MMProg: Praktikum: WiSe 2017/18: Ball03

Musterlösung: SVN-Repository

Inhaltsverzeichnis

1 Ziel

Ziel dieser Praktikumsaufgabe ist es, Lösungen des zweiten Teils des Tutoriums zu modularisieren. Die Modularisierung hat mehrere Vorteile:

Das Prinzip „Don't repeat yourself“ (DRY) wird unterstützt. Um dies zu erreichen, muss sichergestellt werden, dass viele Module in diversen Projekten wiederverwendet werden können.[1][2]
Mehrere Programmierer können gleichzeitig an einem Projekt arbeiten. Hier muss sichergestellt sein, dass möglichst saubere Schnittstellen definiert werden, damit ein Programmierer nicht ständig an die Änderungen eines anderen Programmierers anpassen muss. (Die Definition von Schnittstellen ist eine der Kernaufgaben von Informatikern. Dabei handelt es sich um einen kreativen Prozess. Der Programmierer hingegen braucht „lediglich“ die Spezifikation umzusetzen. Das ist i. Allg. deutliche weniger kreativ.)
Jede Änderung an einem vorhandenen Code kann Fehler zur Fogle haben. Daher ist es von Vorteil, wenn ausgetestete, wiederverwendbare Module bestehen. Bei der Fehlersuche befindet sich der Code i. Allg. nicht in einem dieser Module, sondern im neu erstellten Code. Das vereinfacht die Fehlersuche deutlich.

2 Aufgaben

Laden Sie das leere Projekt WK_Ball03_empty auf Ihren Rechner. **Installieren Sie aber nicht die Node.js-Module**, das machen Sie später. Sie finden das leere Projekt im Repository-Pfad `https://glossar.hs-augsburg.de/beispiel/tutorium/es6` im Unterordner `empty`.

Erstellen Sie ein neues Projekt `praktikum03` und kopieren Sie die Ordner `src` und `web` (samt Inhalt) sowie alle Dateien, die Sie im Wurzelverzeichnis des Projektes `WK_Ball03_Empty` finden, mittels `Ctrl-/Apfel-C` `Ctrl-/Apfel-V` in Ihr eigenes Projekt. (Die Frage, ob WebStorm seinen eigenen File Watcher zum Übersetzen von ES6-Code in ES5-Code verwenden soll, beantworten Sie bitte mit „No“. Das erledigt webpack für Sie.)

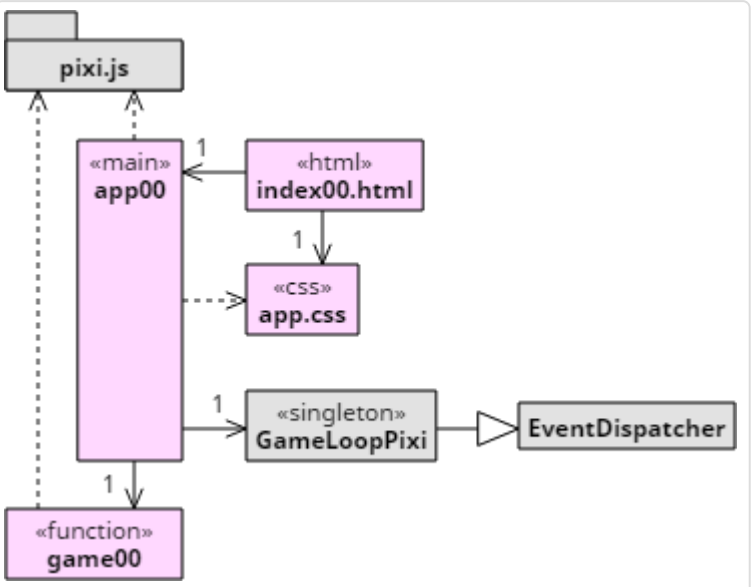
Sie können Ihr Projekt zur Übung auch im Subversion-Repository speichern. Das ist aber nicht so wichtig.

Nun können Sie in Ihrem eigenen Projekt die benötigten Node.js-Module installieren: `npm i`.

In Ihrem Projekt finden Sie zwei Web-Anwendungen: `index00.html` und `index01.html`. `index00.html` verwendet die gepackte Version von `app00.js`, die ihrerseits das Spiel `game00` einbindet. `index01.html` verwendet die gepackte Version von `app01.js`, die ihrerseits das Spiel `game01/game.js` einbindet. Beachten Sie: Im ersten Fall besteht das gesamte Speil aus lediglich einer Datei (`game00`), während im zweiten Fall ein Ordner mit diversen Teilmodulen zum Einsatz kommt.

