Praxisbericht: Objektorientierte Programmierung mit geometrischen Figuren als Komponenten

Christian Wolf

Luisenschule, Gymnasium der Stadt Mülheim an der Ruhr
An den Buchen 36
45470 Mülheim an der Ruhr
christian.wolf@tu-dortmund.de

Abstract: Die Vorgaben und Empfehlungen [Mi99, Br08, Ku89] legen verbindlich den Bereich *Modellieren und Implementieren* als Aufgabe und Ziel des Informatikunterrichts fest. Dabei kommt in der gymnasialen Oberstufe bei der Wahl des objektorientierten Paradigmas und der Programmiersprache Java häufig das Konzept Stifte und Mäuse (SuM) [Sc06] zum Einsatz. Aus guten Gründen erfreut sich SuM der Beliebtheit, insbesondere aufgrund seines einfachen, Stift-basierten Zeichenansatzes. Aufgrund einiger Problematiken jedoch, wird dieses Konzept als durchaus nicht unkritisch für einen zeitgemäßen Informatikunterricht gesehen. Dieser Praxisbericht[1] stellt eine alternative Grafikbibliothek für den Einführungsunterricht in die objektorientierte Modellierung und Programmierung mit Java vor, die die SuM-Problematiken vermeidet, indem sie geometrische Figuren als Komponenten implementiert.

1 Einleitung und Motivation

Bei der Einführung in eine Programmiersprache bietet es sich an, Problemstellungen mit grafischen Ausgaben zu behandeln. Dafür spricht simpel die Anschauung, vielmehr jedoch die durch sie gegebene Möglichkeit einer visuellen Rückmeldung und damit teilweisen Überprüfbarkeit von Programmlösungen. Geradezu ein Paradebeispiel für eine anschauliche Problemstellung ist die Programmierung einer grafischen Verkehrsampel. Diese lässt sich simpel durch ein Rechteck (Ampelgehäuse) und drei darin positionierte sowie ggf. farbig gefüllte Kreise (Lampen) visualisieren. Wie diese Lösung erfordern auch die grafikorientierten Lösungen zahlreicher anderer Problemstellungen die Implementierung und die Handhabung geometrischer Figuren.

Im schulischen Kontext wird bei der Einführung in die objektorientierte Modellierung (OOM) und Programmierung (OOP) mit Java das SuM-Konzept mit dessen gleichnamiger Bibliothek verbreitet eingesetzt. In der Beispiellernsequenz „(2) Sequenz „objektorientiert

[1] Der Großteil dieses Berichts basiert auf [Wo11]. Zur Wahrung der Leserlichkeit sind Passagen und Sätze, die jener Quelle dem Wortlaut oder Sinn nach entnommen sind, nicht als Entlehnung kenntlich gemacht.

allgemein"" des Lehrplans [Mi99] wird es als „Ausgangspunkt erster Anwendungsentwicklungen" nach dem objektorientierten Paradigma gesehen.

SuM unterstützt grafikorientierte Lösungen durch seinen einfachen, dennoch universell einsetzbaren, Zeichenansatz: Ein Stift-Objekt wird über eine Zeichenfläche navigiert und hinterlässt im abgesetzten Zustand eine Zeichenspur auf dieser. Didaktisch ist der Ansatz sehr wertvoll und legitimiert, weil sich den Schülerinnen und Schülern aufgrund der Analogie zu einem realen Stift das programmgesteuerte Zeichnen intuitiv sowie einleuchtend erschließt. Weitere Gründe sprechen für SuM, z.B. erlaubt es die Schulung grundlegender Programmierkenntnisse (wie Kontrollstrukturen) bei ersten Problemlösungen, die eher einen imperativen Charakter aufweisen.

