Diese neue Farbe sollte dann irgendwo irgendwie gespeichert werden, so dass der Benutzer die Farbe später wieder einmal benutzen kann.

Zum anderen soll dem Benutzer eine Palette von vordefinierten Farben vorgelegt werden, aus denen er ohne großen Aufwand eine bestimmte Farbe wählen kann.

Dann wollen wir mal genauer überlegen, wie diese graphische Oberfläche aussehen soll.

- Die Palette soll aus Knöpfen bestehen, von welchen der Benutzer jeweils einen eindrücken kann (dessen Farbe ist dann gewählt).
- Diese Knöpfe sollten natürlich irgendwie die Farbe anzeigen, welche sie repräsentieren. Wir können dies wohl am einfachsten lösen, indem wir die Knöpfe gleich selbst einfärben.
- Dann brauchen wir noch einen Knopf, der einen Dialog öffnet, auf dem der Benutzer selbst eine Farbe herstellen kann.
- Diese selbst hergestellten Farben speichern wir in der Palette, und zwar werden wir die selektierte Farbe einfach überschreiben.
- Und die Farbe, welche wir in der Palette gewählt haben, weisen wir jeweils dem Dialog-Knopf zu. So sieht der Benutzer gleich, welche Farbe gewählt ist, und wir können beim Öffnen des Dialogs eine sinnvolle Voreinstellung machen.

Die ColorComponent

Du hast es wohl schon bemerkt, hier werden wir zwei verschiedene Typen von Knöpfen brauchen: solche die eingedrückt bleiben (für die Palette), und solche auf die man kurz drückt, und dann geschieht etwas.
Aber diese beiden Knopfarten haben große Gemeinsamkeiten: sie stellen eine Farbe dar, und sie können eingedrückt werden.

Deshalb schreiben wir jetzt eine Grundklasse *ColorComponent* welche diese beiden Eigenschaften besitzt.

Versuch es am besten gleich selbst:

- Die *ColorComponent* soll von *JComponent* erben.
- Es kann eine Farbe gesetzt werden, und wir können die Farbe auch wieder abrufen.
  public void setColor(Color color)
  public Color getColor()
- Es gibt ein Status „gedrückt“ bzw. „nicht gedrückt“, den wir jederzeit ändern können.
  public void setSelected(boolean selected)
  public boolean isSelected()
- Und wir müssen die gesetzte Farbe irgendwie darstellen lassen. Dazu überschreiben wir die Methode *paintComponent* und zeichnen dort die Farbe.

Eine Implementation sieht so aus:

```
package tutorial.gui;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Rectangle;

import javax.swing.BorderFactory;
import javax.swing.JComponent;
import javax.swing.SwingUtilities;
import javax.swing.border.BevelBorder;

/**
 * Zeichnet eine Fläche mit einer ausgewählten Farbe. Die
 * ColorComponent kann selektiert sein, dann wird sie eingedrückt
 * dargestellt, ansonsten erscheint sie erhöht.
 */
public abstract class ColorComponent extends JComponent{
	private Color color;
	private boolean selected;

	/**
	 * Defaultkonstruktor. Die Farbe der Component wird auf rot gesetzt.
	 */
	public ColorComponent(){
		this( Color.RED );
	}

	/**
	 * Standardkonstruktor, die Farbe der Component wird direkt gesetzt.
	 * @param color Die Farbe der Component
	 */
	public ColorComponent( Color color ){
	    this.color = color;
	    setPreferredSize( new Dimension( 40, 40 ) );
	    setMinimumSize( new Dimension( 40, 40 ) );
	    setSelected( false );
	}

	/**
	 * Setzt die Farbe, welche von dieser Component angezeigt werden soll
	 * @param color Die Farbe
	 */
	public void setColor( Color color ){
		this.color = color;
```

```
            repaint();
      }

      /**
       * Gibt die Farbe zurück, welche von dieser Component angezeigt wird.
       * @return Die Farbe
       */
      public Color getColor(){
            return color;
      }

      protected void paintComponent( Graphics g ){
            g.setColor( color );

            // Herausfinden, in welchem Teil der Component gezeichnet werden
            // kann, und diesen
            // Bereich mit Farbe füllen
            Rectangle bounds = SwingUtilities.calculateInnerArea( this, null );
            g.fillRect( bounds.x, bounds.y, bounds.width, bounds.height );
      }

      /**
       * Setzt den Selektions-Status dieser ColorComponent. Wenn die Component
       * selektiert ist, wird
       * ein eingedrückter Rand gezeichnet, ansonsten wird ein erhöhter Rand
       * benutzt.
       * @param selected true, wenn die Component selektiert ist, ansonsten
       * false
       */
      protected void setSelected( boolean selected ){
            this.selected = selected;
            if( selected )
                  setBorder(
                        BorderFactory.createBevelBorder( BevelBorder.LOWERED ));
            else
                  setBorder(
                        BorderFactory.createBevelBorder( BevelBorder.RAISED ));

            repaint();
      }

      /**
       * Gibt an, ob die Component als selektiert gilt
       * @return Den Selektions-Status
       */
      protected boolean isSelected(){
            return selected;
      }
}
```

Der ColorToggle

Als nächster kommt der Typ Knopf an die Reihe, welchen wir in der Palette benutzen werden.

Wir werden in *ColorToggle* nennen (Toggle bedeutet Schalter), und er soll eine Erweiterung der *ColorComponent* sein.

Ein graphisches Steuerelement, ganz egal was ist, sollte eigentlich nie „wissen“, was es alles auslöst. Die Reaktion das Runterdrücken hat den *ColorToggle* selbst nicht zu interessieren, das wird an einer anderen Stelle verarbeitet.
Wie müssen aber eine einfache Möglichkeit einbauen, andere Objekte zu benachrichtigen, falls jemand den Knopf drückt.

Wir machen dies mit Hilfe eines Listeners: Jedes Objekt kann einen Listener bei dem *ColorToggle* registrieren, und wenn der Knopf gedrückt wurde, werden die Listener benachrichtigt.
Als Listener verwenden wir den *ChangeListener*, er passt zu diesem Problem, da hier ja etwas verändert wird.

Fassen wir zusammen:

- *ColorToggle* erbt von *ColorComponent*.
- ColorToggle kann mit der Maus gedrückt werden. Der Knopf bleibt dann eingedrückt. (Es ist nicht notwendig, aber es wäre natürlich schön, wenn auch der umgekehrte Weg möglich wäre, also wenn man einen eingedrückten Knopf wieder herausziehen könnte).
- Wann immer der gedrückt/nicht-gedrückt -Status verändert wird, werden ChangeListener benachrichtigt, welche sich früher registriert haben.
  Um die Listener zu registrieren benötigen wir zwei Methoden:
  public void addChangeListener(ChangeListener listener)
  public void removeChangeListener(ChangeListener listener)

Versuch dein Glück!
Ich habe eine Version geschrieben, die auch dafür sorgt, dass man das Eindrücken abbrechen kann, indem man mit der Maus vom Knopf heruntergeht:

```
package tutorial.gui;

import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.util.Vector;

import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

/*
 * Diese Component zeigt eine Farbe an. Sie kann entweder selektiert sein, oder
 * halt nicht.
 */
public class ColorToggle extends ColorComponent{
      /** Die Liste der Listener, welche diese Component abhorchen */
      private Vector<ChangeListener> listeners = new Vector<ChangeListener>();

      /** Angabe, ob derzeit die Maus gedrückt wird */
      private boolean pressed = false;

      /**
       * Angabe, wie der Selektions-Status der Component war, bevor die Maus
       * gedrückt wurde
       */
      private boolean wasSelected = false;

      /**
       * Defaultkonstruktor
       */
      public ColorToggle(){
          // Einen MouseListener registrieren. Dieser Listener wird den
          // ColorToggle steuern.
            addMouseListener( new MouseListener(){
                  public void mousePressed( MouseEvent e ){
                        if( !pressed ){
                              pressed = true;
```

```
                        wasSelected = isSelected();
                        setSelectedNoEvent( !wasSelected );
                  }
            }
            public void mouseReleased( MouseEvent e ){
                  if( pressed ){
                        pressed = false;

                        int x = e.getX();
                        int y = e.getY();

                        if( x >= 0 && y >= 0 && x <= getWidth() &&
                              y <= getHeight ()){

                              setSelected( !wasSelected );
                        }
                  }
            }
            public void mouseClicked( MouseEvent e ){
            }
            public void mouseEntered( MouseEvent e ){
                  if( pressed ){
                        setSelectedNoEvent( !wasSelected );
                  }
            }
            public void mouseExited( MouseEvent e ){
                  if( pressed ){
                        setSelectedNoEvent( wasSelected );
                  }
            }
      });
}

/**
 * Verändert den Selektions-Status <i>ohne</i> ein Event auszulösen.
 * @param selected Der neue Selektions-Status
 */
protected void setSelectedNoEvent( boolean selected ){
      super.setSelected( selected );
}

public void setSelected( boolean selected ){
      super.setSelected( selected );
      fireStateChanged();
}

/**
 * Verschickt an alle registrierten ChangeListener ein Event, dieser
 * Toggle habe sich verändert.
 */
protected void fireStateChanged(){
      ChangeEvent e = new ChangeEvent( this );

      if( listeners != null ){
          for( ChangeListener c : listeners )
              c.stateChanged( e );
      }
}

public void addChangeListener( ChangeListener listener ){
      listeners.add( listener );
}

public void removeChangeListener( ChangeListener listener ){
      listeners.remove( listener );
```

```
        }
}
```

Der ColorButton

Wenden wir uns dem zweiten Knopf zu.

Dieser Knopf soll vom Benutzer kurz gedrückt werden, dann soll ein Dialog erscheinen, auf dem er eine Farbe selbst herstellen kann.

Glücklicherweise müssen wir diesen Dialog nicht selbst schreiben, der *JColorChooser* im Package *javax.swing* wird uns da gute Dienste erweisen. Diese Klasse hat nämlich eine statische Methode *showDialog*, welche eine Farbe zurück liefert (es sei denn, der Benutzer drückt das Abbrechen-Knöpfchen).

Wie schon beim *ColorToggle* werden wir auch hier einen Mechanismus einbauen um andere Objekte mit Hilfe eines Listeners von Veränderungen zu unterrichten.
Allerdings ist ein *ChangeListener* hier nicht ganz passend, der *ColorButton* sollte ein bisschen mehr Informationen übertragen können.
Deshalb werden wir hier *ItemListener*. Da ein *ItemListener* ein *ItemEvent* erwartet, und das *ItemEvent* eine Referenz zu einem *ItemSelectable* benötigt, müssen wir den *ColorButton* das Interface *ItemSelectable* implementieren lassen.

Unsere Aufgabe lautet also:

- Der *ColorButton* wird von der *ColorComponent* erben
- Man kann dem *ColorButton ItemListener* hinzufügen und entfernen. Um dies zu ermöglichen, muss *ColorButton* das Interface *ItemSelectable* implementieren.
  public void addItemListener(ItemListener listener)
  public void removeItemListener(ItemListener listener)
- Wird auf den Knopf gedrückt, erscheint ein *JColorChooser*, welchen wir über die Methode *showDialog* aufrufen.

So, schreib was zusammen!

Mein Vorschlag sieht folgendermaßen aus:

```
package tutorial.gui;

import java.awt.AWTEvent;
import java.awt.Color;
import java.awt.ItemSelectable;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.util.Vector;

import javax.swing.JColorChooser;

/*
 * Diese ColorComponent verhält sich wie ein Button. Man kann mit der Maus auf
 * die Component klicken, und es erscheint ein JColorChooser mit dem eine
 * neue Farbe gewählt werden kann.<br>
 * Man kann dem Button ItemListener hinzufügen, welche angesprochen werden,
 * falls
 * eine neue Farbe ausgewählt wird.
```

```
 */
public class ColorButton extends ColorComponent implements ItemSelectable{
	/** Für das ItemEvent brauchen wir eine neue ID */
	public static final int COLOR_BUTTON_EVENT_ID =
		AWTEvent.RESERVED_ID_MAX + 1;

	/** Eine Liste aller ItemListener, welche angemeldet wurden */
	private Vector<ItemListener> listeners = new Vector<ItemListener>();

	/** Diese Variable gibt an, ob derzeit die Maus gedrückt wird */
	private boolean pressed = false;

	/**
	 * Defaultkonstruktor
	 */
	public ColorButton(){
		// Damit auch KeyEvents verarbeitet werden können, muss die
		// Component fokusierbar sein
		setFocusable( true );

		addMouseListener( new MouseListener(){
			public void mousePressed( MouseEvent e ){
				setSelected( true );
				pressed = true;
			}
			public void mouseClicked( MouseEvent e ){
			}
			public void mouseReleased( MouseEvent e ){
				if( pressed ){
					pressed = false;

					int x = e.getX();
					int y = e.getY();

					if( x >= 0 && y >= 0 && x <= getWidth() &&
						y <= getHeight() ){

						doClick();
					}
				}
			}
			public void mouseExited( MouseEvent e ){
				if( pressed )
					setSelected( false );
			}
			public void mouseEntered( MouseEvent e ){
				if( pressed )
					setSelected( true );
			}
		});

		addKeyListener( new KeyAdapter(){
			public void keyPressed( KeyEvent e ){
				if( e.getKeyCode() == KeyEvent.VK_ENTER )
					doClick();
			}
		});
	}

	/**
	 * Lässt einen JColorChooser erscheinen, mit dem eine neue Farbe gewählt
	 * werden kann.<br>
	 * Sollte eine neue Farbe gewählt werden, so wird ein ItemEvent
	 * abgeschickt.
	 */
```

```
	public void doClick(){
		setSelected( true );

		Color color = JColorChooser.showDialog( this, "Farbe", getColor() );

		if( color != null ){
			setColor( color );
		}

		setSelected( false );
	}

	/**
	 * Setzt die Farbe, welche von diesem Button angezeigt werden soll.
	 * Die Listener werden benachrichtigt.
	 * @param color Die Farbe
	 */
	public void setColor( Color color ){
		super.setColor( color );
		fireColorChanged();
	}

	/**
	 * Diese Methode wird von dem Interface {@link ItemSelectable}
	 * vorgeschrieben.
	 * @return Ein Array der Länge 1, welcher ein Color-Object enthält.
	 */
	public Object[] getSelectedObjects(){
		return new Color[]{ getColor() };
	}

	/**
	 * Unterrichtet alle registrierten Listener, dass die Farbe dieses Knopfes
	 * verändert wurde.
	 */
	protected void fireColorChanged(){
		ItemEvent e = new ItemEvent( this, COLOR_BUTTON_EVENT_ID,
			getColor(), ItemEvent.ITEM_STATE_CHANGED );

		for( ItemListener i : listeners )
			i.itemStateChanged( e );
	}

	/**
	 * Registriert einen ItemListener, welcher angesprochen wird, sollte
	 * sich die Farbe dieses Buttons ändern.
	 * @param listener Der neue ItemListener
	 */
	public void addItemListener( ItemListener listener ){
		listeners.add( listener );
	}

	/**
	 * Entfernt einen bestimmten ItemListener.
	 * @param listener Der zu entfernende Listener
	 */
	public void removeItemListener( ItemListener listener ){
		listeners.remove( listener );
	}
}
```

Die ColorList

Nun müssen wir den *ColorButton* und mehrere *ColorToggle* kombinieren.
Dazu gibt es, wieder einmal, eine neue Klasse, diesmal die *ColorList*.

Die *ColorList* ist eine *Component*, auf der ein *ColorButton* und mehrere *ColorToggles* sitzen.

Die Liste wird die ganze Verwaltung der Farben übernehmen (z.B. was passiert, wenn auf ein *ColorToggle* gedrückt wird), und einer *PictureComponent* jeweils die Farbe setzten, welche der Benutzer ausgewählt hat.

Diese direkte Übergabe der Farbe an die PictureComponent ist eigentlich nicht ganz sauberes Design, für grössere Programme würde man noch eine Zwischenklasse einbauen, die das erledigt. Aber wir schreiben hier ein kleines Programm, und bei kleinen Programmen kann man ab und zu mal einen Trick anwenden, um etwas Code zu sparen.

Also was benötigen wir hier alles?

- Die *ColorList* werden wir von *JComponent* erben lassen, da diese Liste ja irgendwie dargestellt werden muss.
- Die Liste soll so hergestellt werden, dass wir mit wenigen Änderungen eine andere Anzahl ColorToggles setzen können, und dass wir bestimmen können, ob die Liste die Knöpfe nebeneinander oder untereinander anreihen soll.
- Der Benutzer soll eine Palette von Farben vorgesetzt bekommen. Wir werden eine Methode schreiben, welche verschiedene Farben generiert, und diese Farben den *ColorToggles* setzen.
- Wir müssen die Interfaces *ChangeListener* und *ItemListener* implementieren, und den *ColorToggles*, bzw. dem *ColorButton* hinzufügen. Und dann müssen wir eine Logik schreiben, welche auf die Ereignisse reagiert.

Das war schon alles, bitte, benutze die Tastatur und versuch das mal zu schreiben.

