Bitte geben Sie Aufgabe37.java und BildPanel.java bis zum 07.02.2021 ab!
Dabei müssen Sie nicht die komplette Aufgabe lösen, es reichen die Convolution-Filter

Aufgabe 37

Bildbearbeitung – ein Hauch von Photoshop

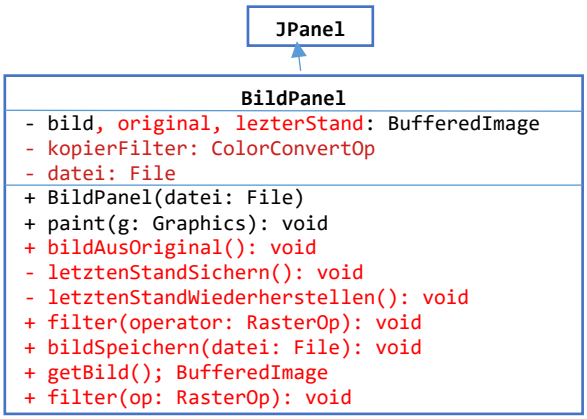
Wir werden in dieser Aufgabe unser Programm zur Bildbetrachtung aus Aufgabe 32 Schritt für Schritt um folgende Funktionen erweitern:

- verschiedene Filter auf ein Bild anwenden,
- den letzten Schritt rückgängig machen,
- das Original Bild wiederherstellen,
- Bilder speichern.

So dass wir am Ende eine kleine Anwendung zur Bildbearbeitung haben.

Vorbereitung zum Original-Bild wiederherstellen und letzten Schritt rückgängig machen

Zunächst erweitern wir die Klasse ***BildPanel*** wie im UML Diagramm gezeigt. Die Variable ***kopierFilter*** wir mit *ColorConvertOp(null)* initialisiert.



Im **Konstruktor** speichert seinen Parameter *datei* zunächst in der entsprechenden Instanzvariable. Dann lesen wir das Bild in die Variable *originalBild* ein, um jederzeit das Original wiederherstellen zu können. Anschließend werden in ***bild*** und ***letzterStand*** zwei neue Objekte vom Typ *BufferedImage* gespeichert, welche die gleiche Größe wie *OriginalBild* haben, aber vom Typ TYPE_INT_RGB sind. Zum Schluss wird *bildAusOriginal* gerufen.

Die Methode ***bildAusOriginal*** ruft *letztenStandSichern*, dann wird das Original nach *bild* kopiert:

```
kopierFilter.filter(original, bild);
```

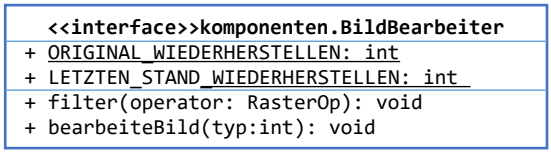
und zum Schluss stellt sie durch *repaint()* das Bild neu dar.

In der Methode ***letztenStandSichern*** wird - falls *bild* nicht *null* ist – *bild* mit Hilfe von *kopierFilter* nach *letzterStand* kopiert.

Die Methode ***letztenStandWiederherstellen*** kopiert - falls *letzterStand* nicht *null* ist – *letzterStand* mit *kopierFilter* nach *bild* und ruft *repaint*.

Die Klasse kann jetzt das Original-Bild wieder herstellen und die letzte Operation rückgängig machen.

Als nächstes schreiben wir das im UML Diagramm gezeigte Interface ***BildBearbeiter***. Die *final* deklarierten Klassenvariablen bekommen beliebige Werte.