Schreiben Sie Ihre Lösungen der Aufgabe 1 in die Dateien, die Sie im Ordner `game01` vorfinden.

2.1 Aufgabe 0: Analyse von `game0.js`



Moduldiagramm von `app00`

Es gibt in `WK_Ball03_empty` eine `app0` mit zugehörigem Modul `game0`. Dabei handelt es sich um eine App, die im Sinne des zweiten Teils des Tutoriums erstellt wurde. Sehen Sie sich zunächst diese App an und analysieren Sie, wie diese funktioniert. Beachten Sie insbesondere Folgendes:

Es wird ein Ball-Modell erzeugt, mit Attributen „Radius“, „Position“, „Geschwindigkeit“ und „Beschleunigung“.
Dieser Ball wird durch einen einfarbigen Kreis mit Rand visualisiert.

Der Ball bewegt sich gemäß seinen Initialparametern schräg über die Bühne, wobei seine Geschwindigkeit stetig zunimmt.
Bei einer Kollission mit dem Bühnenrand ändert er seine Bewegungsrichtung.

Die gesamte beschriebene Spiellogik ist in einer Datei enthalten: `game00.js`. So eine Datei bezeichne ich als **Moloch**. Das Prinzip „Implementiere keinen Moloch“ nennt man fachsprachlich „Modularisierung“.

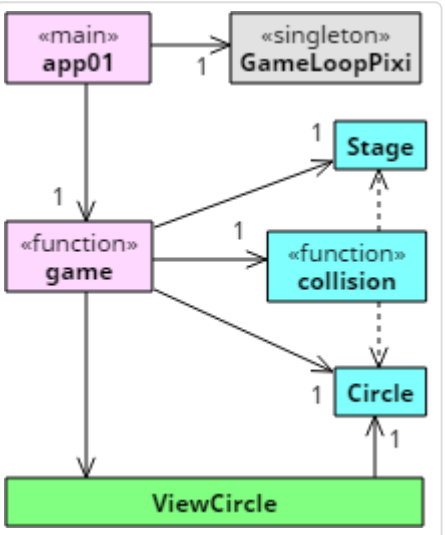
Beachten Sie bitte auch, dass einige Attribute und Funktionen gegenüber dem zweiten Teil des Praktikums umbenannt wurden.

Eigentlich ist das Programm gar nicht so molochartig, wie es zunächst scheint. Es kommen schon einige Module zum Einsatz:

`index.html`: Eine HTML-Seite zum Starten der eigentlichen Web-Anwendung, sobald sie vom Browser geladen wird.
`app.scss`: Eine SCSS-Datei, die das Layout der HTML-Datei festlegt (insbesondere die Hintergrundfarbe).
`pixi.js`: Eine sehr mächtige 2D-Grafik-Bibliothek, die sehr modular aufgebaut ist.
`GameLoopPixi`: Eine Game-Loop-Klasse, die Ihnen im Rahmen des Praktikums zur Verfügung gestellt wird. Diese Modul benutzt seinerseits das Hilfsmodul `EventDispatcher`.
`app0.js`: Das Hauptmodul. Es stellt mit Hilfe der zuvor genannten Module die Spielumgebung zur Verfügung: Eine PIXI-Bühne, eine Gameloop sowie eine CSS-Datei. Sobald die Umgebung erstellt und initialisiert wurde, startet diese Modul das eigentliche Spielmodul `game00.js`.
`app0.js`: Das immer noch molochartige Spielmodul, das im Rahmen des Praktikums in diverse Einzelmodule aufgeteilt werden wird.

Anmerkung: Die Properties der Klassen `EventDispatcher` und `GameLoopPixi` finden Sie in der Grafik „WK Ball03 ClassModel00.png“.