Wie immer gibt es auch diesmal einen Beispielcode:

```
package tutorial.gui;

import java.awt.Color;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.GridLayout;
import java.awt.Insets;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;

import javax.swing.JComponent;
import javax.swing.JPanel;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

/**
 * Mit dieser Component kann der Benutzer eine Farbe auswählen, und auch
 * neue Farben herstellen.
 */
public class ColorList extends JComponent implements ItemListener,
      ChangeListener{
      /** Angabe, dass die Knöpfe horizontal aufgelistet werden sollen */
      public static final int HORIZONTAL = 1;

      /** Angabe, dass die Knöpfe vertikal aufgelistet werden sollen */
      public static final int VERTICAL = 2;

      /** Der Knopf, der derzeit eingedrückt ist */
      private ColorToggle selected;
```

```
	/** Der Knopf zum Herstellen neuer Farben */
	private ColorButton button;

	/** Die Component, der die selektierten Farben übergeben werden */
	private PictureComponent component;

	/**
	 * Eine Hilfsvariable die auf true gesetzt wird, wenn nicht auf
	 * Events reagiert werden soll
	 */
	private boolean invoking = false;

	/**
	 * Standardkonstruktor
	 * @param toggles Die Anzahl ColorToggles, welche auf der Liste sein
	 * sollen.
	 * @param orientation Angabe, ob die Knöpfe untereinander oder
	 * nebeneinander dargestellt werden sollen.<br>
	 * Einer von {@link #HORIZONTAL HORIZONTAL} oder
	 * {@link #VERTICAL VERTICAL}
	 * @param component Die Component, welcher die selektierte Farbe übergeben
	 * werden soll.
	 */
	public ColorList( int toggles, int orientation,
			PictureComponent component ){

		this.component = component;

		button = new ColorButton();
		button.addItemListener( this );

		JPanel panel = createColorToggles( toggles, orientation );

		setLayout( new GridBagLayout() );

		if( orientation == HORIZONTAL ){
			add( button, new GridBagConstraints( 0, 0, 1, 1, 10.0, 10.0,
				GridBagConstraints.CENTER, GridBagConstraints.BOTH,
				new Insets( 0, 0, 0, 0 ), 0, 0 ));

			add( panel, new GridBagConstraints( 1, 0, 1, 1, 100.0, 100.0,
				GridBagConstraints.CENTER, GridBagConstraints.BOTH,
				new Insets( 0, 0, 0, 0 ), 0, 0 ));
		}
		else{
			add( button, new GridBagConstraints( 0, 0, 1, 1, 10.0, 10.0,
				GridBagConstraints.CENTER, GridBagConstraints.BOTH,
				new Insets( 0, 0, 0, 0 ), 0, 0 ));

			add( panel, new GridBagConstraints( 0, 1, 1, 1, 100.0, 100.0,
				GridBagConstraints.CENTER, GridBagConstraints.BOTH,
				new Insets( 0, 0, 0, 0 ), 0, 0 ));
		}
	}

	/**
	 * Generiert ein Panel, auf dem mehrere ColorToggle's sitzen. Den
	 * Toggles wurde diese ColorList als ChangeListener angemeldet.
	 * @param toggles Die Anzahl Knöpfe, welche generiert werden sollen.
	 * @param orientation Angabe, ob die Knöpfe untereinander oder
	 * nebeneinander dargestellt werden sollen.
	 * Einer von {@link #HORIZONTAL HORIZONTAL} oder
	 * {@link #VERTICAL VERTICAL}
	 * @return Das Panel mit den neuen Knöpfen.
	 */
```

```
private JPanel createColorToggles( int toggles, int orientation ){
	JPanel panel = new JPanel();

	if( orientation == HORIZONTAL )
		panel.setLayout( new GridLayout( 2, (toggles+1) / 2 ) );
	else
		panel.setLayout( new GridLayout( (toggles+1) / 2, 2 ) );

	for( int i = 0; i < toggles; i++ ){
		ColorToggle toggle = new ColorToggle();
		toggle.setColor( createColor( i, toggles ));
		toggle.addChangeListener( this );
		panel.add( toggle );

		if( i == 0 )
			toggle.setSelected( true );
	}

	return panel;
}

/**
 * Berechnet eine Farbe aus einem Regenbogen.
 * @param index Der Index dieses Farbstücks.
 * @param length Die Anzahl der Farben, die überhaupt vorkommen sollen.
 * @return Die generierte Farbe.
 */
protected Color createColor( int index, int length ){
	double real = 6.0 * index / length;

	// RGB = 100
	if( real <= 1 )
		return new Color( 255, (int)(255 * real), 0 );

	// RGB = 110
	if( --real <= 1 )
		return new Color( (int)(255 * (1-real)), 255, 0 );

	// RGB = 010
	if( --real <= 1 )
		return new Color( 0, 255, (int)(255 * real) );

	// RGB = 011
	if( --real <= 1 )
		return new Color( 0, (int)( 255 * (1-real)), 255 );


	// RGB = 001
	if( --real <= 1 )
		return new Color( (int)(255 * real), 0, 255 );

	// RGB = 101
	if( --real <= 1 )
		return new Color( 255, 0, (int)(255 * (1-real)) );

	return Color.BLACK;
}

/**
 * Wird normalerweise von dem ColorButton aufgerufen, welcher von dieser
 * Liste angezeigt wird.<br>
 * Die Methode sorgt dafür, dass die PictureComponent, welche im
 * Konstruktor übergeben wurde, die richtige Farbe gesetzt hat.
 * @param e Das Event, das die Veränderung beschreibt.
 */
```

```
	public void itemStateChanged( ItemEvent e ){
		if( selected != null )
			selected.setColor( button.getColor() );

		component.setColor( button.getColor() );
	}

	/**
	 * Wird von den ColorToggles aufgerufen, welche von dieser Liste
	 * angezeigt werden.<br>
	 * Die Methode sogt dafür, dass der angezeigte ColorButton die korrekte
	 * Farbe anzeigt.
	 * @param e Das Event, welches angibt, welcher ColorToggle gedrückt
	 * wurde.
	 */
	public void stateChanged( ChangeEvent e ){
		if( !invoking ){
			invoking = true;

			ColorToggle toggle = (ColorToggle)e.getSource();

			if( selected != null )
				selected.setSelected( false );

			if( toggle.isSelected() ){
				selected = toggle;
				button.setColor( toggle.getColor() );
			}
			else
				selected = null;

			invoking = false;
		}
	}
}
```

Das Hauptmenü

Wir haben nun schon die Hälfte der Arbeit hinter uns. Zeit also, dass wir endlich mal was zu Gesicht bekommen!

Wir werden nun das Menü des Hauptfensters schreiben, aber wir werden es nur soweit ausprogrammieren, dass wir ein neues Bild herstellen, und unsere Undo/Redo-Funktion testen können.

Bisher haben wir nur Klassen geschrieben, welche ein Teilproblem lösten (wie wählt der Benutzer eine Farbe, etc...). Doch mit dem Hauptmenü und dem Hauptfenster kommen wir zu zwei Klassen, welche viel mehr mit der Gesamtorganisation des Programmes zu tun haben.

Es muss nun eine wichtige Entscheidung getroffen werden: wer verwaltet das Bild?
Das Hautpmenü, von welchem aus die Herstellung eines Bildes geschehen wird, oder das Hauptfenster, auf dem alle Komponenten des Programmes angezeigt werden?
Wir wählen das Hauptfenster. Das Hauptfenster wird sozusagen das Zentrum unseres Programmes, hier werden alle Fäden zusammenlaufen.
Für das Hauptmenü bleibt genug, es muss all die Aktionen weiterleiten, welche der Benutzer über die Menüeinträge aufruft.
(Es bleibt anzumerken, dass man Programme auch dezentral aufbauen kann. So könnte es zu jedem Menüeintrag eine eigene Klasse geben, welche eine Methode besitzt, die per Listener aufgerufen wird, sollte der Menüeintrag gewählt werden. Dieses Programm würde durchaus ein bisschen mehr Code haben, andererseits würde es unser kleines Zeichenprogramm an Dynamik weit übertreffen.)

Machen wir eine Übersicht, was wir nun in dieser ersten Fassung alles implementieren sollen.

- Das Hauptmenü soll von *JMenuBar* erben. So können wir die Menüeinträge sehr einfach einbauen.
- Wir müssen eine Methode schreiben, welche das Hauptmenü von dem aktuell gewählten Bild unterrichtet. Diese Methode wird dann vom Hauptfenster aufgerufen.
  public void setPicture(Picture picture)
- Eine Methode, welche neue Bilder herstellt, und dem Hauptfenster übergibt, muss implementiert werden. Vorerst werden wir hier Bilder mit einer fixen Größe herstellen, später werden wir dann einen kleinen Dialog schreiben, auf dem der Benutzer die Größe selbst wählen kann.
  public void newPicture()
- Das Hauptmenü wird eine Instanz des *URManagers* besitzen. Es wird kein großes Problem sein die Methoden *undo* und *redo* aufzurufen.
  Wir müssen aber darauf achten, dass der *URManager* immer auch das aktuelle Bild abhören kann, die Methode *setPicture* muss dem entsprechend aufgebaut sein.
- Und natürlich benötigen wir ein paar Menüeinträge. Dazu werden wir *JMenuItem*s verwenden, welche in *JMenu*s zusammengefasst werden.
  Um genau zu sein: Ein Menu „Datei“ mit „Neues Bild“ und ein Menu „Edit“ mit „Rückgängig“ und „Wiederholen“ müssen sichtbar sein.
  Die *MenuItems* werden wir mit einem ActionListener abhören.
- Da der Benutzer nicht unendlich oft „Rückgängig“ aufrufen kann, soll das entsprechende MenuItem ein- und ausgeschaltet werden (mit Hilfe von *setEnabled*). Das selbe gilt auch für das „Wiederholen“-Item. Um dies zu erreichen werden wir einen *PictureListener* implementieren, denn nur wenn dem Bild etwas hinzugefügt oder entfernt wurde, wird sich die Möglichkeit einer undo- oder redo-Aktion verändern.

Versuche solch ein Menu zu schreiben. Ach ja: einen Namen müssen wir noch wählen: *MainMenu*.

```
package tutorial.gui;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;

import tutorial.core.Figure;
import tutorial.core.Picture;
```

```
import tutorial.core.PictureListener;
import tutorial.core.URManager;

/**
 * Das Hauptmenü des Programms. Es sorgt dafür, dass alle Aufrufe eines
 * Menüeintrags an den richtigen Ort geleitet werden.
 */
public class MainMenu extends JMenuBar implements ActionListener,
      PictureListener{

      /** Das Hauptfenster dieses Programme */
      private MainFrame frame;

      /** Ein URManager um das Undo/Redo zu realisieren */
      private URManager urManager;

      /** Die MenuItems, welche ein- und ausgeschaltet werden müssen */
      private JMenuItem itemUndo, itemRedo;

      /** Das Bild, welches bearbeitet wird */
      private Picture picture;

      /**
       * Standardkonstruktor
       * @param frame Das Hauptfenster des Programmes
       */
      public MainMenu( MainFrame frame ){
            this.frame = frame;

            JMenu file = new JMenu( "Datei" );
            JMenu edit = new JMenu( "Edit" );

            JMenuItem newPicture = new JMenuItem( "neues Bild" );
            newPicture.setActionCommand( "new" );
            newPicture.addActionListener( this );

            itemUndo = new JMenuItem( "Rückgängig" );
            itemUndo.setActionCommand( "undo" );
            itemUndo.addActionListener( this );

            itemRedo = new JMenuItem( "Wiederholen" );
            itemRedo.setActionCommand( "redo" );
            itemRedo.addActionListener( this );

            file.add( newPicture );
            edit.add( itemUndo );
            edit.add( itemRedo );

            add( file );
            add( edit );

            urManager = new URManager();
            updateURItems();
      }

      /**
       * Stellt ein neues Bild her, und sorgt dafür, dass das Programm
       * dieses neue Bild auch benutzt.
       */
      public void newPicture(){
            frame.setPicture( new Picture( 500, 500 ) );
      }

      /**
       * Setzt das Bild, welches vom Programm bearbeitet wird. Diese Methode
```

```
         * wird normalerweise durch das Hauptfenster aufgerufen.
         * @param picture Das Bild
         */
        public void setPicture( Picture picture ){
            if( this.picture != null ){
                this.picture.removePictureListener( this );
            }

            this.picture = picture;
            urManager.setPicture( picture );

            if( picture != null )
                picture.addPictureListener( this );

            updateURItems();
        }

        public void actionPerformed( ActionEvent e ){
            String command = e.getActionCommand();

            if( command.equals( "new" ))
                newPicture();

            else if( command.equals( "undo" ))
                urManager.undo();

            else if( command.equals( "redo" ))
                urManager.redo();
        }

        /**
         * Sichert ab, dass die Menüeinträge für Undo/Redo wählbar sind, falls
         * die Aktion möglich ist, oder dass sie nicht wählbar sind, wenn
         * die Aktion nicht möglich ist.
         */
        public void updateURItems(){
            itemUndo.setEnabled( urManager.isUndoPossible() );
            itemRedo.setEnabled( urManager.isRedoPossible() );
        }

        public void figureAdded( Figure figure, Picture picture ) {
            updateURItems();
        }

        public void figureRemoved( Figure figure, Picture picture ) {
            updateURItems();
        }
}
```

Das Hauptfenster

Wir müssen uns nicht mehr viel Gedanken machen, wie das Hauptfenster aufgebaut werden soll, es muss einfach alles was wir bisher geschrieben haben beinhalten.

Also können wir direkt zur Liste der benötigten Features kommen.

- Das Hauptfenster, wir nennen es *MainFrame*, wird von *JFrame* erben.
- Zuoberst setzen wir das Hauptmenü hin (mit Hilfe von *setJMenuBar*).
- Darunter kommt die Liste der Figuren, an der linken Seite die Palette der Farben, und in der Mitte wird die *PictureComponent* liegen. Die *PictureComponent* werden wir in ein *JScrollPane* stecken, damit wir keine Probleme bekommen, sollte das Bild einmal grösser als der Bildschirm

sein.

- Die *FigureList* benötigt einen Array von Factories, welche gewählt werden können. Wir werden die Erzeugung dieses Arrays jemand anderem überlassen, und selbst einen Array von Factories im Konstruktor des *MainFrames* erwarten.
- Schon früher haben wir gesagt, dass das Hauptfenster das Bild verwalten wird, also benötigen wir Methoden um das Bild zu setzen.
  public void setPicture(Picture picture)
  public Picture getPicture()
- Irgendwie sollte das Programm auch wieder beendet werden können... also benötigen wir eine Methode die *System.exit* aufruft. Dazu werden wir einen *WindowListener* implementieren, welcher bei einem Aufruf von *windowClosing* eine Methode *exit* aufruft.
  Ganz bewusst rufen wir nicht direkt *System.exit* auf, wir wollen uns die Möglichkeit offen halten später vor dem Beenden einen Dialog zu öffnen, und den Benutzer noch einmal zu fragen, ob er das Programm wirklich beenden will (diese Methode werden wir dann auch aus dem Hauptmenü aufrufen können, im Gegensatz zum WindowListener).
  public void exit()

Versuch dein Glück, so könnte dein Code aussehen:

```
package tutorial.gui;

import java.awt.Container;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import javax.swing.JFrame;
import javax.swing.JScrollPane;

import tutorial.core.FigureFactory;
import tutorial.core.Picture;

/**
 * Das Hauptfenster ist das eigentliche Zentrum des Programms. Es sorgt
 * dafür, dass alle graphischen Elemente angezeigt werden, und dass überall
 * das zu bearbeitende Bild bekannt ist.
 */
public class MainFrame extends JFrame{
      /** Die Palette der Farben */
      private ColorList colorList;

      /** Die List aller Figuren, welche der Benutzer auswählen kann */
      private FigureList figureList;

      /** Das Hauptmenü */
      private MainMenu mainMenu;

      /** Die Component auf der das Bild gezeichnet wird */
      private PictureComponent pictureComponent;

      /** Die JScrollPane auf die die pictureComponent gesetzt ist */
      private JScrollPane pictureScrollPane;
```

```
/** Das angezeigte Bild */
private Picture picture;

/**
 * Standardkonstruktor
 * @param factories Die verschiedenen Figuren, welche gewählt werden
 * können.
 */
public MainFrame( FigureFactory[] factories ){
      setTitle( "Tutorial" );
      setDefaultCloseOperation( DO_NOTHING_ON_CLOSE );

      pictureComponent = new PictureComponent();
      pictureScrollPane = new JScrollPane( pictureComponent );
      colorList = new ColorList( 20,
            ColorList.VERTICAL, pictureComponent );
      figureList = new FigureList( factories, pictureComponent,
            FigureList.HORIZONTAL );

      Container content = getContentPane();

      content.setLayout( new GridBagLayout() );

      content.add( figureList, new GridBagConstraints( 0, 0, 2, 1,
            1.0, 1.0, GridBagConstraints.NORTHWEST,
            GridBagConstraints.NONE, new Insets( 2, 2, 2, 2 ), 0, 0 ));

      content.add( colorList, new GridBagConstraints( 0, 1, 1, 1,
            1.0, 1.0, GridBagConstraints.NORTHWEST,
            GridBagConstraints.NONE, new Insets( 2, 2, 2, 2 ), 0, 0 ));

      content.add( pictureScrollPane, new GridBagConstraints( 1, 1, 1, 1,
            100.0, 100.0, GridBagConstraints.CENTER,
            GridBagConstraints.BOTH, new Insets( 2, 2, 2, 2 ), 0, 0 ));

      mainMenu = new MainMenu( this );
      setJMenuBar( mainMenu );

      setSize( 700, 500 );
      setLocationRelativeTo( null );

      addWindowListener( new WindowAdapter(){
            public void windowClosing( WindowEvent e ) {
                exit();
            }
      });
}

/**
 * Diese Methode beendet das Programm.
 */
public void exit(){
    System.exit( 0 );
}

/**
 * Setzt das Bild, welches dargestellt und bearbeitet werden soll.
 * @param picture Das Bild
 */
public void setPicture( Picture picture ){
      this.picture = picture;
      pictureComponent.setPicture( picture );
      pictureScrollPane.validate();
      mainMenu.setPicture( picture );
}
```

```
	/**
	 * Gibt das Bild zurück, welches dargestellt wird.
	 * @return Das Bild oder null
	 */
	public Picture getPicture(){
		return picture;
	}
}
```

public static void main

Es ist nun an der Zeit, unser Programm zu starten, und einen ersten Eindruck zu erhalten.

Wir werden aber zuerst eine eigene Klasse nur für die Startmethode schreiben. Da wir nur eine Startmethode implementieren werden, besteht so nie die Gefahr, einmal etwas falsches aufzurufen.

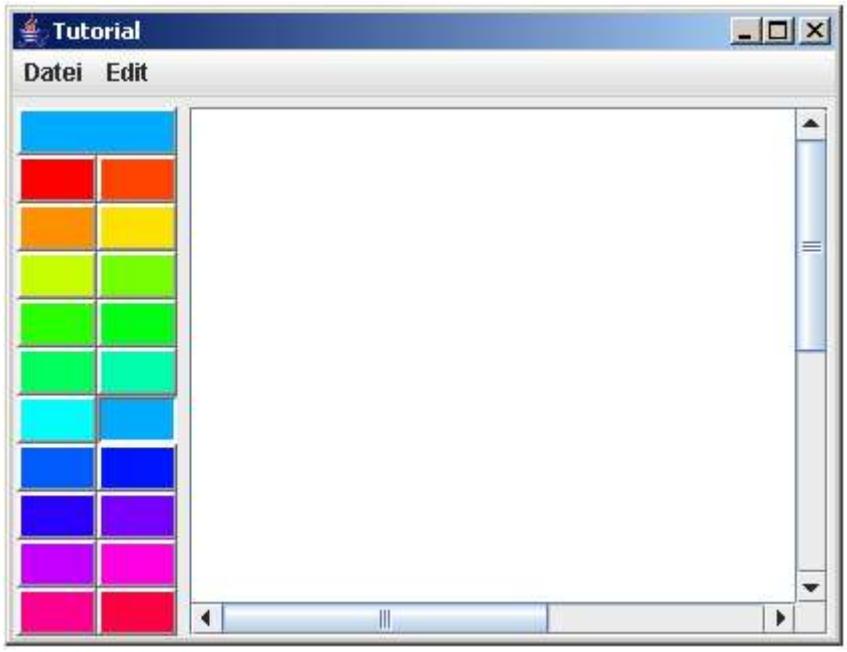
Diese Klasse *Main*, welche in das Package *tutorial* gehört, hat nur wenige Eigenschaften:

- Wir benötigen eine Methode, welche das Programm startet. Diese Methode stellt ein neues *MainFrame* her, und macht es sichtbar.
  public static void main(String[] args)
- Außerdem bereiten wir eine Methode vor, welche uns eine Liste der verfügbaren Factories generieren wird.
  public static FigureFactory[] listFactories()

Um das zu schreiben, brauchst du wohl keine Hilfe.

```
package tutorial;

import tutorial.core.FigureFactory;
import tutorial.gui.MainFrame;

public class Main{
	public static void main( String[] args ){
		MainFrame frame = new MainFrame( listFactories() );
		frame.setVisible( true );
	}

	public static FigureFactory[] listFactories(){
		return new FigureFactory[0];
	}
}
```

Und nun: **Start**



Sieht doch schon ganz gut aus, nicht?

Nur das Zeichnen von Figuren ist noch ein bisschen schwer, weil's so wenige gibt.