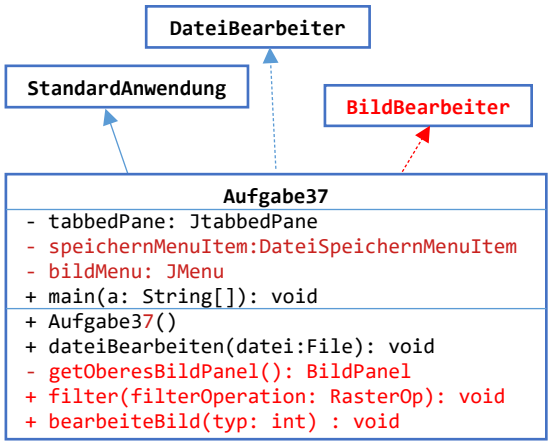


Dann erstellen Sie eine Kopie von *Aufgabe32*, nennen sie ***Aufgabe37*** und erweitern sie um die im UML Diagramm rot dargestellten Elemente.

Die Methode ***getOberesBildPanel*** gibt das Ergebnis von *tabbedPane.getSelectedComponent* zurück, wobei es durch einen *Cast* in *BildPanel* umgewandelt wird.

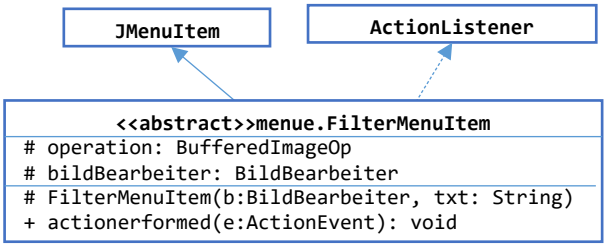
Die Methode ***filter*** ruft *getOberesBildPanel* und - falls diese nicht *null* liefert - ruft sie die Methode *filter* für dieses Panel.

Die Methode ***bearbeiteBild*** ruft *getOberesBildPanel*, stellt fest ob es existiert und prüft dann den Wert *typ*. Falls dieser gleich *ORIGINAL_WIEDERHERSTELLEN* ist ruft sie die Methode *originalWiederherstellen* für das obere Panel, bei *LETZTEN_STAND_WIEDERHERSTELLLEN* ist, ruft sie *letztenStandWiederherstellen* und ansonsten wirft sie eine *IllegalArgumentException*.



Für die verschiedenen Operationen werden wir jeweils einen Menü-Eintrag programmieren.

Wir schreiben uns zunächst die von *JMenuIte* abgeleiteten **abstrakte** Klasse ***FilterMenuItem***, welche *ActionListener* implementiert. Diese Klasse wird als Basisklasse für die verschiedenen Filterklassen dienen.



Der **Konstruktor** ruft den Basisklassenkonstruktor mit *txt* ,registriert sich als der eigene *ActionListener* und speichert den Parameter *b* in der Instanzvariablen.

In ***actionPerformed*** rufen wir für *bildBearbeiter* die Methode *filter* und übergeben ihr die Instanzvariable *operation*.

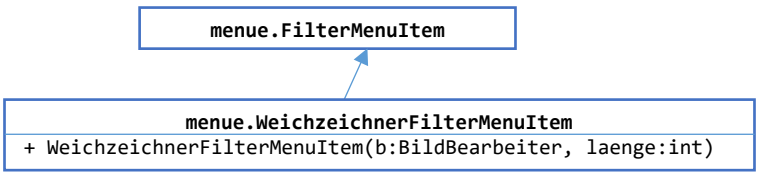
Dann implementieren wir die Methode ***filter*** in der Klasse ***BildPanel***. Sie ruft zunächst *letztenStandSichern* und wendet dann den Filter auf das Raster des letzten Stands an um das angezeigte Bild zu erzeugen:

```
operator.filter(this.letzterStand.getRaster(), this.bild.getRaster());
this.repaint();
```

Convolution-Filter

Diese Filter werden alle mit der Klasse ***ConvolveOp*** realisiert. Unsere Klassen zur Realisierung der Filter sind sehr einfach, sie bestehen nur aus einem Konstruktor. Wir beginnen mit einer Klasse für Weichzeichner

Der **Konstruktor** erzeugt ein Convolve-Op Objekt mit einem laenge*laenge Filter erzeugt:




```
super(b,laenge*laenge+" Weichzeichner");
float[] k = new float[laenge*laenge];
for(int i=0;i<k.length;i++){
   k[i]=1f/(laenge*laenge);
}
Kernel kern = new Kernel(laenge,laenge,k);
operation = new ConvolveOp(kern);
```

und das ist schon alles.