2.2 Aufgabe 1: Modularisierung des Modells



Klassendiagramm von `app01`

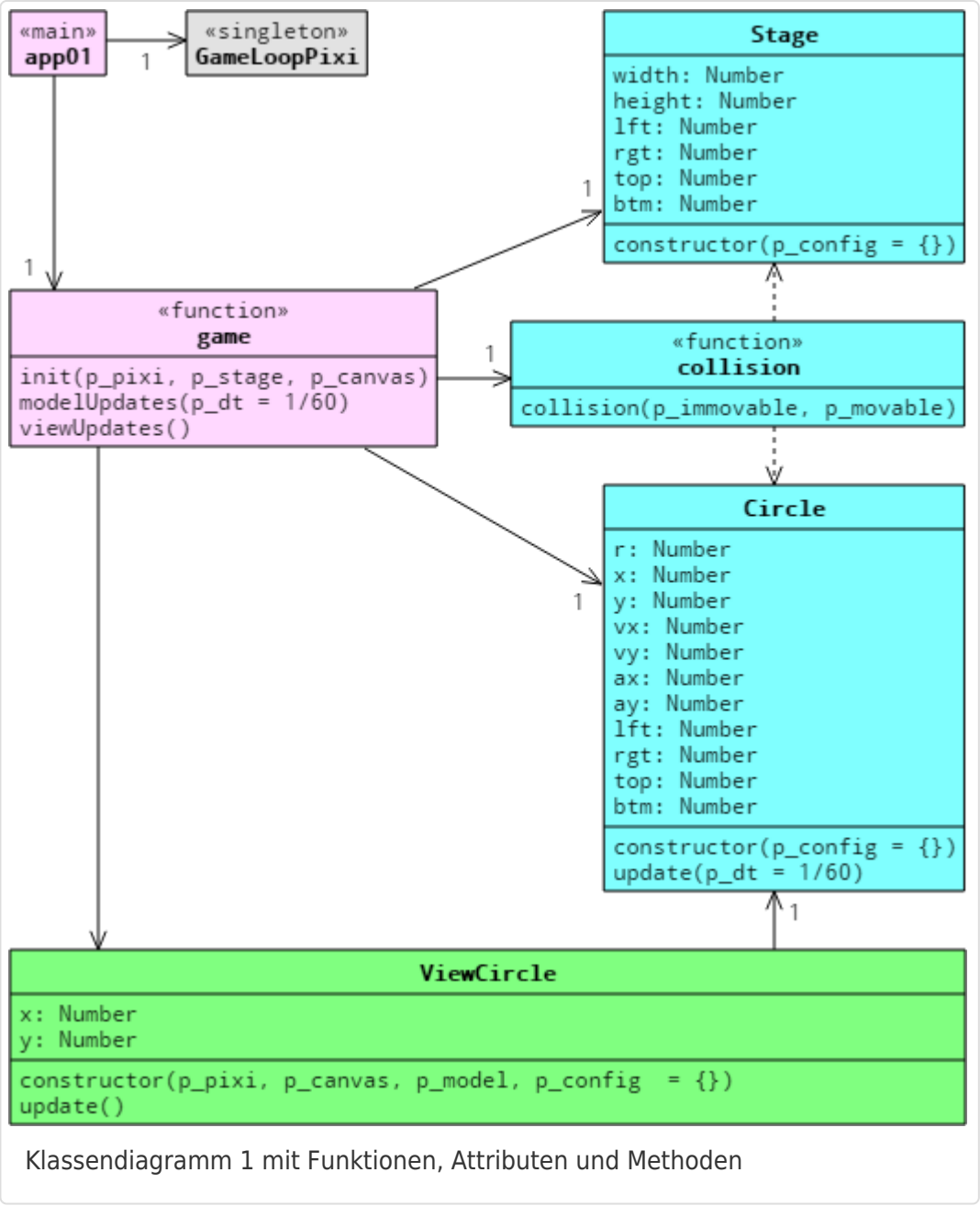
Ein gutes Modul ist nur für eine Aufgabe zuständig. Bislang ist das Game-Modul in dieser Hinsicht noch ziemlich schlecht.

2.2.1 Modularisierung der Modell-Komponenten

Im ersten Schritt werden die Modell-Komponenten „Bühne“, „Ball“ und „Kollissionserkennung und -behandlang“ in eigene Module ausgelagert. Rechts sehen Sie das zugehörige Klassendiagramm, bestehend aus insgesamt fünf Modulen:

`main`: Zuständig für die Initialisierung der Spielumgebung und das anschließende Starten des Spiels.
`game`: Zuständig für das Erzeugen der Spielelemente, den Benutzerinterkationen und der Spiellogik. Die einzelnen Teilaufgaben werden schrittweise an Hilfsmodule übertragen
Klasse `Stage`: Jedes Objekt dieser Klasse repräsentiert das Modell einer Spielbühne. Üblicherweise gibt es nur eine Spielbühne. Es gibt aber auch Spielsituationen mit mehreren Spielbügnen (z. B. wenn neben der Hauptbühne, die nur einen Ausschnitt der Spielwelt zeigt, eine Minimap existiert, die einen Überblick auf die Spielwelt gewährt).
Klasse `Circle`: Eine sehr gut wiederverwendbare Klasse, die für kreisförmige Objekte aller Art in einem Spiel zum Einsatz kommen kann. Normalerweise gibt es zahlreiche kreisförmige Objekte in einem Spiel.
`collision`: Ein Module, das eine Kollissionsfunktion bereit stellt. Diese Kollissionsfunktion ist für die Kollissionserkennung und -behandlung von (beweglichen) Kreisobjekten mit (unbeweglichen) Rändern der Bühne zuständig.