Roar wird Dir nun zeigen, wie Du ein paar Figuren implementieren kannst. Wir sehen uns dann in ein paar Seiten wieder.

Die Figuren

Oben haben wir die Interfaces *Figure* und *FigureFactory* erstellt. Nun wollen wir eigene Figuren implementieren, die gezeichnet werden können. Wir wollen mit einem normalen Rechteck anfangen, welches nicht ausgefüllt wird. Dazu müssen wir zwei Klassen schreiben, die *Figure* bzw. *FigureFactory* implementieren. Schauen wir uns nochmal das Interface *FigureFactory* an. Da müssen wir einige Methoden implementieren und dazu noch die Methoden aus den Interfaces *MouseListener* und *MouseMotionListener*. Das sind ganz schön viele. Wenn wir mehrere Figuren schreiben möchten, dann wird das viel Schreibarbeit. Und da einige Methoden, wie zum Beispiel *addFactoryListener(FactoryListener listener)* in jeder Factory gleich funktionieren, wäre es ziemlich viel Arbeit den gleichen Code immer wieder zu kopieren und einzufügen. Außerdem ist das langweilig und bläht den Sourcecode der Factories nur auf und würde auch zu einer Verschlechterung der Wiederverwendbarkeit des Codes führen. Deshalb werden wir eine abstrakte Klasse erstellen.

Falls Dir die Wörter „abstrakte Klasse“ nichts sagen, lies es im Anhang nach!

In unsere abstrakte Klasse, wir nennen sie *AbstractFigureFactory* werden wir einige Methoden aus *FigureFactory* und alle Methoden aus *MouseListener* und *MouseMotionListener* implementieren. Die Implementierung der Methoden aus den beiden Mouse Interfaces gibt den Programmierern der FigureFactories die Freiheit nur die Methoden zu implementieren, die sie auch wirklich benötigen. Zusätzlich werden wir in der abstrakten Klasse 3 Variablen deklarieren:

- Einen Platzhalter für die nächste zu instantiierende und zu zeichnende Figur.
  AbstractFigure next

- Eine boolesche Variable die anzeigt, ob die aktuelle Figur aktiv ist (wird über *setActive(boolean active)* gesetzt).
  boolean active
- Eine boolesche Variable die anzeigt ob die Maus gerade gedrückt wird.
  boolean pressed

Außerdem werden wir 2 Methoden zum Laden und Speichern des Bildes implementieren, die später weiter erläutert werden. Neben der *AbstractFigureFactory* werden wir noch eine zweite abstrakte Klasse schreiben, die *AbstractFigure*. Von *AbstractFigure* werden wir unsere Figuren ableiten, die unsere FigureFactories erzeugen. *AbstractFigure* implementiert außer den zwei setter und getter Methoden aus *Figure* noch einige nützliche Methoden, die die Position von der Figur darstellen.

AbstractFigureFactory

```
/**
 * Abstrakte Klasse, die die Standard-Methoden von <code>FigureFactory</code>
 * und die von <code>MouseListener</code> bereits implementiert.
 */
public abstract class AbstractFigureFactory implements FigureFactory {

	protected Vector<FactoryListener> listeners =
	                                        new Vector<FactoryListener>();

	/**
	 * Platzhalter für die <code>Figure</code>, die als nächstes
	 * initialisiert, zum <code>Picture</code> hinzugefügt und gezeichnet
	 * werden soll.
	 */
	protected AbstractFigure next;

	protected Color color;
	protected Picture picture;
	protected boolean active = false;
	protected boolean pressed = false;

	/**
	 * Diese Methode schreibt eine Figure in einen DataOutputStream.
	 * Sie castet die übergebene Figure in eine AbstractFigure. Wenn eine
	 * FigureFactory und ihre dazugehörige Figur nicht die Klasse
	 * AbstractFigure benutzt, muss diese Methode unbedingt überschrieben
	 * werden.
	 */
	public void write( Figure figure, DataOutputStream out ) throws
	                                                        IOException {
		AbstractFigure af = (AbstractFigure) figure;
		out.writeInt(af.getX());
		out.writeInt(af.getY());
		out.writeInt(af.getWidth());
		out.writeInt(af.getHeight());
	}

	public Figure read( DataInputStream in ) throws IOException {
		AbstractFigure af = createFigure();
		af.setX(in.readInt());
		af.setY(in.readInt());
		af.setWidth(in.readInt());
		af.setHeight(in.readInt());
		return af;
	}
```

```
    /**
     * Diese Methode muss von den Unterklassen implementiert werden, damit
     * die Methode <code>read()</code> eine <code>Figure</code> erzeugen kann.
     * Wer die Methode read() überschreibt, kann auch hier null zurück
     * liefern, wenn er sie nicht benutzt.
     *
     * @return AbstractFigure Eine Figure
     */
    public abstract AbstractFigure createFigure();

    public void paint( Graphics g ) {
        if( pressed )
            next.paint( g );
    }

    public void setColor( Color color ) {
        this.color = color;
        if(next != null) {
                next.setColor(color);
          }
    }

    public void setPicture( Picture picture ) {
        this.picture = picture;
    }

    public void setActive( boolean active ) {
        pressed = false;
        next = null;

        this.active = active;
    }

    public void addFactoryListener( FactoryListener listener ) {
        listeners.add( listener );
    }

    public void removeFactoryListener( FactoryListener listener ) {
        listeners.remove( listener );
    }

    // Die leeren Methodenrümpfe der MouseListener und MouseMotionListener
    // Methoden
    public void mouseClicked( MouseEvent e ) {
    }

    public void mousePressed( MouseEvent e ) {
    }

    public void mouseReleased( MouseEvent e ) {
    }

    public void mouseEntered( MouseEvent e ) {
    }

    public void mouseExited( MouseEvent e ) {
    }

    public void mouseDragged( MouseEvent e ) {
    }

    public void mouseMoved( MouseEvent e ) {
    }
}
```

AbstractFigure

```
package tutorial.core;


import java.awt.Color;

/**
 * Diese abstrakte Klasse implementiert die getter und setter Methoden aus
 * dem Interface <code>Figure</code>. Zusätzlich dazu noch ein paar extra
 * setter und getter Methoden um X/Y Position und Breite/Höhe zu setzen/bekommen
 */
public abstract class AbstractFigure implements Figure {

	protected Color color;

	protected int x, y, width, height;


	public int getHeight() {
	    return height;
	}

	public void setHeight( int height ) {
	    this.height = height;
	}

	public int getWidth() {
	    return width;
	}

	public void setWidth( int width ) {
	    this.width = width;
	}

	public int getX() {
	    return x;
	}

	public void setX( int x ) {
	    this.x = x;
	}

	public int getY() {
	    return y;
	}

	public void setY( int y ) {
	    this.y = y;
	}

	public Color getColor() {
	    return color;
	}

	public void setColor( Color color ) {
	    this.color = color;
	}
}
```

Nach diesen Vorbereitungen können wir jetzt unsere erste *FigureFactory*, ein Rechteck, schreiben. Zuerst müssen wir uns allerdings überlegen, wie wir vorgehen müssen. Der Benutzer soll auf die Zeichenfläche klicken, die Maus gedrückt halten, irgendwann und irgendwo wieder loslassen.

Dann soll das Rechteck auf das *Picture* geaddet werden. Während des Ziehens der Maus soll das Rechteck aber auch schon gezeichnet werden, wir müssen also regelmäßig neu zeichnen. Das machen wir einfach über den *FactoryListener*, der unserer *FigureFactory* am Anfang automatisch hinzugefügt wird.
Wir brauchen neben der Factory auch noch eine *Figure*, die gezeichnet und auf das *Picture* geaddet wird. Diese erstellen wir als innere Klasse unserer Factory.

Rectangle

```
package tutorial.factories;

import java.awt.Component;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.event.MouseEvent;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

import javax.swing.Icon;

import tutorial.core.AbstractFigure;
import tutorial.core.AbstractFigureFactory;
import tutorial.core.FactoryListener;
import tutorial.core.Figure;
import tutorial.core.FigureFactory;

/**
 * @author Benjamin Sigg
 *                 und Roar hat den Code <i>etwas</i> gekürzt :-P
 * @version 1.0
 */
public class Rectangle extends AbstractFigureFactory {

      protected Point mouse;


      public Rectangle(){
      }


      public String getID() {
            return "RECT";
      }

      /**
       * Hier malen wir ein schönes Icon, welches in der graphischen Oberfläche
       * unseres Zeichenprogramms dargestellt wird.
       */
      public Icon getIcon() {
          return new Icon(){
              public int getIconHeight() {
                  return 16;
              }
              public int getIconWidth() {
                  return 16;
              }
              public void paintIcon( Component c, Graphics g, int x, int y ) {
                  g.setColor( c.getForeground() );
                  g.drawRect( x, y, 16, 16 );
                  g.drawRect( x+1, y+1, 14, 14 );
                  g.drawRect( x+2, y+2, 12, 12 );
```

```
            }
        };
    }

    public String getName() {
        return "Rechteck";
    }

    /**
     * Diese Methode sagt allen geaddeten <code>FactoryListener</code>n, dass
     * neu gezeichnet werden soll.
     */
    protected void fireRepaint(){
        for( FactoryListener f : listeners )
            f.shouldRepaint( this );
    }

    /**
     * Wird aufgerufen, wenn die Maus geklickt wurde.
     * Wenn diese Factory active ist, wird eine neue
     * <code>RectangleFigure</code> instantiiert.
     */
    public void mousePressed( MouseEvent e ) {
        if( !pressed && active ){
            pressed = true;
            next = new RectangleFigure();
            mouse = e.getPoint();

            next.setX( e.getX() );
            next.setY( e.getY() );
            next.setColor( color );

            fireRepaint();
        }
    }

    /**
     * Diese Methode wird aufgerufen, nachdem die Maus losgelassen wurde.
     * Das Rechteck ist jetzt "fertig" und kann unserem <code>Picture</code>
     * hinzugefügt werden. Die entsprechenden Flags werden zurückgesetzt.
     */
    public void mouseReleased( MouseEvent e ) {
        if( pressed ){
            pressed = false;
            picture.add( next );
            next = null;
        }
    }

    /**
     * Wenn die Maus herumgezogen wird, werden neue Koordinaten gesetzt, und
     * unsere <code>Figure</code> wird neu gezeichnet.
     */
    public void mouseDragged( MouseEvent e ) {
        if( pressed ){
            int x = e.getX();
            int y = e.getY();

            next.setWidth( Math.abs( x - mouse.x ));
            next.setHeight( Math.abs( y - mouse.y ));

            next.setX( Math.min( x, mouse.x ));
            next.setY( Math.min( y, mouse.y ));
```

```
                fireRepaint();
            }
        }

        /**
         * Unsere Figur
         */
        private class RectangleFigure extends AbstractFigure{

            public RectangleFigure() {
              }

              /**
               * Hier wird unser Rechteck gemalt
               */
            public void paint( Graphics g ) {
                g.setColor( color );
                g.drawRect( x, y, width, height );
            }

            public FigureFactory getOrigin() {
                return Rectangle.this;
            }

        }
}
```

Jetzt haben wir schon eine eigene FigureFactory geschrieben. Dank unserer abstrakten Klasse müssen wir nicht auch noch die anderen Methoden aus *Figure* und *MouseListener* implementieren.

Unser Rechteck malt aber nur einen Umriss. Nun wollen wir aber auch ein ausgefülltes Rechteck. Jetzt denkt ihr vielleicht „och nö, nicht alles nochmal schreiben“. Dem ist auch nicht so, da wir unsere neue Klasse *FilledRectangle* einfach von unserer bestehenden Klasse *Rectangle* ableiten. So brauchen wir nur die wichtigen Methoden zu überschreiben. Da wir es nicht allzu kompliziert machen wollen, schreiben wir auch noch eine eigene *Figure*, anstatt die bestehende *RectangleFigure* zu erweitern. Auch wenn sich das nach viel Arbeit anhört: Ist es nicht. Wir brauchen nur den Code zu kopieren und an wenigen Stellen ändern:

FilledRectangle

```
package tutorial.factories;

import java.awt.Component;
import java.awt.Graphics;
import java.awt.event.MouseEvent;

import javax.swing.Icon;

import tutorial.core.AbstractFigure;
import tutorial.core.FigureFactory;


/**
 * Ein ausgefülltes Rechteck.
 */
public class FilledRectangle extends Rectangle {

      /**
       * Das Icon muss natürlich auch anders sein :-)
       */
      public Icon getIcon() {
```

```
            return new Icon(){
                public int getIconHeight() {
                    return 16;
                }
                public int getIconWidth() {
                    return 16;
                }
                public void paintIcon( Component c, Graphics g, int x, int y ) {
                    g.setColor( c.getForeground() );
                    g.fillRect( x, y, 16, 16 );
                }
            };
        }

        public String getID() {
              return "FILLED_RECT";
        }

        public String getName() {
            return "Ausgefülltes Rechteck";
        }

        /**
         * Diese Methode müssen wir überschreiben, da hier statt einer
         * <code>RectangleFigure</code> eine <code>FilledRectangleFigure</code>
         * erzeugt werden muss.
         */
        public void mousePressed( MouseEvent e ) {
            if( !pressed && active ){
                pressed = true;
                next = new FilledRectangleFigure();
                mouse = e.getPoint();

                next.setX( e.getX() );
                next.setY( e.getY() );
                next.setColor( color );

                fireRepaint();
            }
        }

        /** Unsere veränderte Figur */
        private class FilledRectangleFigure extends AbstractFigure {

            public FilledRectangleFigure() {
              }

            public void paint( Graphics g ) {
                g.setColor( color );
                g.fillRect( x, y, width, height );
            }

            public FigureFactory getOrigin() {
                return FilledRectangle.this;
            }

        }
}
```

Aufgabe:

Schreibe eine eigene *FigureFactory* mit dem Klassennamen *Line*, die eine Linie erzeugt.
In der anhängenden Datei findest du den Sourcecode meiner Implementierung der Klasse *Line*.

Ebenfalls findest du dort noch andere Figuren wie *Circle* und *FilledCircle*.

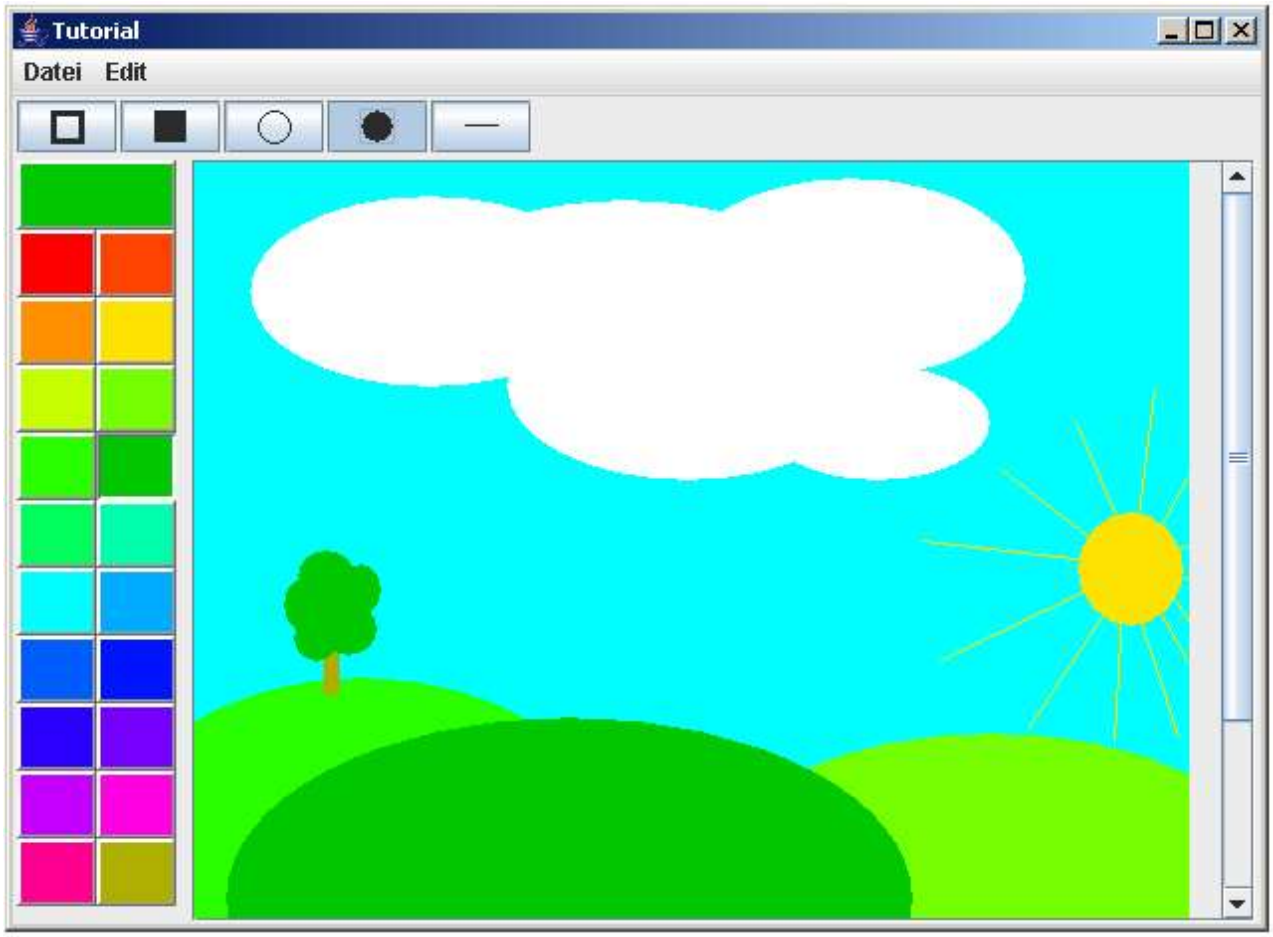
Zwischenstop

Ich danke Roar für die Figuren.

Du hast sicher schon die Methode *listFactories* in *Main* angepasst, um die Figuren auszuprobieren? [2]Falls nicht, dann mach das jetzt.

```
public static FigureFactory[] listFactories(){
      return new FigureFactory[]{
            new Rectangle(),
            new FilledRectangle(),
            new Circle(),
            new FilledCircle(),
            new Line()
      };
}
```

Starten wir das Programm, und sehen, was wir bekommen:



2 Line, Circle und FilledCircle kannst du in dem beiliegenden Code finden. Falls du diesen Code nicht zur Hand hast, dann lösch diese Figuren aus der Liste.

Zusätzliche Features

Unser Programm ist noch nicht fertig. Nun müssen die kleinen Zusätze geschrieben werden, welche ein Programm richtig benutzbar machen.

Natürlich sind auch diese Zusätze wieder von einander abhängig, deshalb beginnen wir mit dem Feature, welches wohl den grössten Einfluss haben wird: dem Speichern und dem Laden von Bildern.

Speichern und Laden

Glücklicherweise haben wir diese Möglichkeit schon früh eingeplant, und jeder *FigureFactory* die Methoden *read*, *write* und *getID* aufgezwungen.

Was wir jetzt benötigen ist ein Algorithmus, der das Bild speichert/lädt, und jemanden der verwaltet, wann wo geschrieben wird.