Den Vorzügen von SuM stehen jedoch auch Problematiken gegenüber, aufgrund derer SuM zur Schulung des Prozessbereiches *Modellieren und Implementieren* nicht uneingeschränkt empfohlen werden kann:

Problematik 1 (Probleminadäquate OOM [Sp04, Sp06]): Kommt man dem Aufforderungscharakter des Stift-Ansatzes zu einer einfachen und naheliegenden Modellierung der grafischen Ampel nach, so besteht diese aus der Klasse Ampel, welche einen Stift zum Zeichnen aggregiert. Sicherlich ist dieses naive Modell zulässig und es existiert auch nicht *das* richtige Modell. Allerdings erscheint mit objektorientiertem Verstand das Modell bestehend aus einer Ampel, welche drei Lampen aggregiert, angemessener. Die Lampen stellen schließlich abgrenzbare Objekte mit eigenen Attributen (An-Aus-Zustand und Leuchtfarbe) und Diensten (an- und ausschalten) dar, die auch als solche gehandhabt werden wollen. Überdies ist dieses Modell geeigneter für die Implementierung (siehe Problematik 2).

Spollwig bezeichnet diese Modellierungsproblematik gar als Dilemma und konstatiert [Sp06]: „SuM bewirkt [...], dass jedes Objekt prinzipiell Attribute aus der Kern-Bibliothek haben muss, ein eigenartiges Abbild der gefundenen Objekte. Überspitzt könnte man auch sagen, wer als einziges Werkzeug einen Stift hat, für den besteht die ganze Welt aus Strichen. Damit wird die Forderung nach sauberer Modellierung konterkariert und den Schülern vom ersten Tage an ein falsches Bild von den Entwurfsaufgaben der Informatik und den Möglichkeiten der OOP vermittelt. [...] Zusammengefasst kann man feststellen, dass das SuM-Konzept zwar alle wichtigen Merkmale von OOP darstellt, aber lediglich die objektorientierte Programmiertechnik aufzeigen kann, jedoch OOA[2] und OOD[3] im Anfangsunterricht nicht richtig unterstützt und so nicht empfohlen werden kann.".

Problematik 2 (Fokussierung von Zeichenprimitiven): Das naive Modell und das Stift-basierte Zeichnen offenbaren eine weitere Problematik. Dazu betrachte man die Programmierung eines Phasenwechsels der Ampel, beispielsweise von Grün auf Gelb, dessen Lösung in etwa so aussieht: Stift hochsetzen, Stift an die Position des grün gefüllten Kreises bewegen, Stift absetzen, Löschen dieses Kreises, Zeichnen eines neuen und mit anderer Farbe gefüllten Kreises, Stift hochsetzen, Stift an die Position des mit gelb zu

[2] OOA: Objektorientierte Analyse
[3] OOD: Objektorientiertes Design

füllenden Kreises bewegen, Stift absetzen, Löschen dieses Kreises sowie Zeichnen eines neuen und mit gelber Farbe gefüllten Kreises. Man erkennt leicht, dass zur eigentlichen Problemlösung der Fokus erheblich auf Zeichenprimitiven liegen muss. Auch tritt damit eine Vermischung von Zuständigkeiten ein, wie man sie eigentlich beim Programmieren zu vermeiden versucht (Paradigma des *Separation of Concerns*). Dabei liegt doch eher der Wunsch nahe, objektorientiert zu programmieren und, statt eines Stift-Objekts, Lampen-Objekte zu benutzen: Schalte die grüne Lampe aus und schalte die gelbe Lampe an. Ein weniger gravierender Nachteil ist, dass Quelltexte aufgebläht werden und sie weniger gut lesbar sind.

Es ist leicht einsehbar, dass die genannten Problematiken beim grafikorientierten Programmieren, z.B. bei der Erstellung grafischer Benutzeroberflächen in realen Software-Projekten, möglichst nicht anfallen dürfen. Java bietet mit Swing [LEW02] eine modular aufgebaute, mächtige und vor allem objektorientierte Grafikbibliothek, die sich für die Entwicklung selbst komplexer Oberflächen eignet.

Warum also sollte nicht der Versuch unternommen werden, Swing in didaktisch aufbereiteter Form im schulischen Informatikunterricht einzusetzen?