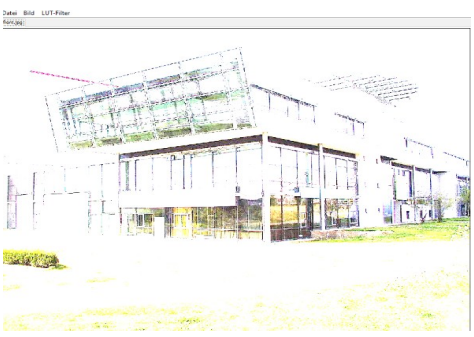
Jetzt initialisieren wir in ***Aufgabe37*** die Instanzvariable ***bildMenu*** mit `new JMenu("Bild")`, fügen zu diesem Menü zwei oder drei *WeichzeichnerFilterMenuItem* mit unterschiedlichen Kerngrößen (3*3, 5*5, 7*7) hinzu und fügen das Menü zur Menüleiste hinzu. Zum Testen können Sie das Bild *front.jpg* aus dem online Kurs verwenden, da es relativ klein ist.

Erzeugen Sie weitere Klassen für Convolution-Filter:

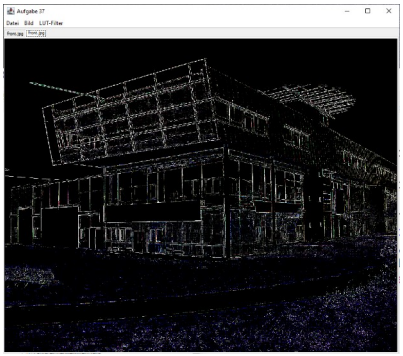
- ***SchaerfeFilterMenuItem***: Ein Schärfe Filter, k = { 0, -1,0, -1,5,-1, 0, -1,0};
- **GaussFilterMenuItem**: ein Gauß Weichzeichner , k = {1,2,1, 2,4,2, 1,2,1};

Eine weitere Kategorie von Filtern sind die Filter, die Konturen oder Kanten im Bild suchen.

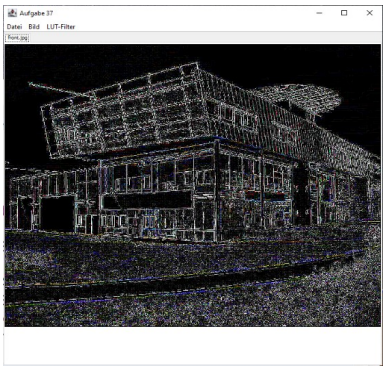
Erstellen Sie zunächst die Klasse ***KantenFilterMenuItem*** mit k= { 0, -1,0, -1,8,-1, 0, 1,0 }; Das Ergebnis sehen Sie im linken Bild. Eine weitere Variante ist ein **LaplaceFilter** mit k= { 1, 2,1, 2,-13,2, 1,2,1}; (mittleres Bild) oder mit { 1,4,1, 4,-20,4, 1, 4,1}; (rechtes Bild).