Wir werden eine Klasse *PictureIO* für den Algorithmus schreiben, sowie eine Klasse *FileManager* für die Verwaltung implementieren. Das hat den Vorteil, dass wir den Algorithmus auch an einer anderen Stelle benutzen könnten (in dem Tutorial wird dies nicht der Fall sein, aber man könnte z.B. eine automatische Speicherfunktion einbauen...).

Zuerst aber noch eine Erklärung von Roar, was die Factories mit dem Speichern zu tun haben.

Factories und das Speichern/Laden

Weiter oben haben wir unsere abstrakte Klasse *AbstractFigureFactory* und unsere Figuren implementiert. Jetzt werden wir uns das noch einmal genauer anschauen. In *AbstractFigureFactory* haben wir drei Methoden deklariert, die im Kontext mit speichern und laden stehen:

- Eine Methode um eine *Figure* zu speichern
  void write(Figure figure, DataOutputStream out)
- Eine Methode um eine *Figure* zu laden
  Figure load(DataInputStream input)
- Und eine abstrakte Methode, die eine Unterklasse von *AbstractFigure* zurück liefern muss
  abstract AbstractFigure createFigure()

Zuerst wenden wir uns der dritten, abstrakten, Methode zu. Diese Methode muss von allen Unterklassen der *AbstractFigureFactory* überschrieben werden, da wir sie als abstrakt deklariert haben. Die Methode soll uns, laut Deklaration, eine Instanz des Typs *AbstractFigure* zurückliefern. Die Unterklassen müssen also eine neue Instanz ihrer Figur, in unserem Beispiel eine *RectangleFigure* erstellen, die sie zurück liefert. Wozu brauchen wir diese Methode? Die Methode *read*() braucht sie, denn sie muss ja eine Figur instantiieren. Das geht nicht anders als diese Aufgabe an die Unterklassen weiter zu delegieren.

Die anderen beiden Methoden sind recht simpel. Sie arbeiten beide mit unserer Klasse *AbstractFigure*. Beide Methoden lesen bzw. schreiben nur die *x*-/*y*-Position und die *width*/*height* Attribute in den Stream. Die Farbe der *Figure* wird bereits von der Klasse *PictureIO* (Die später erläutert wird) geschrieben.

Einiges gilt es noch zu beachten: Wenn die *FigureFactory* nicht von *AbstractFigureFactory* erbt müssen die beiden read/write Methoden natürlich neu implementiert werden. Das gilt vor allem wenn die dazugehörige *Figure* nicht von *AbstractFigure* erbt. Wenn dann die Methoden nicht überschrieben werden, wird eine Exception geworfen.

PictureIO

Es ist nicht weiter schwierig herauszufinden, was diese Klasse alles benötigt:

- Wir müssen Methoden um Bilder zu lesen und zu schreiben implementieren. Diese Methoden sollen *read*, *write* und *getID* von *FigureFactory* verwenden. Um Dateien zu benutzen benötigen wir auch noch einen *FileInputStream*, einen *FileOutputStream*, einen *DataInputStream* und *DataOutputStream*, alle stammen aus *java.io*.
  public void write(Picture picture, File file) throws IOException
  public Picture read(File file) throws IOException
- Um Dateien zu lesen, benötigen wir eine Liste aller verfügbaren Factories, wir bauen deshalb im Konstruktor ein entsprechendes Argument ein.
  public PictureIO(FigureFactory[] factories)

Es gibt nun verschiedene Varianten die Dateien aufzubauen, wichtig ist nur, dass *read* und *write* „symmetrisch“ zueinander sind.

Wir dürfen nicht vergessen die Farben der Figuren abzuspeichern, denn wir haben bei dem Interface *FigureFactory* deutlich gesagt, dass die Farbe *nicht* von den Factories gespeichert werden muss.

Versuch dein Glück, die Klasse könnte so aussehen:

```
package tutorial.io;

import java.awt.Color;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Hashtable;

import tutorial.core.Figure;
import tutorial.core.FigureFactory;
import tutorial.core.Picture;

/**
 * Diese Klasse bietet die Möglichkeit Bilder als Dateien zu speichern, und
 * später wieder auszulesen.
 */
public class PictureIO{
    /** Die Factories, abrufbar über ihre ID's */
    private Hashtable<String, FigureFactory> factories;

    /**
     * Standardkonstruktor
     * @param factories Die Factories des Programmes
     */
    public PictureIO( FigureFactory[] factories ){
        this.factories = new Hashtable<String, FigureFactory>();

        for( FigureFactory f : factories )
            this.factories.put( f.getID(), f );
    }

    /**
     * Schreibt ein Bild in eine Datei.
     * @param picture Das zu speichernde Bild
     * @param file Die Datei
```

```
     * @throws IOException Sollten irgendwelche Probleme auftreten
     */
    public void write( Picture picture, File file ) throws IOException{
        DataOutputStream out = new DataOutputStream(
            new FileOutputStream( file ));
        int size = picture.getSize();

        // Höhe und Breite speichern
        out.writeInt( picture.getWidth() );
        out.writeInt( picture.getHeight() );

        // Anzahl Elemente speichern
        out.writeInt( size );

        for( int i = 0; i < size; i++ ){
            Figure figure = picture.get( i );
            FigureFactory factory = figure.getOrigin();

            // Elemente speichern: Factory -> Figur -> Farbe
            out.writeUTF( factory.getID() );
            factory.write( figure, out );
            out.writeInt( figure.getColor().getRGB() );
        }

        out.close();
    }

    /**
     * Liest den Inhalt einer Datei, und liefert das gespeicherte Bild zurück.
     * @param file Die zu lesende Datei
     * @return Das gespeicherte Bild
     * @throws IOException Sollten Probleme auftreten
     */
    public Picture read( File file ) throws IOException{
        DataInputStream in = new DataInputStream( new FileInputStream( file ));

        // Höhe und Breite auslesen
        Picture picture = new Picture( in.readInt(), in.readInt() );

        // Anzahl Elemente auslesen
        int size = in.readInt();

        for( int i = 0; i < size; i++ ){
            // Elemente lesen: Factory -> Figur -> Farbe
            FigureFactory factory = factories.get( in.readUTF() );
            Figure figure = factory.read( in );
            figure.setColor( new Color( in.readInt() ));

            picture.add( figure );
        }

        in.close();

        return picture;
    }
}
```

SimpleFileFilter

Wir werden zu verschiedenen Anlässen einen Datei-Auswahl-Dialog anzeigen müssen. Dazu werden wir den *JFileChooser* benutzen.

Da der Benutzer nicht wild irgendwelche Dateien erzeugen oder laden soll, müssen wir dem *JFileChooser* sagen können, welche Dateien legal, und welche nicht legal sind.

Wir können dies mit Hilfe der abstrakten Klasse *FileFilter* tun, von der wir erben müssen, noch zwei Methoden schreiben.

Die Aufgabe ist klar:

- Die Klasse *SimpleFileFilter* erbt von *javax.swing.filechooser.FileFilter*, und implementiert mehrere abstrakte Methoden.
  public boolean accept(File file)
  public String getDescription()
- Wir werden garantiert mehrmals einen FileFilter benötigen, der Bilder erkennt. Um nicht jedesmal eine neue Instanz des *SimpleFileFilters* anzulegen, und auch um die Gefahr von Schreibfehler zu minimieren, benutzen wir eine [3]statische, konstante Instanz.
  public static final SimpleFileFilter PICTURE_FILTER
- Der SimpleFileFilter soll mit zwei Strings arbeiten: einen der die Endung der Dateien beschreibt, und einen der den Typ Datei beschreibt (der Benutzer wird diesen zweiten String lesen können). Entsprechende Variablen und Methoden werden benötigt.
  private String extension
  private String description
  verschiedene Getter und Setter

Versuche den *SimpleFileFilter* zu implementieren, hier ist meine Lösung:

```
package tutorial.io;

import java.io.File;

import javax.swing.filechooser.FileFilter;

/**
 * Dieser FileFilter sortiert alle Dateien aus, welche nicht eine übergebene
 * Endung verwenden.
 */
public class SimpleFileFilter extends FileFilter{
    /** Dieser FileFilter beschreibt die Picture-Dateien */
    public static final SimpleFileFilter PICTURE_FILTER =
        new SimpleFileFilter( "pic", "Bilder (*.pic)" );

    /** Endung die eine Datei besitzen muss */
    private String extension;

    /** Beschreibung dieses Formates */
    private String description;

    /**
     * Standardkonstruktor
     * @param extension Die Endung, welche Dateien haben müssen.
     * Zum Beispiel "txt".
     * @param description Eine kleine Beschreibung des Dateityps
     */
    public SimpleFileFilter(String extension, String description) {
        this.extension = extension.toLowerCase();
        this.description = description;
    }

    public void setExtension( String extension ) {
        this.extension = extension.toLowerCase();
```

3 Statisch ist etwas ähnliches wie global: egal wo man im Programm ist, man hat Zugriff auf diese Variable.

```
    }
    public String getExtension() {
        return extension;
    }

    public boolean accept( File f ) {
        return f.isDirectory() || f.getPath().toLowerCase().endsWith(
            "." + extension );
    }

    public void setDescription( String description ) {
        this.description = description;
    }
    public String getDescription() {
        return description;
    }
}
```

FileManager

Gute Programme zeichnen sich unter anderem darin aus, dass sie den Benutzer

1. ... nicht mit unnötigen Fragen quälen („Das Bild wurde nicht verändert, möchten Sie es deshalb auch nicht speichern?“)
2. ... für Aktionen welche oft getätigt werden Abkürzungen bieten.
3. Die Einstellungen welche der Benutzer gesetzt hat, nicht gleich wieder löschen (Eine Krankheit die viele Programme haben ist z.B. die Vermutung, dass der Benutzer ausschließlich Dateien im Home-Verzeichnis besitzt, und der Datei-Dialog deshalb grundsätzlich in das Home-Verzeichnis zeigen muss...).

Es gibt also hohe Ansprüche an die nächste Klasse, die wir schreiben.

Der *FileManager* soll immer dann benutzt werden, wenn es um das Schreiben oder Lesen eines Bildes geht. Auch wenn wir jetzt noch nichts entsprechendes einbauen, so wird der *FileManager* auch benutzt werden um JPEG (und andere) Bilder herzustellen. Wir müssen die Implementierung also genug offen lassen, dass die Erweiterung später nicht schwer sein wird.

Wichtig für den *FileManager* ist, dass er sich die zuletzt geöffnete/geschriebene Datei merkt, damit der Benutzer ein Bild in diese Datei speichern kann, ohne dass er den Datei-Dialog überhaupt öffnen muss (Punkt 2 auf der Liste).

Zusätzlich soll der *FileManager* wissen, ob ein Bild verändert wurde, oder nicht. Dies wird uns die Möglichkeit geben, z.B. vor dem Öffnen einer anderen Datei den Benutzer zu fragen, ob er das veränderte Bild zuerst speichern möchte (Punkt 1 auf der Liste: wenn das Bild nicht verändert wurde, muss es auch nicht gespeichert werden).

Wir übertragen dem *FileManager* auch gleich die Verantwortung den Datei-Dialog zu verwalten.

Die Liste der Methoden, welche wir schreiben müssen, ist länger:

- Der Manager muss sicher unterrichtet werden, welches Bild zur Zeit bearbeitet wird.
  public void setPicture(Picture picture)
- Dann soll der Manager eine Angaben machen können, ob das Bild seit dem Laden oder Speichern verändert wurde. Um dies herauszufinden, lassen wir ihn den PictureListener implementieren, immer wenn eine Methode des PictureListeners angesprochen wird, wissen wir, dass das Bild verändert wurde.
  implements PictureListener
  public boolean isSaved()

- Ziemlich offensichtlich ist, dass eine Methode geschrieben werden muss, welche das Bild speichert. Diese Methode soll den Datei-Dialog öffnen, damit der Benutzer eingeben kann, wo das Bild gespeichert wird.
  Um die aufrufende Methode zu unterrichten, ob der Benutzer das Speichern abgebrochen hat, geben wir einen *boolean* zurück (dieser *boolean* ist *true*, wenn der Benutzer nicht abgebrochen hat).
  Eine weitere Anforderung: die Methode soll den Benutzer warnen, wenn er eine Datei überschreiben lassen will, in diesem Fall kann der Benutzer eine andere Datei wählen (sofern er die Aktion nicht abbrechen will).
  public boolean saveAs()
- Wurde bereits eine Datei geöffnet oder geschrieben, kann man eine Erleichterung einbauen: der Benutzer kann das Bild direkt wieder in diese Datei schreiben lassen.
  Sollte die Methode aber aufgerufen werden, ohne dass die Datei bekannt ist, leiten wir den Aufruf einfach an *saveAs* weiter.
  public void save()
- Es wird ab und zu vorkommen, dass wir den Benutzer warnen müssen, er könnte das Bild löschen. Eine solche Situation tritt z.B. ein, wenn der Benutzer ein neues Bild herstellt, ohne das alte zu speichern...
  Wir verwenden dazu einen Dialog, der dem Benutzer auf die Frage „Das Bild wurde verändert, möchten Sie es speichern?“ drei Antwortmöglichkeiten bietet: „Ja“, „Nein“ und „Abbrechen“. Wenn der Benutzer „Ja“ wählt, wird die Methode *saveAs* aufgerufen, bei den anderen beiden Optionen geschieht nichts.
  Um „Nein“ und „Abbrechen“ doch noch sinnvoll zu verarbeiten, lassen wir die Methode einen *boolean* zurückgeben. Dieser *boolean* ist *true*, wenn das Programm weiterarbeiten kann, *false* wenn der Benutzer „Abbrechen“ gewählt hat. Diese Definition deckt sich „zufälligerweise“ mit der Definition des Rückgabewerts von *saveAs*, und sollte der Benutzer „Ja“ gedrückt haben, geben wir genau diesen Rückgabewert (von *saveAs*) zurück.
  public boolean questionSave()
- Nach all diesen Methoden die speichern, brauchen wir noch eine Methode, die eine Datei liest. Doch bevor eine Datei gelesen wird, sollte die Methode nachfragen, ob der Benutzer das alte Bild speichern möchte. Dies können wir nun elegant an *questionSave* delegieren.
  public void read()
- Noch ein Wort zum internen Aufbau: wir werden den *JFileChooser* für verschiedene Zwecke benötigen. Deshalb ist es eine gute Idee Methoden zu schreiben, welche den Filechooser auf eine spezielle Aufgabe vorbereiten.

Lass dir Zeit, und versuche etwas stabiles aufzubauen. Wie immer gebe ich eine Alternativversion an.

```
package tutorial.io;

import java.io.File;
import java.io.IOException;

import javax.swing.JFileChooser;
import javax.swing.JOptionPane;
import javax.swing.filechooser.FileFilter;

import tutorial.core.Figure;
```

```java
import tutorial.core.FigureFactory;
import tutorial.core.Picture;
import tutorial.core.PictureListener;
import tutorial.gui.MainFrame;

/**
 * Der FileManager kann Bilder als Datei schreiben oder lesen. Außerdem
 * merkt sich der FileManager, welche Datei zuletzt benutzt wurde, so kann
 * ein Bild gespeichert werden, ohne dass der Benutzer jedesmal eine Datei
 * auswählen muss.
 */
public class FileManager implements PictureListener{
    /** Die zuletzt geöffnete oder gespeicherte Datei */
    private File file;
    /** Angabe, ob das aktuelle Bild in seinem derzeitigen Zustand gespeichert
      * ist */
    private boolean saved = false;

    /** Das aktuelle Bild */
    private Picture picture;
    /** Die Algorithmen um ein Bild zu laden und zu speichern */
    private PictureIO pictureIO;

    /** Der Dialog, mit dem Dateien ausgewählt werden */
    private JFileChooser fileChooser;
    /** Das Hauptfenster dieses Programmes */
    private MainFrame mainFrame;

    /**
     * Standardkonstruktor
     * @param mainFrame Das Hauptfenster dieses Programmes
     * @param factories Die Factories
     */
    public FileManager( MainFrame mainFrame, FigureFactory[] factories ){
        this.mainFrame = mainFrame;
        pictureIO = new PictureIO( factories );
    }

    /**
     * Öffnet eine Dialog, auf dem der Benutzer eine Datei auswählen kann.
     * Diese Datei wird dann ausgelesen, und sollte tatsächlich ein Bild
     * gespeichert sein, wird dieses Bild angezeigt.<br>
     * Sollte ein nicht gespeichertes Bild existieren, wird der Benutzer
     * zuerst gefragt, ob dieses Bild gespeichert werden soll.
     */
    public void read(){
        if( picture != null && !isSaved() )
            if( !questionSave() )
                return;

        prepareReadPicture();

        if( fileChooser.showOpenDialog( mainFrame ) ==
                JFileChooser.APPROVE_OPTION ){
            File file = fileChooser.getSelectedFile();

            try{
                Picture picture = pictureIO.read( file );

                mainFrame.setPicture( picture );
                saved = true;
                this.file = file;
            }
            catch( IOException ex ){
                JOptionPane.showMessageDialog( mainFrame,
```

```
                    "Ein Fehler trat beim Lesen auf:\n"
                    + ex.getMessage(), "Lesefehler",
                    JOptionPane.ERROR_MESSAGE  );
        }
        catch( Throwable t ){
            JOptionPane.showMessageDialog( mainFrame,
                    "Ein schwerer Fehler trat beim lesen auf.\n" +
                    "Überprüfen Sie, ob sie tatsächlich eine "+
                    "Bild-Datei geöffnet haben:\n"
                    + t.getMessage(), "Lesefehler",
                    JOptionPane.ERROR_MESSAGE  );
        }
    }
}

/**
 * Öffnet einen Dialog, auf welchem der Benutzer 3 Auswahlmöglichkeiten
 * hat.<br>
 * Das aktuelle Bild speichern, dann wird {@link #saveAs()} aufgerufen.<br>
 * Das aktuelle Bild nicht speichern.<br>
 * Die Aktion abzubrechen.
 * @return false, sollte die Aktion abgebrochen worden sein, ansonsten true
 */
public boolean questionSave(){
    int answer = JOptionPane.showConfirmDialog( mainFrame,
            "Das Bild ist nicht gespeichert,\nmöchten Sie das Bild " +
              "speichern?",
            "Bild speichern", JOptionPane.YES_NO_CANCEL_OPTION,
            JOptionPane.WARNING_MESSAGE );

    if( answer == JOptionPane.CANCEL_OPTION )
        return false;
    else if( answer == JOptionPane.NO_OPTION )
        return true;
    else
        return saveAs();
}

/**
 * Speichert das aktuelle Bild in der zuletzt geöffneten oder geschriebenen
 * Datei.<br>
 * Sollte keine solche Datei existieren, wird die Methode
 * {@link #saveAs() saveAs()} aufgerufen.
 */
public void save(){
    if( picture != null ){
        if( file == null ){
            saveAs();
        }
        else{
            try{
                pictureIO.write( picture, file );
                saved = true;
            }
            catch( IOException ex ){
                JOptionPane.showMessageDialog( mainFrame,
                        "Ein Fehler trat beim Speichern auf:\n"
                        + ex.getMessage(), "Schreibfehler",
                        JOptionPane.ERROR_MESSAGE  );

                saved = false;
            }
        }
    }
}
```

```
    /**
     * Speichert das aktuelle Bild in einer neuen Datei. Der Benutzer kann
     * von einem JFileChooser die Datei wählen.
     * @return false, wenn der Benutzer das Speichern abgebrochen hat, ansonsten
     * true.
     */
    public boolean saveAs(){
        if( picture != null ){
            prepareWritePicture();

            while( true ){
                // Die Aktion wird solange wiederholt, bis der Benutzer entweder
                // das Speichern abbricht, oder das Bild gespeichert ist.
                if( fileChooser.showSaveDialog( mainFrame ) ==
                        JFileChooser.APPROVE_OPTION ){
                    File file = fileChooser.getSelectedFile();

                    SimpleFileFilter filter =
                        (SimpleFileFilter)fileChooser.getFileFilter();
                    if( !filter.accept( file ))
                        file = new File( file.getPath() + "." +
                              filter.getExtension() );

                    if( file.exists() ){
                        int answer = JOptionPane.showConfirmDialog( mainFrame,
                                "Die Datei \"" + file.getName() +
                                "\" existiert bereits,\nsoll sie "+
                                   "überschrieben werden?",
                                "Datei überschreiben",
                                JOptionPane.YES_NO_CANCEL_OPTION,
                                JOptionPane.QUESTION_MESSAGE );

                        if( answer == JOptionPane.CANCEL_OPTION )
                            return false; // Abbruch des Speicherns
                        else if( answer == JOptionPane.NO_OPTION )
                            continue; // Andere Datei anfordern
                    }

                    try{
                        pictureIO.write( picture, file );
                        saved = true;
                        this.file = file;
                        return true;
                    }
                    catch( IOException ex ){
                        JOptionPane.showMessageDialog( mainFrame,
                                "Ein Fehler trat beim Speichern auf:\n"
                                + ex.getMessage(), "Schreibfehler",
                                JOptionPane.ERROR_MESSAGE  );

                        saved = false;
                        return false;
                    }
                }
                else
                    return false;
            }
        }
        else
            return true;
    }

    /**
     * Präpariert den FileChooser so, dass Picture-Dateien gelesen
```

```
     * werden können.
     */
    private void prepareReadPicture(){
        ensureFileChooserInit();
        emptyFileChooser();
        fileChooser.addChoosableFileFilter( SimpleFileFilter.PICTURE_FILTER );
        fileChooser.setAcceptAllFileFilterUsed( true );
        fileChooser.setFileFilter( SimpleFileFilter.PICTURE_FILTER );
    }

    /**
     * Präpariert den FileChooser so, dass Picture-Dateien geschrieben
     * werden können.
     */
    private void prepareWritePicture(){
        ensureFileChooserInit();
        emptyFileChooser();
        fileChooser.addChoosableFileFilter( SimpleFileFilter.PICTURE_FILTER );
        fileChooser.setAcceptAllFileFilterUsed( false );
    }

    /**
     * Entfernt alle FileFilter des FileChoosers
     */
    private void emptyFileChooser(){
        FileFilter[] filters = fileChooser.getChoosableFileFilters();
        for( FileFilter f : filters )
            fileChooser.removeChoosableFileFilter( f );
    }

    /**
     * Überprüft, ob der FileChooser initialisiert wurde. Falls dies
     * noch nicht geschehen ist, wird es gemacht.
     */
    private void ensureFileChooserInit(){
        if( fileChooser == null ){
            fileChooser = new JFileChooser();
            fileChooser.setMultiSelectionEnabled( false );
        }
    }

    /**
     * Setzt das aktuelle Bild
     * @param picture Das aktuelle Bild
     */
    public void setPicture( Picture picture ){
        if( this.picture != null )
            this.picture.removePictureListener( this );

        this.picture = picture;

        if( picture != null )
            picture.addPictureListener( this );

        file = null;
        saved = false;
    }

    /**
     * Gibt an, ob das aktuelle Bild in seinem jetzigen Zustand
     * gespeichert wurde.
     * @return true, sollte das Bild gespeichert sein, false sonst
     */
    public boolean isSaved(){
        return saved;
```

```
    }

    public void figureAdded( Figure figure, Picture picture ) {
        saved = false;
    }

    public void figureRemoved( Figure figure, Picture picture ) {
        saved = false;
    }
}
```