2 GeoFaSC: Geometric Figures as Swing Components

2.1 Entwicklung des Konzeptes

Beobachtung 1 (Grafische Lösungen erfordern geometrische Figuren): Das Beispiel der grafischen Ampel deutete bereits an, dass grafische Lösungen häufig die Implementierung geometrischer Figuren, die Komposition dieser zu komplexen Figuren und die Handhabung dieser Figuren erfordern. Weitere treffende Beispiele aus [Sc06] verdeutlichen dies:

- Pfeil und Dartscheibe (Linie, Kreis und mehrere konzentrische Kreise),
- abprallende Kugeln (Kreise),
- Wolf und Rotkäppchen (Polygonzug/ Polylinie und Kreis) oder auch
- Lokomotive, Güter- und Personenwagen (Rechtecke, Kreise und Linien).

Beobachtung 2 (Swing als objektorientierte, komponentenbasierte Grafikbibliothek): Zwar stellt Swing grafische Elemente zur Benutzerinteraktion zur Verfügung, diese werden aber objektorientiert erzeugt, behandelt und können einfach zu Komponenten zusammengesetzt werden. Zudem können fertige Swing Komponenten ohne Verwendung von Zeichenprimitiven verwaltet werden.

Beobachtung 3 (Grafikbibliothek uGrafik): Spollwig schlägt in [Sp06, Sp08] eine Grafikbibliothek mit Grafikklassen für Delphi namens *uGrafik* vor. uGrafik soll „zum

gleichen didaktischen Ansatz im Anfangsunterricht verwendet werden wie SuM", allerdings „mit einem entscheidenden Unterschied: Die Schüler arbeiten vom ersten Tage an durchgängig mit Objekten, die erzeugt und manipuliert werden." [Sp06].

Folgerung: Diese Beobachtungen führten zur Entwicklung des GeoFaSC-Konzeptes. Dieses Konzept steht namentlich für eine Sammlung *geometrischer Figuren als Swing Komponenten*. Leitgedanke des Konzeptes ist, dass jede Figur nicht als ein Artefakt eines (Zeichen)Werkzeuges, sondern selbst als Werkzeug verstanden wird. Die konzeptuellen Anforderungen an eine Figur sind:

1. Eine Figur ist ein Objekt.
2. Eine Figur ist ohne Zeichenaufwand erzeugbar und verwaltbar.
3. Eine Figur besitzt Attribute (z.B. Größe und Position), die die Figur eindeutig in einem Bezugssystem beschreiben.
4. Eine Figur besitzt Abfrage- und Änderungsdienste, mit denen man den Zustand der Figur lesen und schreiben kann.
5. Eine Figur ist eine (Swing) Komponente, d.h. sie kann andere Figuren inkludieren, andere Komponenten inkludieren und auch selbst inkludiert werden.

Der genannte Leitgedanke stellt eine Maßnahme gegen Problematik 1 dar. Zum einen besteht beim Modellieren grafikorientierter Lösungen mit GeoFaSC nicht der Aufforderungscharakter, ständig nur ein Zeichenwerkzeug zu verwenden. Dies ist durch das Fehlen *eines* universellen Zeichenwerkzeuges gegeben. Zum anderen ermutigt der GeoFaSC-Ansatz eher, problemspezifische Klassen für nicht-triviale, wirklichkeitsnähere Modelle vorzusehen und diese dann auch einfacher zu implementieren (vgl. Teilkapitel 2.4).

2.2 Programmierwerkzeug des Konzeptes

Die GeoFaSC-Bibliothek ist die Umsetzung des GeoFaSC-Konzeptes als Programmierwerkzeug in Java. Die Bibliothek stellt, aufbauend auf Swing und ihrem Komponentenansatz, die geometrischen Figuren `Circle`, `Ellipse`, `LineSegment`, `Point`, `Polyline` (geschlossen auch als Polygon nutzbar), `Rectangle` und `Square` wie Komponenten für grafische Benutzeroberflächen bereit.

Diese Basisfiguren sind nach dem in Swing üblichen Architekturmuster Model-UI-Component [Fo] implementiert und trennen so die Zuständigkeiten Datenmodell, Präsentation und Programmsteuerung. Die Figuren sind über ihre UI-Klassen selbst verantwortlich, sich zu zeichnen. Änderungen auf ihnen führen zum automatischen Neuzeichnen. Das Zeichnen wird also gekapselt und liegt nicht zwingend in der Zuständigkeit des Programmierers. Dies stellt offensichtlich eine Maßnahme gegen Problematik 2 dar. Durch die Entlastung des Programmierers vom Zeichnen, kann dieser die Problemlösung besser fokussieren.