KanteFilter { 0, -1,0, -1,8,-1, 0, 1,0}



Laplace { 1, 2,1, 2,-13,2, 1,2,1}



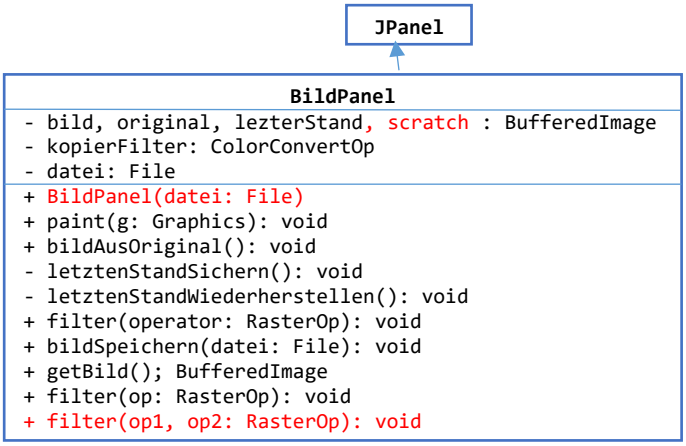
Laplace { 1,4,1, 4,-20,4, 1, 4,1}

Diese Filter haben offensichtlich eine ganz unterschiedliche Wirkung auf die Farbe und die Konturen. Allerdings sind sie unbefriedigend wenn die Kanten nicht mehr gerade sind. Hier bringen oft **zweistufige Filter** ein besseres Ergebnis.

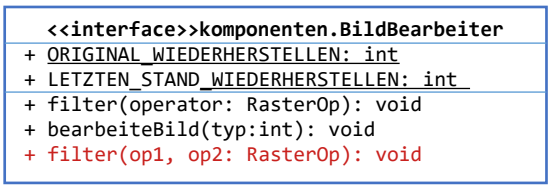
Dazu erweitern wir zunächst die Klasse **BildPanel** um die neue Instanzvariable ***scratch*** und die überladene Methode ***filter***.

Im **Konstruktor** initialisieren wir ***scratch*** genau wie *bild* und *letzterStand*.

In der neuen ***filter*** Methode rufen wir *letztenStandSichern*, wenden *op1* auf *letzterStand* an und speichern das Ergebnis in *scratch*, dann wenden wir *op2* auf *scratch* an und speichern das Ergebnis in *bild*. Diese Filter-Methode führt also zwei Raster-Operatoren hintereinander aus.



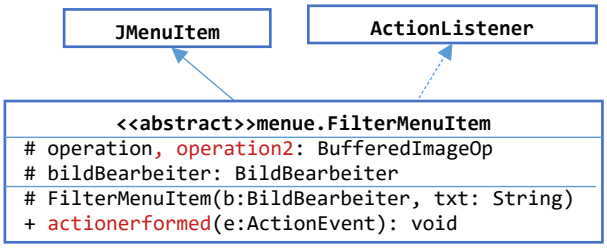
Dann erweitern wir das Interface ***BildBearbeiter*** um die überladene Methode *filter* wie rechts gezeigt.



Schließlich bekommt die Klasse ***Aufgabe37*** eine überladene ***filter*** Methode, die für das obere Bildpanel *filter* mit zwei Operatoren ruft.

Die Klasse ***FilterMenuItem*** bekommt die weitere Instanzvariable ***operation2***, und in ***actionPerformed*** rufen wir die neue *filter* Methode, falls *operation2* nicht *null* ist:

```
if(operation2==null) bildBearbeiter.filter(operation);
else bildBearbeiter.filter(operation, operation2);
```