Das Ganze einbauen

Jetzt müssen wir unsere neuen Klassen natürlich testen.

Wir werden den *FileManager* in das Hauptmenü integrieren (dies ist ein reiner Willkürentscheid, ebenso gut können wir das Hauptfenster nehmen).
Da der Konstruktor des Managers eine Liste von verfügbaren Factories benötigt, werden wir den Konstruktor des *MainMenu's* leicht abändern, so dass auch er eine Liste der Factories benötigt. Diese Factories können wir dann vom *MainFrame* aus leicht setzen.
Zusätzlich werden wir 3 neue Menüeinträge schreiben, „Öffnen“, „Speichern“ und „Speichern als“, wobei die letzten beiden nur dann verfügbar sein sollen, wenn auch tatsächlich ein Bild existiert.

Bau den *FileManager* ein. Solltest du Probleme haben: hier sind die Teile die im Code verändert werden müssen.

Veränderungen im MainMenu

```
public class MainMenu extends JMenuBar implements ActionListener,
            PictureListener{


      /** Der FileManager koordiniert schreiben und lesen von Dateien */
      private FileManager fileManager;

      /** Die MenuItems, welche das Speichern des Bildes erlauben */
      private JMenuItem itemSave, itemSaveAs;

      [...]

      public MainMenu( MainFrame frame, FigureFactory[] factories ){

            [...]

            JMenuItem readPicture = new JMenuItem( "Öffnen" );
            readPicture.setActionCommand( "read" );
            readPicture.addActionListener( this );

            itemSave = new JMenuItem( "Speichern" );
            itemSave.setActionCommand( "save" );
            itemSave.addActionListener( this );
            itemSave.setEnabled( false );

            itemSaveAs = new JMenuItem( "Speichern als..." );
            itemSaveAs.setActionCommand( "saveAs" );
            itemSaveAs.addActionListener( this );
            itemSaveAs.setEnabled( false );

            [...]

            file.add( newPicture );
```

```
            file.addSeparator();
            file.add( readPicture );
            file.add( itemSave );
            file.add( itemSaveAs );

            [...]

            fileManager = new FileManager( frame, factories );

            [...]
        }

        /**
         * Stellt ein neues Bild her, und sorgt dafür, dass das Programm
         * dieses neue Bild auch benutzt.
         */
        public void newPicture(){
            if( picture != null && !fileManager.isSaved())
                if( !fileManager.questionSave() )
                    return;

            if( newPictureDialog == null )
                newPictureDialog = new NewPictureDialog( frame );

            Picture picture = newPictureDialog.createPicture();

            if( picture != null )
                frame.setPicture( picture );
        }

        /**
         * Gibt den FileManager des Programmes zurück.
         * @return Der FileManager
         */
        public FileManager getFileManager(){
            return fileManager;
        }

        public void setPicture( Picture picture ){
            [...]

            fileManager.setPicture( picture );

            itemSave.setEnabled( picture != null );
            itemSaveAs.setEnabled( picture != null );

            [...]
        }

        public void actionPerformed( ActionEvent e ){
            [...]
            else if( command.equals( "read" ))
                fileManager.read();

            else if( command.equals( "save" ))
                fileManager.save();

            else if( command.equals( "saveAs" ))
                fileManager.saveAs();
        }
        [...]
}
```

Veränderungen im MainFrame

```
public class MainFrame extends JFrame{
	[...]
	public MainFrame( FigureFactory[] factories ){
		[...]
		mainMenu = new MainMenu( this, factories );
		setJMenuBar( mainMenu );
	}
	[...]
}
```

It's not a bug, it's a feature

Vielleicht fällt dir etwas seltsames auf: wenn du das Programm startest, und eine Datei lädst, ist das Undo/Redo für das gerade eben geladene Bild vorhanden.
Die Ursache ist, dass wir den *URManager* so aufgebaut haben, dass er aufgrund der noch im Bild vorhandenen Figuren entscheidet, ob ein Undo möglich ist.

Natürlich könnten wir diesen Fehler „gewaltsam“ unterdrücken (z.B. mit einem zusätzlichen Zähler), aber wieso sollten wir das tun?
Der Fehler stört niemanden, im Gegenteil, er könnte sogar praktisch sein. Und deshalb definieren wir dieses Verhalten von „Bug“ in „Feature“ um.

Solche unerwarteten Nebeneffekte treten besonders bei grösseren Projekten auf. Mit einem soliden Design ist es uns möglich diese Effekte auszunutzen, und unserem Programm neue Features einzubauen. Haben wir das Programm zu starr gebaut (z.B. wenn die Abhängigkeiten verschiedener Klassen zu groß sind), kriegen wir lediglich einen Programmabsturz.

(P.S: ich war übrigens ebenfalls ziemlich überrascht, als Roar diesen Fehler entdeckte)

Programm schließen

Nach diesem harten Brocken wollen wir mal wieder etwas einfaches machen: das Menu benötigt endlich einen Eintrag „Beenden“. Wenn der Benutzer diesen Eintrag wählt, soll die Methode *exit* in *MainFrame* aufgerufen werden.

Die Methode *exit* wiederum soll so gebaut sein, dass der Benutzer zuerst gefragt wird, ob er das aktuelle Bild nicht gerne speichern möchte. Glücklicherweise haben wir im *FileManager* eine entsprechende Methode geschrieben (*questionSave*), welche wir nun benutzen werden.

Implementiere das! Ich glaub zwar nicht, dass du Hilfe benötigst, doch für alle Fälle:

Veränderungen in MainMenu

```
public class MainMenu extends JMenuBar implements ActionListener,
		PictureListener{

	[...]

	public MainMenu( MainFrame frame, FigureFactory[] factories ){
		[...]

		JMenuItem exit = new JMenuItem( "Beenden" );
		exit.setActionCommand( "exit" );
		exit.addActionListener( this );

		[...]

		file.addSeparator();
		file.add( exit );
```

```
            [...]
        }


        public void actionPerformed( ActionEvent e ){

            [...]

            else if( command.equals( "exit" ))
                frame.exit();
        }
        [...]
}
```

Veränderungen im MainFrame

```
public class MainFrame extends JFrame{
        [...]

        /**
         * Diese Methode beendet das Programm.
         */
        public void exit(){
            FileManager file = mainMenu.getFileManager();

            if( picture != null && !file.isSaved() )
                if( !file.questionSave() )
                    return;

            System.exit( 0 );
        }
        [...]
}
```

Neues Bild herstellen

Wir müssen dem Benutzer die Möglichkeit geben verschiedene Größen für das Bild zu wählen.

Dazu werden wir einen neuen Dialog bauen, der zwei *JSpinner* besitzt. In diese *JSpinner* können Integer-Werte eingetragen werden, sofern dieser Werte grösser als 0 sind.
Außerdem werden wir einen „Ok“ und einen „Abbrechen“-Knopf auf den Dialog setzen.

Außerdem werden wir auch vor dem Erzeugen eines neuen Bildes den Benutzer fragen, ob er das alte, veränderte Bild speichern möchte. Wir werden wiederum *questionSave* aus dem *FileManager* benutzen.

Der Dialog, wir nennen ihn *NewPictureDialog* hat geringe Anforderungen:

- Wir müssen eine Möglichkeit besitzen den Dialog sichtbar zu machen. Sollte der Dialog mit „Ok“ geschlossen werden, geben wir das neue Bild zurück, ansonsten null.
  Um weitere Benutzereingaben zu verhindern, werden wir einen modalen Dialog verwenden. Solche Dialoge blockieren das Programm, solange sie sichtbar sind.
  public Picture createPicture
- Es sollten zwei Buttons „Ok“ und „Abbrechen“ sichtbar sein. Drückt der Benutzer „Ok“, wird ein neues Bild hergestellt, sonst nicht.
  Wenn der Benutzer den Standard-Dialog-Schließen-Knopf drückt, soll dasselbe passieren, wie wenn er auf „Abbrechen“ drückt.

- Die Zahlen werden mit Hilfe von zwei *JSpinnern* eingegeben. Damit die Spinner nur positive Integerwerte zulassen, werden wir zwei *SpinnerNumberModel's* benutzen. Diese Models werden den Spinnern im Konstruktor übergeben, und beeinflussen, was der Spinner anzeigen kann.

Versuch es, hier ist ein Bild, wie der Dialog aussehen könnte.
Vergiss nicht, auch die Methode *newPicture* im Hauptmenü muss überarbeitet werden.

Und hier der dazugehörende Code:

```
package tutorial.gui;

import java.awt.Container;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.GridLayout;
import java.awt.Insets;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSpinner;
import javax.swing.SpinnerNumberModel;

import tutorial.core.Picture;

/**
 * Dieser Dialog kreiert ein neues Bild. Der Benutzer
 * kann die Größe wählen.
 */
public class NewPictureDialog extends JDialog{
    /** Die NumberModels beschreiben Breite und Höhe des neuen Bildes */
    private SpinnerNumberModel pictureWidth, pictureHeight;

    /** Die Knöpfe um den Dialog zu schließen */
    private JButton buttonOk, buttonCancel;

    /** Angabe, ob der Dialog mit "OK" geschlossen wird */
    private boolean ok = false;

    /**
     * Erstellt einen neuen modalen Dialog.
     * @param owner Das Frame, welches von dem Dialog verdeckt werden soll
     */
    public NewPictureDialog( JFrame owner ){
        super( owner, "Neues Bild", true );

        // Initialisieren der Elemente
        pictureWidth = new SpinnerNumberModel( 500, 1, Integer.MAX_VALUE, 1 );
```

```
        pictureHeight = new SpinnerNumberModel( 500, 1, Integer.MAX_VALUE, 1 );

        JSpinner spinnerWidth = new JSpinner( pictureWidth );
        JSpinner spinnerHeight = new JSpinner( pictureHeight );

        buttonOk = new JButton( "Ok" );
        buttonCancel = new JButton( "Abbrechen" );

        // Aktionen für das Schliessen festlegen
        Listener listener = new Listener();
        buttonOk.addActionListener( listener );
        buttonCancel.addActionListener( listener );
        setDefaultCloseOperation( JDialog.DISPOSE_ON_CLOSE );

        // Graphische Oberfläche zusammenbauen
        Container content = getContentPane();
        content.setLayout( new GridBagLayout() );

        JPanel buttonPanel = new JPanel( new GridLayout( 1, 2 ));
        buttonPanel.add( buttonOk );
        buttonPanel.add( buttonCancel );

        content.add( new JLabel( "Breite (Pixel)" ), new GridBagConstraints(
                0, 0, 1, 1, 1.0, 1.0, GridBagConstraints.NORTHWEST,
                GridBagConstraints.NONE, new Insets( 3, 3, 3, 3 ), 0, 0 ));
        content.add( spinnerWidth, new GridBagConstraints(
                1, 0, 1, 1, 100.0, 1.0, GridBagConstraints.NORTH,
                GridBagConstraints.HORIZONTAL, new Insets( 3, 3, 3, 3 ),
                0, 0 ));

        content.add( new JLabel( "Höhe (Pixel)" ), new GridBagConstraints(
                0, 1, 1, 1, 1.0, 1.0, GridBagConstraints.NORTHWEST,
                GridBagConstraints.NONE, new Insets( 3, 3, 3, 3 ), 0, 0 ));
        content.add( spinnerHeight, new GridBagConstraints(
                1, 1, 1, 1, 100.0, 1.0, GridBagConstraints.NORTH,
                GridBagConstraints.HORIZONTAL, new Insets( 3, 3, 3, 3 ),
                0, 0 ));

        content.add( buttonPanel, new GridBagConstraints(
                0, 2, 2, 1, 1.0, 1.0, GridBagConstraints.SOUTHEAST,
                GridBagConstraints.NONE, new Insets( 3, 3, 3, 3 ), 0, 0 ));

        // Größe setzen
        pack();
        setLocationRelativeTo( owner );
    }

    /**
     * Erstellt ein neues Bild. Der Dialog wird geöffnet, und die Methode
     * blockiert solange, bis der Dialog wieder geschlossen wurde.
     * @return Das neue Bild oder null, sollte "Abbrechen" gewählt worden sein
     */
    public Picture createPicture(){
        ok = false;
        setVisible( true );
        if( ok )
            return new Picture( pictureWidth.getNumber().intValue(),
                    pictureHeight.getNumber().intValue() );
        else
            return null;
    }

    /**
     * Eine Implementation eines ActionListeners. Dieser Listener wird dem
     * "Ok"- und dem "Abbrechen"-Knopf hinzugefügt.
```

```
     */
    private class Listener implements ActionListener{
        public void actionPerformed( ActionEvent e ) {
            ok = e.getSource() == buttonOk;
            dispose();
        }
    }
}
```

Außerdem muss das Hauptmenü leicht abgeändert werden.

Veränderungen in MainMenu

```
public class MainMenu extends JMenuBar implements ActionListener,
            PictureListener{

      /** Dialog um neue Bilder herzustellen */
      private NewPictureDialog newPictureDialog;

      [...]
      public void newPicture(){
          if( picture != null && !fileManager.isSaved())
              if( !fileManager.questionSave() )
                  return;

          if( newPictureDialog == null )
              newPictureDialog = new NewPictureDialog( frame );

            Picture picture = newPictureDialog.createPicture();

            if( picture != null )
                frame.setPicture( picture );
      }
      [...]
}
```

Exportieren

Was jetzt noch fehlt ist eine Möglichkeit unsere Bilder in „echte“ Bild umzuwandeln, also z.B. in ein PNG oder ein JPEG-Bild.

Als erstes müssen wir ein *Picture* in ein *Image* umwandeln können. Hierzu erstellen wir einfach ein neues *BufferedImage*, und fordern über *createGraphics* eine Instanz von *Graphics* an.
Dieses *Graphics*-Objekt übergeben wir der *paint*-Methode des *Pictures*, welches sich damit selbst auf das *Image* zeichnet.

Anschliessend werden wir mit einer der *write*-Methoden von *ImageIO* das *Image* speichern.

Es bleibt die Frage, wohin mit dem Code? Wir werden den FileManager erweitern, schließlich haben wir es hier mit einer Datei zu tun.

Wir müssen ansonsten keine großen Veränderungen machen (das Hauptmenü braucht noch einen zusätzlichen Eintrag), zusammengefasst sieht das so aus:

- Im *FileManager* benötigen wir eine Methode welche den Benutzer mit dem *JFileChooser* eine Datei wählen lässt, das *Picture* in ein *BufferedImage* zeichnet, und dieses neue *Image* anschließend mit der *ImageIO* in eine Datei umwandelt.
  public void export()
- Wir müssen dem Datei-Dialog die verschiedenen Bildformate vorgeben. Dazu erstellen wir einen Array von *SimpleFileFiltern*. Über *ImageIO* und die Methode *getWriterFormatNames* können

wir herausfinden, welche Bildtypen unterstützt werden.
private void prepareExportImage()

- Das Hauptmenü erhält einen neuen Eintrag „Exportieren“. Dieser Eintrag soll nur dann wählbar sein, wenn ein Bild vorhanden ist.