Dennoch können zur Laufzeit Modell und Präsentation einer Figur bei Bedarf ausgetauscht werden. Damit besteht nach wie vor die Möglichkeit, die volle Kontrolle über das Zeichnen

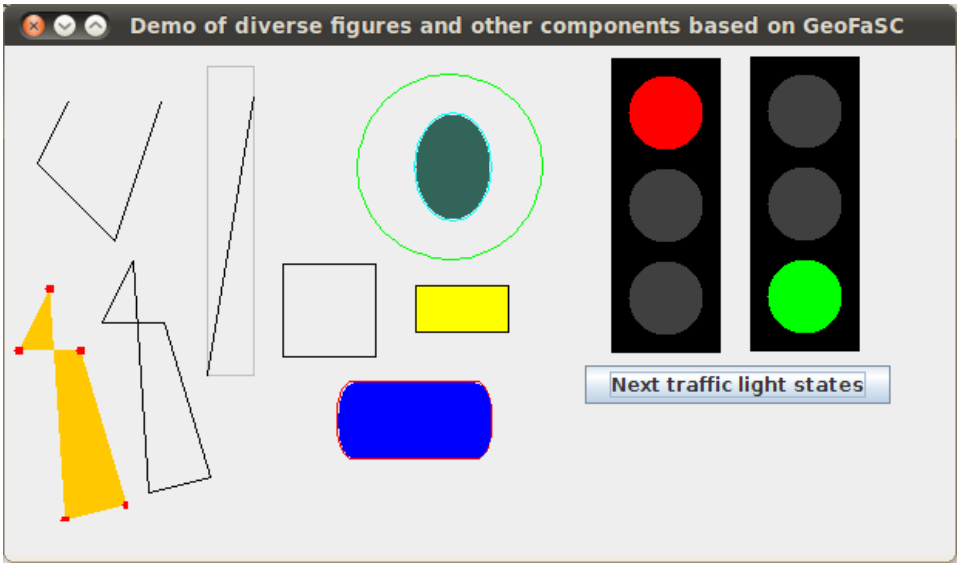

Abbildung 1: Schnappschuss von GeoFaSC-Basisfiguren, den daraus kombinierten Ampel-Figuren und einem JButton in einem gemeinsamen Container (vgl. die Klasse `FiguresDemo` in den Quellen von GeoFaSC [Wo10]).

einer Figur zu erlangen. Erfordernisse hierfür könnten beispielsweise das Zeichnen einer dickeren Linie oder die Notwendigkeit eines spezielleren Figurenmodells sein.

Die Erzeugung einer komplexen Figur (z.B. für die Ampel die drei Kreise/ Lampen in einem Rechteck) oder Komponente (z.B. ein JLabel in einer Figur zur Beschriftung derselben) wird einfach durch Schachtelung einer Figur mit einer Figur bzw. einer Figur mit einer Komponente erreicht.

Alle Anforderungen des Konzeptes sind mit dieser Bibliothek realisiert. Abbildung 1 zeigt GeoFaSC-Figuren in Ausführung. Abbildung 2 veranschaulicht die Architektur der GeoFaSC-Bibliothek und zeigt die Schnittstellen zu Swing. Die GeoFaSC-Bibliothek ist als Open-Source-Software mit ausführlicher Klassen-Dokumentation unter [Wo10] frei verfügbar.

2.3 Exemplarische Implementierung: Geometrische Figuren

Die folgende Sequenz zeigt exemplarisch die objektorientierte Verwendung von Basisfiguren, insbesondere wird die Umsetzung der ersten vier konzeptuellen Anforderungen belegt:

```
Rectangle rectangle = new Rectangle(width, height);
rectangle.setRoundedCorners(true);
```