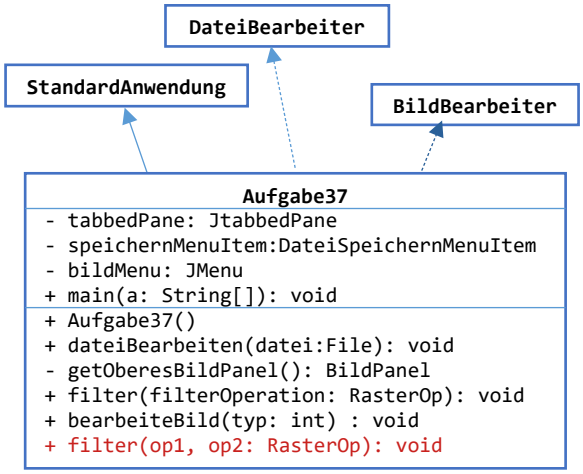


Dann fügen Sie folgende Klasse zum Paket *menu* hinzu:

```
public class SobelFilterMenuItem extends FilterMenuItem {

        public SobelFilterMenuItem(BildBearbeiter b) {
                super(b, "Sobel-Filter");
                float[] k1  = {-1,2,-1,   0,0,0,  1,2,1};
                float[] k2  = {-1,0,1,   -2,0,2,  -1,0,1};

                Kernel kern1 = new Kernel(3,3,k1);
                operation = new ConvolveOp(kern1);
                Kernel kern2 = new Kernel(3,3,k2);
                operation2 = new ConvolveOp(kern2);
        }
}
```



Dieser sogenannte **Sobel-Filter** liefert ein besseres Ergebnis bei unregelmäßigen Konturen.

Wenn Sie mehr über diese Filter nachlesen möchten, können Sie auf z.B. diesen Link zur Dissertation von Dirk Bröder klicken, der die theoretischen Hintergründe schön beschreibt:

https://sundoc.bibliothek.uni-halle.de/diss-online/03/04H003/t5.pdf

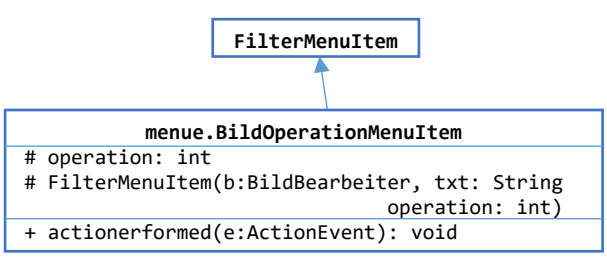
Dort finden Sie auch noch weitere Filter die Sie bei Interesse implementieren können, z.B. den Prewitt-Filter oder den „Laplace of Gauß“ Operator.

Das Originalbild wiederherstellen oder den letzten Schritt rückgängig machen

Dazu schreiben wir nebenstehende Klasse ***BildOperationMenuItem***.

Der **Konstruktor** ruft den Basisklassenkonstruktor und speicher seinen dritten Parameter in der Instanzvariable.

In ***actionPerformed*** rufen Sie `bildbearbeiter.bearbeiteBild(operation)`.



Jetzt fügen Sie in Aufgabe37 zwei solche Menü-Items zum Menü „Bild“ hinzu und übergeben dem einen den Wert LETZTEN_STAND_WIEDERHERSTELLEN, dem anderen ORIGINAL_WIEDERHERSTELLEN.

Bild speichern und speichern unter

Zunächst implementieren wir die Methode ***bildSpeichern*** in der Klasse ***BildPanel*** – diese Methode ist ja bisher noch leer. Sie prüft, ob der Parameter *datei null* ist, wenn ja, setzt sie ihn zu *this.datei*. Dann besorgen wir uns die Datei-Endung und schreiben das Bild auf die Festplatte:

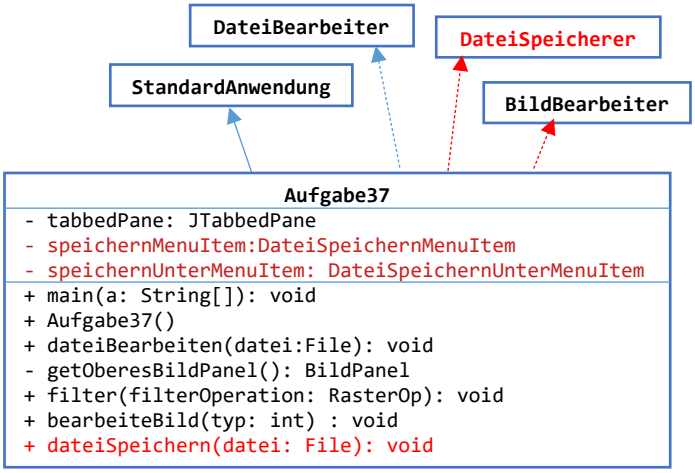
```
String endung = datei.getName().substring(datei.getName().lastIndexOf(".")+1);
ImagerIO.write(bild, endung, datei);
```

Fehler müssen wie gewohnt mit try/catch abgefangen werden.