Versuch die Veränderungen zu implementieren, hier ist eine Lösung:

Veränderungen in FileManager

```
public class FileManager implements PictureListener{
    [...]

    /** Die verschiedenen Bildformate die schreibbar sind */
    private SimpleFileFilter[] imageFilters;
    /** Das zuletzt gewählte Bildformat */
    private SimpleFileFilter imageFilter;

    [...]

    /**
     * Speichert das Bild als Image (z.B. eine PNG-Datei) ab.
     */
    public void export(){
        prepareExportImage();

        while( true ){
            if( fileChooser.showSaveDialog( mainFrame ) ==
                  JFileChooser.APPROVE_OPTION ){
                File file = fileChooser.getSelectedFile();

                SimpleFileFilter filter =
                        (SimpleFileFilter)fileChooser.getFileFilter();
                if( !filter.accept( file ))
                    file = new File( file.getPath() + "." +
                        filter.getExtension() );

                if( file.exists() ){
                    int answer = JOptionPane.showConfirmDialog( mainFrame,
                            "Die Datei \"" + file.getName() +
                            "\" existiert bereits,\n"+
                            "soll sie überschrieben werden?",
                            "Datei überschreiben",
                            JOptionPane.YES_NO_CANCEL_OPTION,
                            JOptionPane.QUESTION_MESSAGE );

                    if( answer == JOptionPane.CANCEL_OPTION )
                        break;
                    else if( answer == JOptionPane.NO_OPTION )
                        continue; // Andere Datei anfordern
                }

                // Bild herstellen
                BufferedImage image = new BufferedImage( picture.getWidth(),
                        picture.getHeight(), BufferedImage.TYPE_INT_RGB );

                Graphics g = image.createGraphics();
                picture.paint( g );

                // Bild speichern
                try{
```

```
                    ImageIO.write( image, filter.getExtension(), file );
                }
                catch( IOException ex ){
                    JOptionPane.showMessageDialog( mainFrame,
                            "Ein Fehler trat beim Speichern auf:\n"
                            + ex.getMessage(), "Schreibfehler",
                            JOptionPane.ERROR_MESSAGE  );
                }
                break;
            }
            else
                break;
        }

        imageFilter = (SimpleFileFilter)fileChooser.getFileFilter();
    }


    /**
     * Präpariert den FileChooser so, dass ein Bild, wie ein JPEG, akzeptiert
     * wird.
     */
    private void prepareExportImage(){
        ensureFileChooserInit();
        emptyFileChooser();

        if( imageFilters == null ){
            String[] formats = ImageIO.getWriterFormatNames();
            int size = formats.length;

            imageFilters = new SimpleFileFilter[ size ];
            for( int i = 0; i < size; i++ )
                imageFilters[i] = new SimpleFileFilter( formats[i],
                        "Bild (*." + formats[i] + ")" );

            imageFilter = imageFilters[0];
        }

        for( FileFilter f : imageFilters )
            fileChooser.addChoosableFileFilter( f );

        fileChooser.setFileFilter( imageFilter );
        fileChooser.setAcceptAllFileFilterUsed( false );
    }
    [...]
}
```

Veränderungen in MainMenu

```
public class MainMenu extends JMenuBar implements ActionListener,
            PictureListener{
      [...]
      /** Dieses MenuItem erlaubt das Exportieren von Bildern */
      private JMenuItem itemExport;
      [...]
      public MainMenu( MainFrame frame, FigureFactory[] factories ){
            [...]
            itemExport = new JMenuItem( "Exportieren" );
            itemExport.setActionCommand( "export" );
            itemExport.addActionListener( this );
            itemExport.setEnabled( false );
            [...]
            file.add( itemExport );
            [...]
      }
```

```
        public void actionPerformed( ActionEvent e ){
              [...]
              else if( command.equals( "export" ))
                  fileManager.export();
        }
}
```

Eine JAR-Datei herstellen

Hiermit können wir unser Programm als fertig betrachten.

Wir müssen nun noch dafür sorgen, dass wir das Programm einfach an den Benutzer weitergeben können.

Dazu erstellen wir eine sogenannte JAR-Datei.
JAR-Dateien sind Archive, in welchen die *.class-Dateien gespeichert sind. Außerdem besitzen JAR's eine spezielle Datei namens „Manifest“, in welcher geschrieben steht, welche Klasse die Main-Class, also die Klasse mit dem *public static void main*, ist.

Manche IDE's können JAR's herstellen, aber es ist immer nützlich, wenn man auch den Querfeldeinweg kennt.

(Ich werde hier nur den Weg für Windows-Systeme beschreiben, alle Linux-Freaks und andere OS-Anhänger können aber in der API unter *Tool Docs* > *Basic Tools* > *jar* weitere Informationen finden)

Mache nun folgendes:

1. Erstelle im Ordner „tutorial“ eine Datei „`manifest.txt`“.
2. In diese Datei schreibst du „`Main-Class: tutorial.Main`“. Danach drückst du ENTER um eine neue Zeile einzufügen (dies ist wichtig, Neue Zeile ist Teil des Befehles).
3. Jetzt erstelle eine Datei „`jarSource.txt`“.
4. In diese neue Datei schreibst du:
   ```
   cmf0 manifest.txt Tutorial.jar
   -C bin
   ```
   cmf0 bedeutet „Neues Archiv erstellen“, „Manifest wird vorgegeben“, „Der Name der JAR-Datei wird vorgegeben“, „Keine Kompression verwenden“.
   In derselben Reihenfolge wie diese Flags erfolgt die Eingabe der Manifest-Datei und des Namens der neuen JAR-Datei.
   -c bin bedeutet, dass die *.class-Dateien in dem Verzeichnis `bin` sind. Es kann sein, dass du die class-Dateien in einem anderen Ordner hast, dann musst du `bin` entsprechend ersetzen.
5. Nun stelle eine Datei „`JAR create.bat`“ her.
6. In diese Datei schreibst du

   ```
   "c:\jdk1.5.0\bin\jar" @d:\dateien\java\tutorial\jarSource.txt
   pause
   ```

   Die erste Angabe ist der Pfad zu dem Programm jar.exe, welches die JAR-Datei herstellen wird.
   Die zweite Angabe zeigt auf die Datei, welche beschreibt, wie die JAR-Datei hergestellt werden soll.
   Danach bauen wir eine Pause ein, um Fehlermeldungen zu sehen, sollten welche auftreten.
7. Du kannst nun das Batch-Programm *JAR create.bat* mit einem Doppelklick ausführen lassen.

8. In dem Tutorial-Ordner wird dann eine neue Datei „*Tutorial.jar*“ erscheinen.
   Mache einen Doppelklick auf diese Datei: unser Programm startet.
9. Die *Tutorial.jar* ist lauffähig, ohne dass wir weitere Dateien mitliefern müssen, du kannst sie nun all deinen Freunden schicken (auch wenn sie ein anderes Betriebssystem verwenden), und unser Programm wird bei ihnen funktionieren (natürlich nur, wenn sie Java 5.0 installiert haben).

Herstellen einer Dokumentation

Dies ist die Fortsetzung zum Kapitel „Ein Wort zur Dokumentation“ ganz am Anfang.

Das Programm „javadoc“ erlaubt es uns, speziell vorbereitete Kommentare aus Java- in mehrere HTML-Dateien umzuwandeln.

Erstelle dazu eine neue Datei „DOC Create.bat“ in dem Ordner, in dem das Projekt lagert.

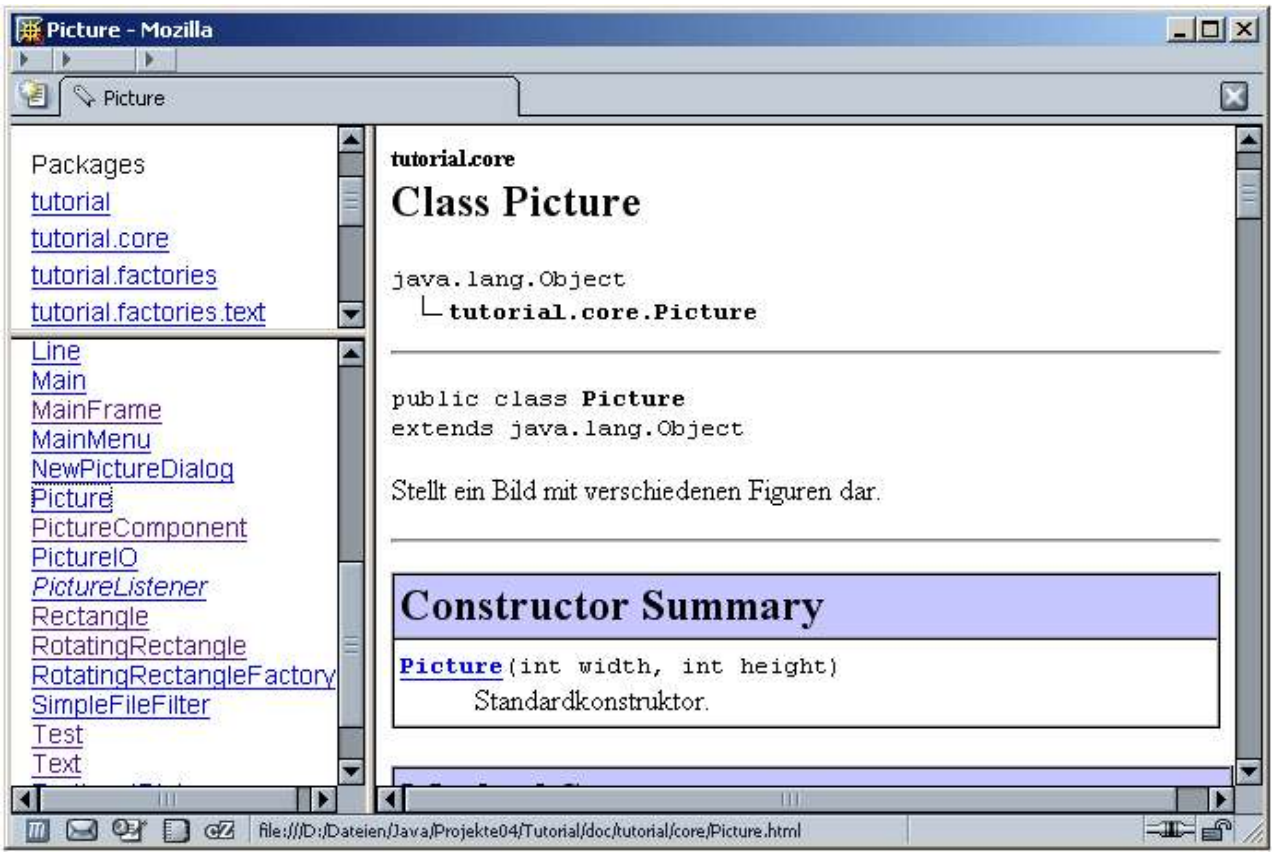
Öffne nun diese Datei, und schreib folgendes hinein:

```
"c:\program files\java\jdk1.5.0\bin\javadoc" -d
      "d:\dateien\java\projekte04\Tutorial\doc" -sourcepath
            "d:\dateien\java\projekte04\Tutorial\src" -subpackages tutorial

pause
```

Zwischen den ersten drei Zeilen ist übrigens *kein* „Neue Zeile“-Zeichen (ENTER) eingefügt!

Die erste Angabe ist die Position von Javadoc.
-d gibt das Ausgabeverzeichnis an, dort werden die HTML-Dateien hineingeschrieben.
-sourcepath Ist der Pfad zu dem Wurzelverzeichnis der *.java-Dateien
-subpackages Gibt an, dass alle Packages welche mit dem Namen `tutorial` beginnen dokumentiert werden sollen.

Die Pause am Schluss gibt uns die Möglichkeit allfällige Fehlermeldungen zu lesen.



Die generierten Dateien. Es sind mehr, als man erwartet.

Hacks und schmutzige Tricks

... gibt es auch in diesem Programm.

Guck dir mal die Reihenfolge der Befehle in der Methode *URManager.undo* und *MainMenu.setPicture*. Du siehst vielleicht (oder du kannst ein bisschen rumspielen in diesen beiden Methoden), dass der Enabled-Status des Menüeintrags für rückgängig und wiederholen, verrückt spielen kann, wenn wir die Reihenfolge nur leicht verändern.

Wir haben uns dadurch den *URManagerListener* gespart, sauberes Design war das allerdings nicht. Solche Hacks kann man sich in kleinen Programmen (wie diesem) erlauben, wenn man sich sicher sein kann, dass diese Verbindung nie mehr angerührt wird. In 99% aller Fälle wird eine solche Abkürzung aber früher oder später zu einem Problem führen... also gewöhne es dir nicht zu sehr an.

Neue Figuren

Ein Programm ist niemals fertig, man findet immer wieder etwas, was man noch hinzufügen könnte.

Und deshalb werden auch wir noch ein bisschen Spaß haben, und zwei weitere Figuren erfinden (Du kannst diese Kapitel auch überspringen, wir werden hier nur sehen, dass unser Progi noch mehr kann als die ganz einfachen Figuren zu zeichnen).

Das gedrehte Rechteck

Wir werden hier keine große Ableitungs- und Interface-Hierarchie aufbauen, auch wenn dies durchaus möglich wäre. Es geht uns jetzt aber nur darum eine weitere Figur herzustellen, welche sich von den anderen etwas unterscheidet.

Die Eingabe eines gedrehten Rechteckes benötigt mehr Schritte, als die Eingabe eines normalen Rechteckes. Der Benutzer soll das Rechteck folgendermaßen eingeben können:

1. Er drückt die Maus, damit beginnt die Herstellung
2. Er bewegt die gedrückte Maus um Breite und Höhe des Rechteckes zu setzen
3. Er lässt die Maustaste los, und signalisiert damit, dass er nun das Rechteck drehen möchte.
4. Mit dem Bewegen der Maus, ohne dass eine Taste gedrückt wäre, wird das Rechteck nun gedreht.
5. Der Benutzer drückt die Maus noch einmal, um die Figur zu verschieben.
6. Mit gedrückter Maustaste verschiebt er das Rechteck nun.
7. Er lässt die Maus los, und das Rechteck ist fertig eingegeben.

Eine Auflistung aller Methoden, welche wir schreiben müssen, entfällt. Wir müssen nur die Interfaces *FigureFactory* und *Figure* implementieren.

Nur noch ein Wort zur Mathematik: Die Drehung eines Punktes kann man mit Sinus und Cosinus darstellen, und mit Arcustangens kann man den Winkel zwischen einer Geraden und der x-Achse herausfinden.
In der JRE gibt es dafür die Klasse *java.lang.Math* mit den statischen Methoden *sin*, *cos* und *atan2*.

Falls Du jetzt keine Ahnung hast, wie das funktionieren soll: guck Dir meinen Quellcode an, und kopier die Methoden *paint* aus *RotatingRectangle* und *angle* aus *RotatingRectangleFactory*.
(Es geht in diesem Tutorial nicht um Mathematik, deshalb werde ich nicht genauer erklären, wie das funktioniert. Aber such Dir ein nettes Mathe-Buch (Stichwort Trigonometrie und auch Abbildungen), und versuch es selbst herauszufinden).

Schreib die beiden benötigten Klassen, hier ist meine Version:

RotatingRectangle

```
package tutorial.factories;

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

import tutorial.core.Figure;
import tutorial.core.FigureFactory;

/**
 * Diese Figure ist ein gefülltes Rechteck, welches um ein Zentrum
 * gedreht ist.
 */
public class RotatingRectangle implements Figure{
    /** Die Farbe der Figure */
    private Color color;
    /** Die Factory, welche die Figur hergestellt hat */
    private FigureFactory origin;

    /** Zentrum der Figur */
```

```
    private int centerX, centerY;
    /** Breite und Höhe der Figure */
    private int width, height;

    /** Der Winkel, um den Rotiert wird */
    private double rotation;

    /**
     * Standardkonstruktor
     * @param origin Die Factory, welche die Figur hergestellt hat
     */
    public RotatingRectangle( FigureFactory origin ){
        this.origin = origin;
    }

    /**
     * Standardkonstruktor
     * @param origin Die Factory, welche die Figur hergestellt hat
     * @param color Die Farbe der Figur
     */
    public RotatingRectangle( FigureFactory origin, Color color ){
        this( origin );
        setColor( color );
    }

    public void paint( Graphics g ) {
        g.setColor( color );

        // Die Rotation von Punkten kann man einfach mit
        // Sinus und Cosinus beschreiben

        int[] x = new int[4];
        int[] y = new int[4];

        double sin = Math.sin( rotation );
        double cos = Math.cos( rotation );

        x[0] = (int)( centerX + cos * width/2.0 - sin * height/2.0 );
        y[0] = (int)( centerY + sin * width/2.0 + cos * height/2.0 );

        x[1] = (int)( centerX + cos * width/2.0 + sin * height/2.0 );
        y[1] = (int)( centerY + sin * width/2.0 - cos * height/2.0 );

        x[2] = (int)( centerX - cos * width/2.0 + sin * height/2.0 );
        y[2] = (int)( centerY - sin * width/2.0 - cos * height/2.0 );

        x[3] = (int)( centerX - cos * width/2.0 - sin * height/2.0 );
        y[3] = (int)( centerY - sin * width/2.0 + cos * height/2.0 );

        g.fillPolygon( x, y, 4 );
    }

    public Color getColor() {
        return color;
    }

    public void setColor( Color color ) {
        this.color = color;
    }

    public FigureFactory getOrigin() {
        return origin;
    }
```

```
    public int getCenterX() {
        return centerX;
    }
    public void setCenterX( int centerX ) {
        this.centerX = centerX;
    }
    public int getCenterY() {
        return centerY;
    }
    public void setCenterY( int centerY ) {
        this.centerY = centerY;
    }
    public int getHeight() {
        return height;
    }
    public void setHeight( int height ) {
        this.height = height;
    }
    public double getRotation() {
        return rotation;
    }
    public void setRotation( double rotation ) {
        this.rotation = rotation;
    }
    public int getWidth() {
        return width;
    }
    public void setWidth( int width ) {
        this.width = width;
    }
    public void setOrigin( FigureFactory origin ) {
        this.origin = origin;
    }
}
```