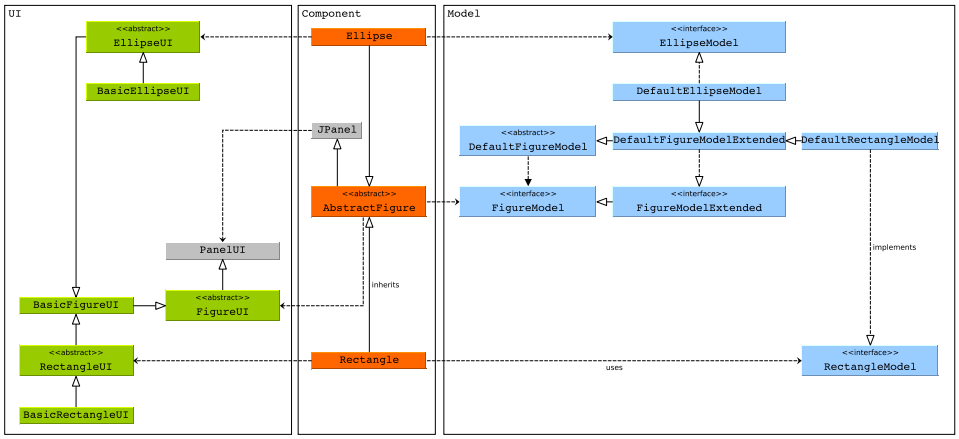

Abbildung 2: Allgemeine Struktur von GeoFaSC (die weiteren Component-Klassen `Circle`, `LineSegment`, `Point`, `Polyline` und `Square` sowie deren Model- und UI-Klassen sind nicht dargestellt). Weitere Klassendiagramme können der Quelltextreferenz von GeoFaSC [Wo10] entnommen werden.

```
rectangle.setArcSize(arcWidth, arcHeight);
...

Circle circle = new Circle();
// Zugriff auf den Kreis nur über sein Modell
CircleModel model = circle.getModel();
model.setRadius(60);
model.setFigureLocation(x, y);
...

// Die geschlossene Polylinie ist ein Polygon
Polyline polygon =
     new Polyline(new Point(30,10),  new Point(10,50),
                  new Point(50,50),  new Point(80,150),
                  new Point(40,160), new Point(30,10));
polygon.removePoint(10,50);
...

LineSegment line = new LineSegment(x1, y1, x2, y2);
// Austausch der Präsentation
line.setUI(new BasicLineSegmentUI() {

  @Override
  protected void paintFigure(Graphics g, AbstractFigure f) {
    FigureModel model = f.getModel();
```

```
    // Zeichnen der Bounding-Box der Linie
    g.setColor(Color.LIGHT_GRAY);
    g.drawRect(0, 0, model.getWidth(), model.getHeight());
    // Zeichnen der Linie
    super.paintFigure(g, f);
  }
});
```

2.4 Exemplarische Implementierung: Grafische Ampel

Im Folgenden ist eine Implementierung des Beispiels der grafischen Ampel aufgeführt. Es zeigt die Umsetzung der fünften konzeptuellen Anforderung. Die Kernlösung umfasst die zwei Klassen `Lamp` und `TrafficLight` mit einer Kompositionsbeziehung zwischen diesen. Die grafische Ausgabe zweier `TrafficLight`s zeigt Abbildung 1.

Die Klasse `Lamp` ist aus `Circle` abgeleitet und repräsentiert namentlich Lampen als gefüllte Kreise, die an- und ausgeschaltet werden können. Beim Instanziieren einer Lampe werden ihre Position bezüglich eines Containers (hier eine `TrafficLight`), dem diese noch hinzugefügt werden kann, ihre Größe sowie An-Farbe und ihr initialer Zustand festgelegt.

```
import java.awt.Color;

public class Lamp extends geofasc.swing.Circle {

  private static Color OFF_COLOR = Color.DARK_GRAY;
  private Color mOnColor;
  private boolean mIsOn;

  public Lamp(int x, int y, int radius, Color onColor,
              boolean isOn) {
    super(x, y, radius);
    mOnColor = onColor;
    setFigureFilled(true);
    setOn(isOn);
  }

  public boolean isOn() {
    return mIsOn;
  }

  public void setOn(boolean on) {
    mIsOn = on;
    if (mIsOn)
      setFigureFillColor(mOnColor);
```

```
      else
        setFigureFillColor(OFF_COLOR);
    }

  } // class Lamp
```