Lassen Sie dann die Klasse ***Aufgabe37*** das Interface *DateiSpeicherer* implementieren.

Fügen Sie wie in Aufgabe 36 ein *DateiSpeichernMenuItem* sowie ein *DateiSpeichernUnterMenuItem* zum „Datei“ Menü hinzu. Dem *DateiSpeichernUnterMenuEintrag* geben Sie den File-Filter mit, der nur Bild-Dateien zulässt. Rufen sie für beide Menüeinträge zunächst `setEnabled(false)`.

Die beiden Menüeinträge werden als Instanzvariablen gespeichert, damit wir sie aktiv setzen können, sobald das erste Bild geladen ist – vorher macht das ja keinen Sinn. Dazu rufen Sie in ***dateiBearbeiten*** für die Menüeinträge `setEnabled(true)`



Lookup-Table Filter

Diese Filter verwenden eine Farbpalette, die für jeden Farbwert die neue Farbe angibt. Implementieren Sie die von *FilterMenuItem* abgeleitete Klasse Klasse ***NegativFilterMenuItem***, die das Negativ eine Bildes erzeugt wie im Video 8.6 gezeigt. Eine weitere Klasse ***ComicFilterMenuItem*** reduziert die Farben auf 4 Werte pro Kanal wie im gleichen Video gezeigt.

Jetzt wollen wir uns ein ganzes Menü von Lookup-Table Filtern erzeugen, die wir uns von der folgenden Webseite von Dave Williamson holen:

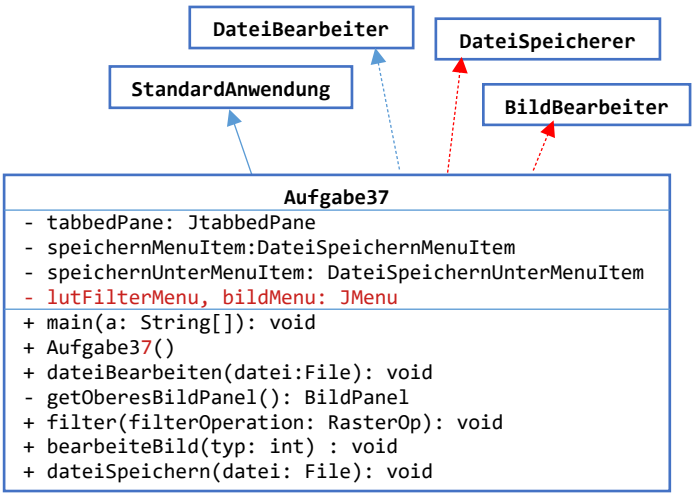
https://github.com/quokka79/DavLUT

Sie finden dort für 14 Filter jeweils eine Datei mit der Endung „.lut“. Holen Sie sich diese Dateien in das Verzeichnis LUT das Sie unterhalb von *src* erstellen.

Um aus diesem Verzeichnis ein komplettes Menü zu erstellen, schreiben wir die Klasse ***LUTFilterMenu*** wie im Video gezeigt.

Legen Sie das Menü wieder in eine Instanzvariable und rufen zunächst *setEnabled(false)* um es erst dann zu aktivieren, wenn das erste Bild eingelesen wurde.

Eine riesige Sammlung von Lookup-Tabellen findet man hier:



http://soliton.vm.bytemark.co.uk/pub/cpt-city/

Diese Filter sind aber nicht im rohen Format wie die von Dave Williamson, sondern für CSS, Gimp, Photoshop und anderen Formaten, d.h. wir können sie nicht so einfach lesen.