RotatingRectangleFactory

```
package tutorial.factories;

import java.awt.Color;
import java.awt.Component;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.event.MouseEvent;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.util.Vector;

import javax.swing.Icon;

import tutorial.core.FactoryListener;
import tutorial.core.Figure;
import tutorial.core.FigureFactory;
import tutorial.core.Picture;

/**
 * Diese Factory erstellt ein {@link RotatingRectangle RotatingRectangle}.<br>
 * Der Benutzer arbeitet folgendermaßen:<br>
 * <ol>
 * <li>Er drückt die Maus, um mit der Bearbeitung zu beginnen</li>
 * <li>Er bewegt die Maus um die Größe zu bestimmen</li>
 * <li>Er lässt die Maus los, nun kann er die Drehung bestimmen</li>
 * <li>Er drückt die Maus erneut, um die Figur zu verschieben</li>
```

```
 * </ol>
 */
public class RotatingRectangleFactory implements FigureFactory{
    /** Angabe, dass derzeit nichts bearbeitet wird */
    private static final int STATUS_DO_NOTHING       = 0;
    /** Angabe, dass die Größe der Figur bearbeitet wird */
    private static final int STATUS_RESIZE_FIGURE    = 1;
    /** Angabe, dass die Drehung der Figur verändert wird */
    private static final int STATUS_ROTATE_FIGURE    = 2;
    /** Angabe, dass die Position der Figur bestimmt wird */
    private static final int STATUS_MOVE_FIGURE      = 3;

    /** Der aktuelle Arbeitsschritt */
    private int status = STATUS_DO_NOTHING;
    /** Die letzte Position der Maus */
    private Point mouse;
    /** Die Figur, die hergestellt wird */
    private RotatingRectangle rectangle;

    /** Die Farbe der neuen Figuren */
    private Color color;
    /** Das Bild, dem die neuen Figuren hinzugefügt werden sollen */
    private Picture picture;

    /** Die Liste aller Listener, welche diese Factory belauschen */
    private Vector<FactoryListener> listeners = new Vector<FactoryListener>();

    /**
     * Defaultkonstruktor
     */
    public RotatingRectangleFactory(){
    }

    public void paint( Graphics g ) {
        if( status != STATUS_DO_NOTHING ){
            // Falls eine Figur bearbeitet wird, soll sie gezeichnet werden
            rectangle.paint( g );

            if( status == STATUS_ROTATE_FIGURE ){
                // Falls die Figur gedreht wird, wird eine Linie zur Maus
                // gezeichnet
                g.setColor( new Color( 255 - color.getRed(),
                        255 - color.getGreen(),
                        255 - color.getBlue() ) );

                g.drawLine( mouse.x, mouse.y, rectangle.getCenterX(),
                        rectangle.getCenterY() );
            }
        }
    }

    public void setColor( Color color ) {
        this.color = color;
        if( rectangle != null ){
            rectangle.setColor( color );
            fireRepaint();
        }
    }

    public void setPicture( Picture picture ){
        this.picture = picture;
    }

    public void setActive( boolean active ){
        // Was auch immer passiert ist, jetzt wird von vorne begonnen
```

```
        status = STATUS_DO_NOTHING;
        rectangle = null;
    }

    public void write( Figure figure, DataOutputStream out )throws IOException {
        RotatingRectangle rect = (RotatingRectangle)figure;

        out.writeInt( rect.getCenterX() );
        out.writeInt( rect.getCenterY() );
        out.writeInt( rect.getWidth() );
        out.writeInt( rect.getHeight() );
        out.writeDouble( rect.getRotation() );
    }

    public Figure read( DataInputStream in ) throws IOException {
        RotatingRectangle rect = new RotatingRectangle( this );

        rect.setCenterX( in.readInt() );
        rect.setCenterY( in.readInt() );
        rect.setWidth( in.readInt() );
        rect.setHeight( in.readInt() );
        rect.setRotation( in.readDouble() );

        return rect;
    }

    public Icon getIcon() {
        return new Icon(){
            public int getIconHeight() {
                return 16;
            }
            public int getIconWidth() {
                return 16;
            }
            public void paintIcon( Component c, Graphics g, int x, int y ) {
                g.setColor( c.getForeground() );

                int[] px = new int[]{ x+3, x+16, x+13, x };
                int[] py = new int[]{ y, y+3, y+16, y+13 };

                g.fillPolygon( px, py, 4 );
            }
        };
    }

    public String getName() {
        return "Gedrehtes Rechteck";
    }

    public String getID() {
        return "FILLED_ROTATED_RECTANGLE";
    }

    public void addFactoryListener( FactoryListener listener ) {
        listeners.add( listener );
    }

    public void removeFactoryListener( FactoryListener listener ) {
        listeners.remove( listener );
    }

    protected void fireRepaint(){
        for( FactoryListener f : listeners )
            f.shouldRepaint( this );
    }
```

```
    public void mouseClicked( MouseEvent e ) {
    }

    public void mousePressed( MouseEvent e ) {
        if( status == STATUS_DO_NOTHING ){
            // Falls noch nichts getan wird: neue Figur erstellen,
            // und Größe setzen
            rectangle = new RotatingRectangle( this, color );
            rectangle.setCenterX( e.getX() );
            rectangle.setCenterY( e.getY() );
            status = STATUS_RESIZE_FIGURE;
            mouse = e.getPoint();

            fireRepaint();
        }
        else if( status == STATUS_ROTATE_FIGURE ){
            // Vom Status "Figur drehen" in den Status
            // "Figur verschieben" wechseln
            status = STATUS_MOVE_FIGURE;
            mouse = e.getPoint();
            fireRepaint();
        }
    }

    public void mouseReleased( MouseEvent e ) {
        if( status == STATUS_RESIZE_FIGURE ){
            // In den Status "Figur drehen" wechseln
            status = STATUS_ROTATE_FIGURE;
            mouseMoved( e );
        }
        else if( status == STATUS_MOVE_FIGURE ){
            // Die Bearbeitung beenden, und die Figur dem Bild hinzufügen
            status = STATUS_DO_NOTHING;
            picture.add( rectangle );
            rectangle = null;
            fireRepaint();
        }
    }

    public void mouseDragged( MouseEvent e ) {
        if( status == STATUS_RESIZE_FIGURE ){
            // Die Größe der Figur verändern
            mouse = e.getPoint();
            int dx = Math.abs( e.getX() - rectangle.getCenterX() );
            int dy = Math.abs( e.getY() - rectangle.getCenterY() );

            rectangle.setWidth( dx * 2 );
            rectangle.setHeight( dy * 2 );

            fireRepaint();
        }
        else if( status == STATUS_MOVE_FIGURE ){
            // Die Position der Figur verändern
            int dx = rectangle.getCenterX() - mouse.x;
            int dy = rectangle.getCenterY() - mouse.y;

            mouse = e.getPoint();

            rectangle.setCenterX( e.getX() + dx );
            rectangle.setCenterY( e.getY() + dy );

            fireRepaint();
        }
    }
```

```
    public void mouseMoved( MouseEvent e ) {
        if( status == STATUS_ROTATE_FIGURE ){
            //Die Drehung der Figur verändern
            mouse = e.getPoint();
            rectangle.setRotation( angle( rectangle.getCenterX(),
                    rectangle.getCenterY(), e.getX(), e.getY(),
                    e.isShiftDown() ));
            fireRepaint();
        }
    }

    /**
     * Berechnet den Winkel den die Figur haben sollte.
     * @param centerX Die x-Koordinate des Zentrums der Figur
     * @param centerY Die y-Koordinate des Zentrums der Figur
     * @param mouseX Die x-Position der Maus
     * @param mouseY Die y-Position der Maus
     * @param shift Angabe, ob die shift-Taste gedrückt ist. Falls hier
     * <code>true</code> übergeben wird, wird der Winkel auf ein
     * Vielfaches von 45° gerundet.
     * @return Der Winkel, ein Wert zwischen -PI und +Pi
     */
    protected double angle( int centerX, int centerY, int mouseX, int mouseY,
            boolean shift ){
        int dx = mouseX - centerX;
        int dy = mouseY - centerY;

        double angle = 0;

        if( dx != 0 || dy != 0 ){
            angle = Math.atan2( dy, dx );
        }

        if( shift ){
            // angle auf die nächste Zahl, welche ein
            // Vielfaches von PI/4 ist runden

            double mul = angle / (Math.PI / 4.0);
            mul = Math.round( mul );
            angle = mul * (Math.PI / 4.0);
        }

        return angle;
    }

    public void mouseEntered( MouseEvent e ) {}
    public void mouseExited( MouseEvent e ) {}
}
```

Text

Zu jedem Zeichenprogramm gehört auch eine Möglichkeit Text zu schreiben, darum werden wir zum Schluss noch eine Text-Factory schreiben. Dazu überlegen wir uns wieder, wie oben, wie der Ablauf aussehen soll:

1. Der Benutzer klickt einmal um den Dialog zu öffnen in dem er seinen Text eingeben und formatieren kann.
2. Die Benutzer gibt den Text ein und formatiert ihn. Dann soll er auf OK klicken (oder auf Abbrechen.)
3. Der Text wird an der richtigen Stelle gemalt.

Zum Anfang schreiben wir unsere Figur, die eine innere Klasse von unserer Factory ist. Das machen wir so und haben wir bei den anderen Beispielen auch so gemacht, da niemand anderen, außer der Factory, angeht was die *Figure*, die die Factory zurück liefert, überhaupt ist. Hauptsache sie implementiert die entsprechenden Methoden.

```
private class TextFigure extends AbstractFigure {

	private String text = "";
	private Font font = new Font("Dialog", Font.PLAIN, 12);


	public void setText(String text) {
		this.text = text;
	}

	public String getText() {
		return text;
	}

	public void setFont(Font font) {
		this.font = font;
	}

	public Font getFont() {
		return font;
	}

	public void paint( Graphics g ) {
		g.setColor( color );
		g.setFont(font);
		String[] lines = text.split("\n");
		for(int i=0; i<lines.length; i++) {
			g.drawString(lines[i], x, y+((i+1)*g.getFontMetrics().
                                                        getAscent()));
		}
	}

	public FigureFactory getOrigin() {
		return Text.this;
	}
}
```

Die getter- und setter-Methoden sollten selbsterklärend sein. In der *paint*() Methode müssen wir zuerst unseren Text in die einzelnen Zeilen zerlegen und dann jede Zeile einzeln malen, da die Methode *Graphics#drawString*() Zeilenumbrüche nicht mit zeichnet.

Als nächstes wollen wir unseren Dialog bauen. Wir schreiben dazu eine eigene Klasse und leiten sie von *JDialog* ab. Wiederum überlegen wir, was der Dialog können soll:

- Man soll Schriftart -Farbe und -Größe einstellen können
- Man soll den Text eingeben können, sodass er automatisch upgedatet wird, wenn Änderungen an der Formatierung vorgenommen werden.
- Der Dialog soll Modal sein, sodass man nicht vom Dialog zum Hauptprogramm wechseln kann. Außerdem ist das nützlich, da wir so nach dem Schließen des Dialoges einfach auf die eingestellten Schriftformatierungen und den Text zugreifen können.

Beim folgendem Code gibt es ein paar neue Dinge und Sachen zu beachten:

Der Konstruktor von *JDialog* erwartet ein *JFrame*. Da wir aber beim erstellen der Factory keines besitzen, warten wir bis der Dialog das erste mal benötigt wird. Mit den *SwingUtilities* und dem

Wissen, auf welcher *Component* die Maus gedrückt wurde, können wir das *MainFrame* finden, und übergeben.
Es wäre auch möglich den Superkonstruktor mit `super((javax.swing.JFrame) null, "Texteingabe", true);` aufzurufen, der Dialog würde trotzdem funktionieren. Aber man kann nie wissen, ob solch ein Trick auch in Zukunft, und bei allen Betriebssystemen funktionieren wird...

Mit: *GraphicsEnvironment#getAvailableFontFamilyNames*() bekommen wir alle auf dem System verfügbaren Schriftarten in einem String Array zurückgeliefert.

Außerdem habe ich 3 Hilfsmethoden implementiert, die mir die eingestellte Größe und Formatierung zurück liefert.

getFont() und *getText*() werden von unserer Factory benutzt um die Eingaben zu bekommen.

```
package tutorial.factories.text;

import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.Insets;
import java.awt.GraphicsEnvironment;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;

import javax.swing.JDialog;
import javax.swing.JTextArea;
import javax.swing.JScrollPane;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JSpinner;
import javax.swing.SpinnerNumberModel;
import javax.swing.SwingUtilities;
import javax.swing.JPanel;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;


/**
 * Unsere Klasse, die dem Benutzer ermöglicht Text einzugeben und zu
formatieren.<br>
 * Die Klasse implementiert 3 Listener:<br>
 * <ul>
 * <li>ActionListener - um auf die Events zu reagieren die von den Buttons
ausgehen</li>
 * <li>ItemListener - um auf Änderungen der JComboBoxes zu reagieren</li>
 * <li>ChangeListener - ebenso für den JSpinner</li>
 * </ul>
 */
public class TextInputDialog extends JDialog implements ActionListener,
                                                         ItemListener,
                                                         ChangeListener {

      /** Der Text */
      private String text = "";

      /** Die Schriftart + Formatierung */
```

```
    private Font font = new Font("Dialog", Font.PLAIN, 12);

    /* Unsere Komponenten */
    private JTextArea textarea = new JTextArea(10, 20);
    private JComboBox fonts;
    private JComboBox formats;
    private JSpinner size;
    private JButton okButton = new JButton("OK");
    private JButton cancelButton = new JButton("Abbrechen");


    public TextInputDialog( Component parent ) {
          // Wir müssen zuerst den Superkonstruktor aufrufen.
          // mit einem kleinen Trick benötigen wir kein parent Fenster
          super((JFrame) SwingUtilities.getRoot(parent), "Texteingabe", true);

          // unsere Komponenten zur Schriftart Auswahl
          // so kriegen wir alle verfügbaren Schriftarten auf dem System als
          // string array
          String[] fontNames = GraphicsEnvironment.getLocalGraphicsEnvironment
                                                ().getAvailableFontFamilyNames();

          fonts = new JComboBox(fontNames);
          fonts.addItemListener(this);

          String[] formatTypes = new String[]{"Normal", "Fett", "Kursiv",
                "Fett & Kursiv"};
          formats = new JComboBox(formatTypes);
          formats.addItemListener(this);

          // wie man einen JSpinner erstellt haben wir ja schon oben gesehen.
          size = new JSpinner(new SpinnerNumberModel(12, 2, 50, 1));
          size.addChangeListener(this);

          JPanel fontFormatPanel = new JPanel(new GridBagLayout());
          fontFormatPanel.add(fonts, new GridBagConstraints( 0, 0, 2, 1, 10.0,
                1.0, GridBagConstraints.NORTH,GridBagConstraints.HORIZONTAL,
                new Insets( 2, 2, 2, 2 ), 0, 0 ));
          fontFormatPanel.add(formats, new GridBagConstraints( 0, 1, 1, 1,6.0,
                1.0, GridBagConstraints.NORTH,GridBagConstraints.HORIZONTAL,
                new Insets( 2, 2, 2, 2 ), 0, 0 ));
          fontFormatPanel.add(size, new GridBagConstraints( 1, 1, 1, 1, 4.0,
                1.0,GridBagConstraints.NORTH, GridBagConstraints.BOTH,
                new Insets( 2, 2, 2, 2 ), 0, 0 ));

          textarea.setFont(font);
          JScrollPane scrp = new JScrollPane(textarea);

          okButton.addActionListener(this);
          cancelButton.addActionListener(this);

          JPanel buttons = new JPanel(new FlowLayout(FlowLayout.RIGHT));
          buttons.add(okButton);
          buttons.add(cancelButton);

          // alles hinzufügen
          add(fontFormatPanel, BorderLayout.NORTH);
          add(scrp, BorderLayout.CENTER);
          add(buttons, BorderLayout.SOUTH);
          pack();
          setLocationRelativeTo(null);
    }

    public String getText() {
```

```
            String ret = text;
            setText("");
            return ret;
      }

      public void setText(String text) {
            this.text = text;
            textarea.setText(text);
      }

      public Font getFont() {
            return font;
      }

      public void actionPerformed(ActionEvent e) {
            Object s = e.getSource();
            if(s == okButton) {
                  text = textarea.getText();
            }
            setVisible(false);
      }

      public void stateChanged(ChangeEvent e) {
            updateFont();
      }

      public void itemStateChanged(ItemEvent e) {
            updateFont();
      }

      /**
       * Diese Methode wird aufgerufen wenn in den Komponenten zur Schriftart
       * Formatierung ein anderer Wert eingegeben wurde.
       */
      protected void updateFont() {
            font = new Font((String) fonts.getSelectedItem(), getFontStyle(),
                                                                  getFontSize());
            textarea.setFont(font);
      }

      protected int getFontStyle() {
            String style = (String) formats.getSelectedItem();
            if("Normal".equals(style)) {
                  return Font.PLAIN;
            } else if("Kursiv".equals(style)) {
                  return Font.ITALIC;
            } else if("Fett".equals(style)) {
                  return Font.BOLD;
            } else {
                  return Font.BOLD | Font.ITALIC;
            }
      }

      protected int getFontSize() {
            return ((Integer) size.getValue()).intValue();
      }
}
```

Noch eine kleine Anmerkung: Dir ist sicher aufgefallen dass ich anstatt `style.equals("Normal")` `"Normal".equals(style)` geschrieben hab. Das ist hier zwar nicht unbedingt notwendig, aber wenn irgendwo mal der Fall eintreten sollte, dass die zu vergleichende Variable `null` ist, wird keine *NullPointerException* geworfen, sondern einfach `false` zurückgegeben.

Als Letzes wollen wir noch unsere *FigureFactory* implementieren. Das wird nicht weiter schwierig

sein, da nicht viel neuer Code hinzukommt und sich der Rest aus den Texten über dem ergibt.

```
package tutorial.factories;

import java.awt.Component;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

import javax.swing.Icon;

import tutorial.core.AbstractFigure;
import tutorial.core.AbstractFigureFactory;
import tutorial.core.Figure;
import tutorial.core.FigureFactory;
import tutorial.factories.text.TextInputDialog;


public class Text extends AbstractFigureFactory {

	protected TextInputDialog dialog;


	public AbstractFigure createFigure() {
		return new TextFigure();
	}

	public void write( Figure figure, DataOutputStream out ) throws
                                                                IOException {
		super.write(figure, out);
		TextFigure tf = (TextFigure) figure;
		out.writeUTF(tf.getText());
		Font f = tf.getFont();
		out.writeUTF(f.getName());
		out.writeInt(f.getStyle());
		out.writeInt(f.getSize());
	}

	public Figure read( DataInputStream in ) throws IOException {
		Figure af = super.read(in);
		TextFigure tf = (TextFigure) af;
		tf.setText(in.readUTF());
		String fName = in.readUTF();
		int fStyle = in.readInt();
		int fSize = in.readInt();
		Font f = new Font(fName, fStyle, fSize);
		tf.setFont(f);
	    return tf;
	}

	public String getID() {
		return "TEXT";
	}

	public Icon getIcon() {
	    return new Icon(){
	        public int getIconHeight() {
	            return 16;
	        }
	        public int getIconWidth() {
```

```
                return 16;
            }
            public void paintIcon( Component c, Graphics g, int x, int y ) {
                g.setColor( c.getForeground() );
                g.drawString("T", x+8-g.getFontMetrics().stringWidth("T")/2,
                                                                    y+13);
            }
        };
    }

    public String getName() {
        return "Text";
    }

    public void mouseClicked( MouseEvent e ) {
          TextFigure tf = new TextFigure();
          tf.setX( e.getX() );
          tf.setY( e.getY() );
          tf.setColor(color);

          if( dialog == null )
                dialog = new TextInputDialog( e.getComponent() );

          dialog.setVisible(true);
          tf.setText(dialog.getText());
          tf.setFont(dialog.getFont());

          picture.add( tf );
    }

    // Hier noch der Code von der TextFigure

}
```

Wie geht es weiter?

Es muss dir immer klar sein, dass wir nur ein kleines Programm geschrieben haben.

Gerade wenn es um die graphische Oberfläche ging, haben wir manche Abkürzung genommen, die man sich in einem großen Programm nicht erlauben kann, weil es doch schnell einmal unübersichtlich werden kann.

Man kann aber dennoch von einem sauberen Programm sprechen: wir können zu jeder Klasse, zu jeder Methode, ja sogar zu jeder Variable einen genauen Verwendungszweck angeben. Alles ist thematisch sortiert, und das Programm bietet durchaus Möglichkeiten, weitere Features einzubauen.