Die Klasse `TrafficLight` ist aus `Rectangle` abgeleitet und repräsentiert namentlich Ampeln mit ihren verschiedenen Phasen Rot, Rot-Gelb, Gelb und Grün. Der Konstruktor von `TrafficLight` verdeutlicht den Komponentenansatz von GeoFaSC. Dort werden die drei Lampen einer Ampel erzeugt und dieser einfach hinzugefügt. Die Positionen der Lampen sind relativ zur linken oberen Ecke des rechteckigen Ampelgehäuses. Eine Ampel wird damit zur komplexen Figur, dennoch kann auf dieselbe und ihre Lampen objektorientiert zugegriffen werden (siehe `setState`-Methode).

```
  import java.awt.Color;

  public class TrafficLight extends geofasc.swing.Rectangle {

    private Lamp mRedLamp, mYellowLamp, mGreenLamp;

    // 1 = RED, 2 = RED_YELLOW, 3 = GREEN, 4 = YELLOW
    private int mState;

    public TrafficLight() {
      super(70, 190);
      setFigureFillColor(Color.BLACK);
      setFigureFilled(true);
      mState = 1;

      mRedLamp = new Lamp(10, 10, 50, Color.RED, true);
      mYellowLamp = new Lamp(10, 70, 50, Color.YELLOW, false);
      mGreenLamp = new Lamp(10, 130, 50, Color.GREEN, false);

      add(mRedLamp);
      add(mYellowLamp);
      add(mGreenLamp);
    }

    public void setNextState() {
      setState(mState + 1);
    }

    public void setState(int state) {
      mState = state;
      if (mState > 4)
```

```
      mState = 1;

    switch (mState) {
      case 1: // RED
        mRedLamp.setOn(true);
        mYellowLamp.setOn(false);
        mGreenLamp.setOn(false);
        break;
      case 2: // RED_YELLOW
        ...
      ...
    }
  }

} // class TrafficLight
```

Zur Visualisierung einer Ampel kann diese einem entsprechenden Container hinzugefügt werden. Aufgrund der Kompatibilität zu Swing ist man dabei nicht auf eigens implementierte Container angewiesen.

```
TrafficLight tLight = new TrafficLight();
JLayeredPane canvas = new JLayeredPane();
JFrame frame = new JFrame("Traffic Light");
frame.setContentPane(canvas);
frame.setVisible(true);
canvas.add(tLight);
```

Grafische Änderungen der Ampel können wie folgt programmiert werden. Die Figur und ihre inkludierten Figuren werden automatisch und positionsgerecht neu gezeichnet.

```
tLight.moveFigureLocatioBy(40, 30);
tLight.setNextState();
tLight.setToolTipText("I'm a traffic light");
tLight.setNextState();
tLight.moveFigureLocationXBy(-20);
...
```

3 Praktischer Einsatz

3.1 Anwendungsmöglichkeiten

GeoFaSC ist mächtig genug, um vielfältige Anwendungsmöglichkeiten zu schaffen. Ganz allgemein können mit GeoFaSC die einführenden, objektorientierten Themen und Konzep-

te unter Berücksichtigung verbesserter Modellierungen thematisiert werden. Hierzu lassen sich im Einzelnen zählen (vgl. orientierend [Sc06]):

- Klassen und Objekte:
  - Erzeugung und Benutzung von Objekten aus vorgegebenen Klassen,
  - Aufruf von Diensten der Objekte,
  - Unterscheidung der Dienste hinsichtlich Anfragen und Aufträgen,
  - Punktnotation hinsichtlich Attributen und Diensten.
- Struktur und Aufbau einer Java-Klasse und eines Java-Programms;
- Kontrollstrukturen;
- Programmierung eigener, auch abstrakter, Klassen, besonders unter Benutzung der verschiedenen Objekt- und Klassenbeziehungen (Assoziation, Aggregation, Komposition und Vererbung);
- Ereignisorientierung/ Ereignisbehandlung (allerdings Swing-like);
- Komponentenprinzip (Figuren enthalten Figuren oder Komponenten);
- Entwurfsmuster Model-View-Controller (MVC) z.B. durch die Erweiterung von GeoFaSC um die geometrische Figur `Arc`.