Weitere Lookup-Tabellen für bestimmte Farbstimmungen gibt es entweder frei oder zu kaufen wie z.B hier:

https://www.freepresets.com/ http://www.cineplus.ch/kinolut.html

und natürlich haben Photoshop/Affinity/Lightroom u.s.w. schon eine Menge LUT's mit. Sie werden im Laufe Ihres Studiums noch intensiv damit arbeiten.

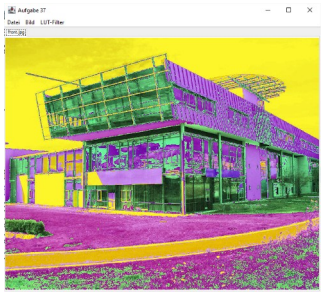
Erzeugen Sie einen Weichzeichner- und Kanten-Filter, indem Sie Klassen von *FilterMenuItem* ableiten, die jeweils einen anderen Kern definieren.

Band-Combine Filter

Die dritte Art von Filtern kombiniert die Original Farben eines Pixels zu neuen Farben mittels einer linearen Operation. Dazu definiert man eine 3x4 Matrix, die mit dem Vektor der Original-Farben multipliziert wird um die neuen Farben zu erzeugen.

Den rechts gezeigten Farbeffekt erzeugt man z.B. mit folgendem Band-Combine Filter:

```
float[][] matrix = { { 1f, 0f, 0f},
                     { 0.5f, 1f, 0.5f},
                     { 0.2f, 0.4f, 0.6f}};
this.operation = new BandCombineOp(matrix, null);
```



In diesem Fall wird das neue Blau B' aus den Farben R,G,B als

B' = 0.2 R + 0.5 G + 0.6 B

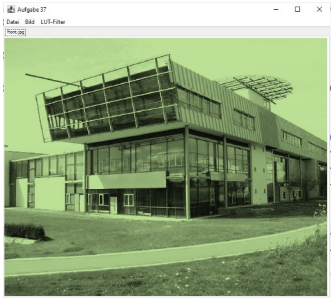
berechnet.

Einen Sepia Effekt bekommt man mit dieser Matrix:

```
float[][] m =  {{0.28f, 0.55f, 0.13f},
                {0.25f, 0.49f, 0.12f},
                {0.19f, 0.38f, 0.09f}};
```

Oder man kann einen Farbton stark betonen, indem man zwei Band-Combine Filter hintereinander schaltet, hier wird grün erst ein Grauton erzeugt und dann grün stark betont, Rot mittelstark und blau gering:

```
 float[][] m1 = { { 0.3f, 0.3f, 0.3f},
                  { 0.3f, 0.3f, 0.3f},
                  { 0.2f, 0.3f, 0.3f}};
 this.operation = new BandCombineOp(m1, null);
 float[][] m2 = { { 0.25f, 0.25f, 0.25f},
                  { 0.3f, 0.3f, 0.3f},
                  { 0.2f, 0.2f, 0.2f}};
this.operation2 = new BandCombineOp(m2, null);
```



Im Web finden Sie jede Menge weiterer Filter zum experimentieren.

Weitere Java Filter Klassen sind ***AffineTransformOp*** für geometriche Transformationen, ***RescaleOp*** zum linearen Aufhellen oder Abdunkeln sowie ***ColorConvertOp*** für Farbrauim-Konversionen.

Eine verbesserte Bibliothek für Bildverarbeitung in Java ist *Java Advanced Imaging (JAI)*, die sehr viel mehr Möglichkeiten bietet, u.A. mehr Dateiformate und eine Redering Pipeline.

Und wer tiefer in die Bildbearbeitung einsteigen will, kann das mit dem Buch „Digitale Bildverarbeitung" von Burger und Burge tun, die mit dem Paket *ImageJ* arbeiten.