Dir, mein treuer Leser, empfehle ich nun, selbst ein Programm zu schreiben. Dieses Programm soll nicht grösser sein, als das was wir eben gemacht haben; aber es soll ohne fremde Hilfe gebaut sein.

Anhang:

Was ist...

... ein Interface?

Ein Interface (zu deutsch, eine *Schnittstelle*) ist nicht mehr als eine Liste von Methoden, die von einer Klasse implementiert werden müssen.
Nebst der Tatsache, dass man in einem Interface keinen einzigen Befehl (wie etwa „int a = b+c;“) schreiben kann, ist einer der wichtigen Unterschiede, dass eine Klasse *mehrere* Interfaces implementieren kann, jedoch nur von *einer* Klasse erbt.

Betrachten wir mal ein Beispiel:

```
public interface Fahrzeug{
    /**
     * Gibt die Anzahl Achsen zurück, welche dieses Fahrzeug besitzt.
     * @return Die Anzahl Achsen
     */
    public int getAnzahlAchsen();
}

// ************************************************************

public interface Putzgeraet{
    /**
     * Gibt die Anzahl Quadratmeter zurück, welche mit diesem
     * Gerät in 1 Minute gereinigt werden können.
     * @return Gereinigte Quadratmeter pro Minute
     */
    public int getQuadratMeter();
}

// ************************************************************

public interface Stromverbraucher{
    /**
     * Gibt die benötigte Betriebsspannung zurück, welche von
     * diesem Verbraucher benötigt wird.
     * @return Die Spannung in Volt
     */
    public int getBetriebsspannung();
}

// ************************************************************

public class Lokomotive implements Fahrzeug, Stromverbraucher{
    public int getAnzahlAchsen(){
        return 6;
    }

    public int getBetriebsspannung(){
        return 1500;
    }
}

// ************************************************************

public class Auto implements Fahrzeug{
    public int getAnzahlAchsen(){
        return 2;
```

```
    }
}

// ************************************************************

public class Staubsauger implements Putzgeraet, Stromverbraucher{
    public int getQuadratMeter() {
        return 20;
    }

    public int getBetriebsspannung() {
        return 230;
    }
}
```

Uns interessieren hier verschiedene Größen, z.B. wie viele Achsen ein Fahrzeug hat, wie viele Quadratmeter in einer Minute gesäubert werden könne, wie viel Strom eine Maschine braucht... Für diese Informationen schreiben wir Interfaces. Später können wir irgendwelche Klassen diese Interfaces implementieren lassen, und sie so zu *Fahrzeugen* oder anderem werden lassen.

Benutzen können wir das Ganze dann so:

```
// Ein Vector herstellen, der nur Instanzen vom Typ
// „Fahrzeug“ aufnehmen kann.
Vector<Fahrzeug> fahrzeuge = new Vector<Fahrzeug>();

// Eine Lokomotive ist ein Fahrzeug, und wird akzeptiert.
fahrzeuge.add( new Lokomotive() );

// Auch ein Auto ist ein Fahrzeug
fahrzeuge.add( new Auto() );

// Ein Staubsauger ist kein Fahrzeug, dies wird vom Compiler
// nicht akzeptiert.
fahrzeuge.add( new Staubsauger() );

// Interfaces selbst kann man nicht initialisieren (was
// würde dieses Objekt wohl bei der Methode „getAnzahlAchsen()“
// zurückgeben?). Deshalb gibt auch dies ein Compilerfehler.
fahrzeuge.add( new Fahrzeug() );
```

... eine abstrakte Klasse?

Eine abstrakte Klasse hat folgende Eigenschaften:

- Man kann sie nicht instantiieren.
- Sie müssen nicht alle Methoden eines implementierten Interfaces implementieren.
- Sie können Methoden definieren, die Unterklassen implementieren müssen.

Ein gutes Beispiel für eine abstrakte Klasse ist eine Klasse *Mensch*. Die beiden Klassen *Mann* und *Frau* erben beide von *Mensch*. Da vieles bei Mann und Frau gemeinsam ist, können wir diese Funktionen bereits in der Klasse *Mensch* implementiere. Andere Dinge muss man dann separat für beide Klassen implementieren.
Hier ein Pseudo-Beispielcode:

```
package natur.lebewesen;

import natur.nahrung.Essen;
```

```java
public abstract class Mensch extends Säugetier {

	public void issWas(Essen watZuFuttern) {
		System.out.println("Hmm, lecker. Danke");
	}

	public abstract Mensch macheKind(Mensch m);
}
```

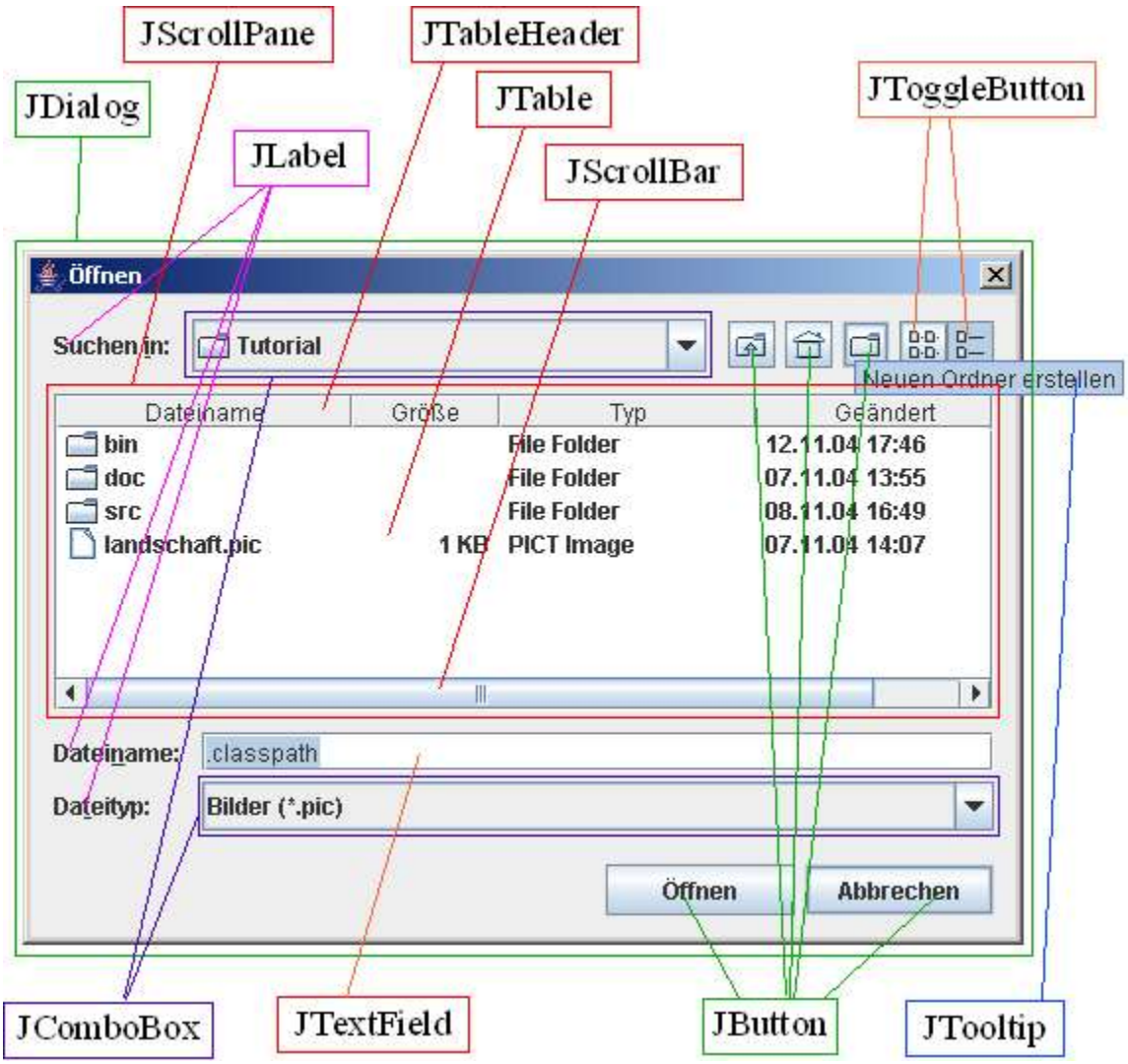
```java
package natur.lebewesen;

public abstract class Mann extends Mensch {

	public Mensch macheKind(Mensch m) {
		if(m instanceof Mann)
			throw new IllegalArgumentException("Mit mir nicht!");
		Mensch ergebnis;
		// Der Rest wurde ausgelassen :-P
		return ergebnis;
	}

}
```

... Swing?

Swing ist eine Sammlung von Klassen, welche verschiedene graphische Oberflächenelemente darstellen, und mehreren, zum Teil ziemlich aufwendigen, Hilfsklassen und Interfaces.

Swing ist eigentlich eine Weiterentwicklung von AWT (Abstract Window Toolkit), und so können viele Teile des AWT's auch in Swing verwendet werden, ein Beispiel sind die LayoutManager.

Hier sind ein paar Beispiele von verschiedenen Componenten, welche in Swing vorkommen:



Wie können diese Components aufeinander setzen, indem wir die Methode *add* verwenden.

Um die Position der Components relativ zueinander zu bestimmen, verwenden wir die *LayoutManager* aus dem AWT. Ein ganz einfaches Fenster mit 2 Knöpfen sieht so aus:

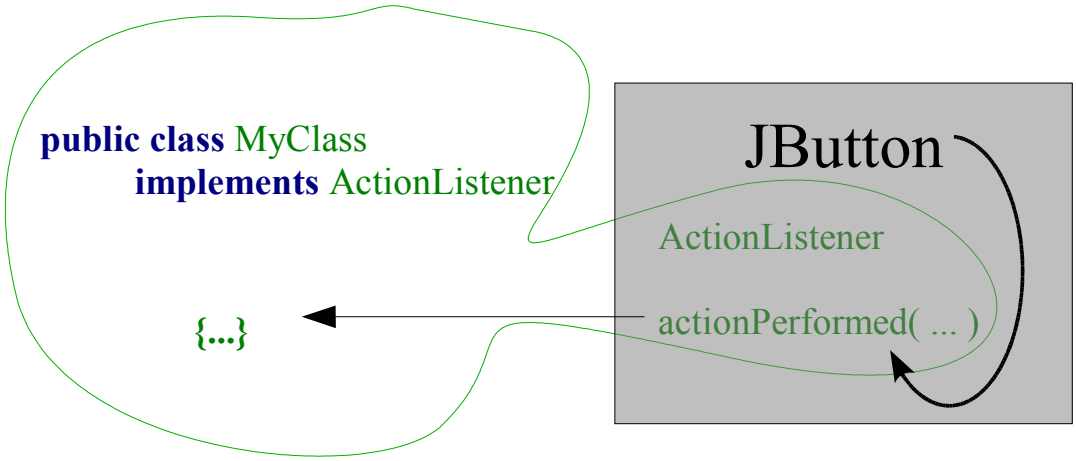
```
import java.awt.Container;
import java.awt.GridLayout;

import javax.swing.JButton;
import javax.swing.JFrame;

public class Test{
      public static void main( String[] args ){
            // Ein Frame erstellen
            JFrame frame = new JFrame( "Test" );

            // Was auf dem Frame sichtbar sein soll, muss auf das
            // ContentPane gesetzt werden.
            Container content = frame.getContentPane();

            // Die Position und Größe der Components wird durch
            // den LayoutManager bestimmt
            content.setLayout( new GridLayout( 2, 1 ));
```

```
            // Zwei Buttons erstellen, und dem ContentPane hinzufügen
            content.add( new JButton( "Hallo" ));
            content.add( new JButton( "Tschau" ));

            // Wenn das Fenster geschlossen wird, soll das Programm
            // beendet werden
            frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

            // Die optimale Größe des Frames berechnen
            frame.pack();

            // Das Frame in die Mitte des Bildschirmes platzieren
            frame.setLocationRelativeTo( null );

            // Das Frame sichtbar machen
            frame.setVisible( true );
        }
}
```

Natürlich benötigen wir eine Möglichkeit auf Ereignisse wie „Button gedrückt“ oder „Maus gedrückt“.

Zu diesem Zweck gibt es die Listener. Listener sind Interfaces, mit einer oder mehreren Methoden. Die Implementation eines Listeners wird einer Component übergeben (nicht alle Components nehmen alle Listener auf. Ein Button der einen TextListener aufnimmt, würde wohl auch nicht allzu viel Sinn ergeben), welche, wann immer etwas interessantes passiert, die Methode(n) des Listeners aufruft.

Das folgende Bild kann vielleicht helfen, das Prinzip von Listenern zu verstehen:
Wir haben eine Klasse *MyClass* geschrieben. Eine unbekannte Methode dieser Klasse soll irgendetwas tun, wenn der Benutzer einen Button drückt.
Wir implementieren nun einen *ActionListener*, und fügen eine Instanz von *MyClass* dem Button hinzu, auf welchen wir reagieren wollen.
Der *ActionListener* bildet eine Art Auswuchs, und der *JButton* weiß genau, was er mit diesem Auswuchs machen soll: die Methode *actionPerformed(ActionEvent e)* aufrufen.



Oder ein anderes Beispiel, die Hintergrundfarbe eines Fenster soll verändert werden, wenn der Benutzer auf eine bestimmte Component klickt (mit der Maus).

Dann fügen wir dieser Component einen *MouseListener* hinzu, und reagieren, wenn *mousePressed* und *mouseReleased* aufgerufen wird.

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

import javax.swing.JComponent;
import javax.swing.JFrame;

// Diese Klasse Test ist eine Component, welche das Interface MouseListener
// implementiert
public class Test extends JComponent implements MouseListener{
    private boolean pressed = false;

    public Test(){
        // Die Klasse registriert sich selbst als MouseListener
        addMouseListener( this );
    }

    // Diese Methode wird aufgerufen, wenn die Maus gedrückt wird.
    public void mousePressed( MouseEvent e ) {
        pressed = true;
        repaint(); // Diese Component neu zeichnen
    }

    // Diese Methode wird aufgerufen, wenn die Maus losgelassen wird.
    public void mouseReleased( MouseEvent e ) {
        pressed = false;
        repaint();
    }

    // Diese Methode wird von der Component aufgerufen, und zeichnet die
    // Oberfläche der Component.
    protected void paintComponent( Graphics g ) {
        // Falls die Maus gedrückt ist, zeichnen wir mit roter Farbe,
        // ansonsten mit Grün.
        if( pressed )
            g.setColor( Color.RED );
        else
            g.setColor( Color.GREEN );

        // Hier zeichnen wir ein gefülltes Rechteck
        g.fillRect( 0, 0, getWidth(), getHeight() );
    }

    public void mouseClicked( MouseEvent e ) {}
    public void mouseEntered( MouseEvent e ) {}
    public void mouseExited( MouseEvent e ) {}

    public static void main( String[] args ){
        // Zusammenbauen eines Frames, um die Test-Component anzuzeigen
        JFrame frame = new JFrame();
        frame.add( new Test() ); // dieses „add" funktioniert erst ab Java 5.0!
        frame.setBounds( 20, 20, 500, 500 );
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setVisible( true );
    }
}
```

... ein Input-/ OutputStream?

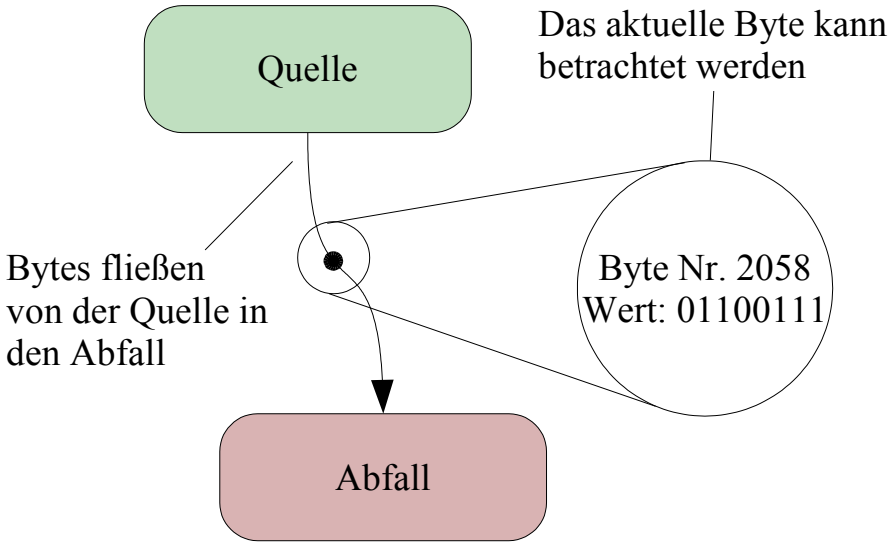
Daten, wie wir sie verwenden, sind eine Folge von Bytes.

Ein Input-/OutputStream ist nun ein Strom von Bytes. Wir können immer nur ein Byte betrachten,

dann schwimmt es wieder davon, und das nächste Byte kommt.

Streams unterscheiden sich in der Richtung, InputStreams lesen Daten, OutputStreams schreiben sie.

Ein InputStream kann man sich so vorstellen:



Der zweite Unterschied von Streams sind die Art der Quelle (bzw. des Ziels). So kann ein Stream als Quelle ein Byte-Array haben (*ByteArrayInputStream*), oder eine Datei (*FileInputStream*), von Außen sieht man keinen Unterschied.

Und dann gibt es noch Streams, welche die Daten gleich interpretieren, der *DataInputStream* kann z.B. 4 Bytes als einen Integer lesen. Der *DataOutputStream* ist das Gegenstück, wir können ihm einen Integer übergeben, und er wird 4 Bytes schreiben.

... so böse an static?

Static erlaubt es globale Variablen zu definieren. Man kann auf diese Variablen von überall aus dem Programm zugreifen, was das Schreiben eines Programmes natürlich einfacher macht.

Es ist möglich, dass Du dich gefragt hast, weshalb wir in diesem Programm praktisch nie „static“ verwendet haben.

Static hat durchaus eine Existenzberechtigung, aber es wird oft falsch verwendet. Viele Programmierer hacken einfach ein „static“ hin, wenn ihnen keine bessere Lösung für irgendwelche fehlenden Variablen einfällt.

Static sollte für einmalige Werte/Objekte benutzt werden, z.B. eine Repräsentation der Tastatur.

Es stellt sich die Frage, gibt es in unserem Programm nur ein Hauptfenster? Die Antwort ist „nein“, nichts hindert uns daran, mehr als ein Hauptfenster zu öffnen. Wenn wir das Hauptfenster nun irgendwo „static“ verwenden, wird unser Progi ganz schön Probleme bekommen, sobald das zweite Fenster offen ist...

Copyright

Im folgenden bezeichnet „dieses Tutorial“ sowohl dieses Dokument, als auch das mitgelieferte Übungsprogramm.

Das Copyright dieses Tutorials, liegt bei Benjamin Sigg und Janni Kovacs.

benjamin_sigg@gmx.ch

jannikovacs@roar-software.net

Die kommerzielle Nutzung dieses Tutorials, auch nur Teilen davon, ist untersagt.

Es ist erlaubt Kopien für persönliche Zwecke anzulegen, und auch weiterzureichen. Es ist jedoch nicht erlaubt, dieses Tutorial (oder eine Kopie davon) verändert weiterzugeben.