Ein inhärenter Schwerpunkt des Konzeptes ist seine Grafikorientierung. Ganz allgemein eignet es sich damit besonders für Projekte und Aufgabenstellungen, deren Lösungen grafische Ausgaben umfassen. Hingegen definiert das Konzept ganz bewußt keine speziellen, über seine Zwecke hinausgehenden Klassen, mit denen sich weiterführende Themen, wie z.B. dynamische Datenstrukturen oder Netzwerkprogrammierung, behandeln ließen.

Darüber hinaus eröffnet und übt GeoFaSC generelle sowie spezielle Programmierprinzipien grafischer Benutzeroberflächen. Dies liegt selbstverständlich an der Orientierung an Swing selbst. Dieses Wissen kann gewinnbringend bei der Programmierung von Benutzeroberflächen eingesetzt werden (z.B. Komponentenprinzip, Ereignisbehandlung).

Beispiele für konkrete, unterrichtspraktische Beispiele basierend auf GeoFaSC sind die folgend aufgeführten. Deren Quelltexte und teilweise auch Unterrichtsmaterialien finden sich in [Wo10, Wo11]. Die Beispiele sind an die aus [Sc06] angelehnt, um die Anwendbarkeit von GeoFaSC als (Teil)Alternative für SuM aufzuzeigen:

- `Lamp`, `TrafficLight` und `DiscoLights`: Ein Objekt DiscoLights repräsentiert einen Kasten, in dem mehrere bunt leuchtende Lamps angeordnet sind. DiscoLights kann in Anlehnung an TrafficLight (vgl. Kapitel 2.4) implementiert werden.
  (→ Klassenbeziehungen, Komponentenansatz, Arrays)
- `Bullet` und `FlyingBullets`: Eine Bullet soll eine Kugel (visualisiert als Kreis) repräsentieren, die man in eine bestimmbare Richtung bewegen kann. FlyingBullets

bewegt mehrere Bullets über eine abgegrenzte Zeichenfläche, wobei eine Bullet an den Rändern der Zeichenfläche abprallt (einfach erweiterbar auch zum Abprallen bei Kollision mit anderen Bullets).
(→ Klassenbeziehungen, Animation)

- `Pencil`, `Wolf` und `Rotkaeppchen`: Die Klasse Pencil ahmt die Stift-Klasse von SuM mit derselben Funktionalität nach. Implementiert wird sie mit Hilfe der Klasse Polyline. Mit dem eigenen Pencil lässt sich die Suchfährte eines Wolfes nach Rotkaeppchen programmieren. Alternativ lässt sich die Klasse Wolf auch direkt mit der Klasse Polyline realisieren.
(→ Eigene Klassen, Aufbau einer Java Klasse)

- `Windradpark`, `Windrad` und `TwinFluegel`: Selbstredend besteht ein Windradpark aus mehreren animierten Windrädern. Ein Windrad ist aus einem Twinfluegel und einem Mast zusammengesetzt. Ein Twinfluegel kann um beliebige Grad rotiert werden.
(→ Alle Kontrollstrukturen, Animation)

3.2 Voraussetzungen

Für die Anwendung von GeoFaSC gibt es einige Voraussetzungen didaktischer und technischer Art zu beachten. GeoFaSC ist generell für Lerngruppen der Sekundarstufe II konzipiert. Zu den Lernvoraussetzungen der Schüler aus diesen Lerngruppen sollte gehören, dass sie die grundlegenden geometrischen Figuren und ihre mathematischen Bestimmungen beherrschen sollten. Letzt genannte Voraussetzung ist hilfreich, da die Bibliothek die Figuren nach ihren mathematischen Bestimmungen implementiert. Die Schüler sollten zudem den Koordinatenbegriff beherrschen und im Umgang mit Koordinatensystemen geübt sein. Der Grund ist, dass die GeoFaSC-Figuren überwiegend koordinatenabhängig gezeichnet und verwaltet werden. Auch für das Verschachteln von Figuren zu komplexen Figuren ist der Koordinatenbegriff essentiell.

Da GeoFaSC gänzlich in englischer Sprache implementiert ist, sollten die Schüler über grundlegende Englisch-Kenntnisse verfügen. Der Wortschatz muss ggf. nur um wenige neue Begriffe, wie etwa `Canvas` oder `Frame` erweitert werden. Ansonsten sind alle Bezeichner möglichst selbstsprechend, beispielsweise `moveFigureLocation(x, y)`.

Obwohl GeoFaSC die Zuständigkeiten Modell, Präsentation und Steuerung trennt, reichen für den Umgang mit GeoFaSC die Figurenklassen und einige wenige ihrer Konstruktoren und Methoden aus. Die Benutzung von Klassen für eigene Modelle und Präsentationen kann der Auseinandersetzung mit dem MVC-Entwurfsmuster vorbehalten bleiben.

Einführend sollte das einfache Prinzip zum Anzeigen der Figuren in GeoFaSC geklärt werden. Es würde hierzu die Verwendung eines beliebigen Top-Level-Containers aus Swing als Elter-Container für die Figuren ausreichen. Viel einfacher verwendet man dazu aus dem Paket `geofasc.swing.tool` die Klassen `Canvas` und `Frame`: Ein `Frame`-Objekt hat ein `Canvas`-Objekt, welches beliebig viele Figuren aufnimmt und ebenenartig anordnet.

Zu den technischen Voraussetzungen gehören die Auswahl und Bereitstellung einer Java-Entwicklungsumgebung mit einem Java Development Kit ab Version 1.5.20. GeoFaSC kann in BlueJ, Netbeans oder auch Eclipse als Bibliothek einfach eingebunden werden.

Literaturverzeichnis

[Br08] Brinda, T. et al: Grundsätze und Standards für die Informatik in der Schule. Bildungsstandards Informatik für die Sekundarstufe I. Empfehlungen der Gesellschaft für Informatik e. V. erarbeitet vom Arbeitskreis "Bildungsstandards". *LOG IN*, 28(150/151), 2008.

[Fo] Fowler, A.: A Swing Architecture Overview. Bericht, Oracle Sun Developer Network (SDN). `http://java.sun.com/products/jfc/tsc/articles/architecture/` (abgerufen: 4. März 2011).

[Ku89] Kultusministerkonferenz: Einheitliche Prüfungsanforderungen Informatik. `http://www.kmk.org/fileadmin/veroeffentlichungen_beschluesse/1989/1989_12_01-EPA-Informatik.pdf` (abgerufen: 31. März 2011), 1989. i.d.F. vom 05.02.2004.

[LEW02] Loy, M.; Eckstein, R. und Wood, D.: *Java Swing*. O'Reilly Media, 2002.

[Mi99] Ministerium für Schule und Weiterbildung: *Richtlinien und Lehrpläne für die Sekundarstufe II - Gymnasium/Gesamtschule in Nordrhein-Westfalen. Informatik*. Ritterbach Verlag, 1999.

[Sc06] Schriek, B.: *Informatik mit Java - Eine Einführung mit BlueJ und der Bibliothek Stifte und Mäuse, Band I*. Nili-Verlag, 2006.

[Sp04] Spollwig, S.: Kritisches zu Stiften und Mäusen - Was ist objektorientierte Modellierung? *LOG IN*, 130, 2004.

[Sp06] Spollwig, S.: Von Stiften und Mäusen oder „Was heisst objektorientierte Modellierung?" `http://oszhdl.be.schule.de/gymnasium/faecher/informatik/didaktik/sum/sum-kritik.htm` (abgerufen: 4. März 2011), 2006.

[Sp08] Spollwig, S.: *delphi class in a box*. Cornelsen, 2008.

[Wo10] Wolf, C.: GeoFaSC - Geometric Figures as Swing Components. `http://www.geofasc.de` (abgerufen: 4. März 2011), 2010.

[Wo11] Wolf, C.: Geometrische Figuren als Swing Komponenten – Ein Konzept zur Einführung in die objektorientierte Modellierung und Programmierung. Schriftliche Hausarbeit im Rahmen der Zweiten Staatsprüfung für das Lehramt an Gymnasien und Gesamtschulen, 2011.