Im Ordner `js/app/game01` finden Sie für jedes dieser drei Module eine Datei. Im Ordner `js` finden Sie außerdem das Hauptmodul `app01.js`. Der notwendige Code, um die Spielumgebung bereitzustellen ist darin bereits denthalten. Ihre Aufgabe ist es, die Funktionen und Klassen, die in den anderen Dateien enthalten sind, zu implementieren. Welche Elemente die entsprechenden Module enthalten sollen, können Sie dem nachfolgenden Diagramm entnehmen.



Klassendiagramm 1 mit Funktionen, Attributen und Methoden

Gehen Sie folgendermaßen vor:

Kopieren Sie den Inhalt der Datei `game00.js` in die Datei `game01/game.js`, **ohne die darin enthaltenen Import-Befehle zu überschreiben.** (Wenn Sie jetzt `index01.html` im Browser öffnen, sollte der Ball über die Bühne flitzen – zumindest, wenn Sie nicht vergessen haben, `grunt watch` zu aktivieren :-))
Lagern Sie den Code der Kollissionserkennung und -behandlung aus der Funktion `modelUpdates` in die Funktion `collision` in der Datei `model/collision.js` aus. (*Achtung*: Der Code zur Berechnung der neuen Position und neuen Geschwindigkeit des Balls darf nicht in diese Datei ausgelagert werden.) Beachten Sie, dass die Funktion `collision` die Inputparameter `p_immovable` und `p_movable` besitzt. Die im kopierten Code enthaltenen Variablennamen `v_ball` und `v_stage` müssen entsprechend umbenannt werden.
Rufen Sie in der Funktion `modelUpdates` die neu erstellte (und von `game.js` auch schon importierte) Funktion `collision` geeignet auf. Bei einem Test der Web-Anwendung sollte sich der Ball wieder wie gewohnt über die Bühne bewegen.
Erstellen Sie nun die Klasse `Stage`. Gemäß dem obigen Diagramm hat sie sechs Attribute: `width`, `height`, `lft`, `rgt`, `top` und `btm`. Jedes dieser Attribute kann entweder als Wert geseichert oder mit

Hilfe von Getter- und Setter-Methoden berechnet werden. Die Entscheidung darüber, wie man vorgehen sollte, hängt davon ab, wie oft man lesend auf die Attribute zugreift und wie oft sie sich verändern. Bei einer Bühne ändern sie sich (nachdem die Bühne einmal initialisiert wurde) meist gar nicht oder nur selten (wenn z. B. ein Eventhandler nach dem Skalieren des Browserfensters die Bühnengröße automatisch anpasst). Gelesen werden die Attribute dagegen oft (bei jeder Kollissionserkennung und -behandlung eines beweglichen Objektes mit der Bühne). Daher bietet es sich hier an, alle Attribute in (internen) Objektvariablen zu speichern. Erstellen Sie also in der Klasse folgenden Konstruktor:

```
constructor()
{
  this.v_lft = 0;
  this.v_rgt = 0;
  this.v_top = 0;
  this.v_btm = 0;

  this.v_width  = 0;
  this.v_height = 0;
}
```

Dieser Konstruktor legt für jedes der gewünschten Attribute ein passendes objektinternes Attribut an und initialisiert es mit dem Wert . Da es in ECMAScript bislang noch keine privaten Klassenmitglieder gibt, kann man bei jedem Bühnenobjekt, das man später anlegt, auch auf diese Attribute zugreifen. Das sollten Sie aber tunlichst lassen. Die Namen wurde extra so gewählt, dass man erkennt, dass es sich eigentlich um private Elemente des Objekts handelt. Um lesend auf die Attribute zugreifen zu können, müssen Sie als nächstes für jedes Attribut eine passende Getter-Methode hinter dem Konstruktor einfügen:

```
get width()  { return this.v_width; }
get height() { return this.v_height; }

get lft() { return this.v_lft; }
get rgt() { return this.v_rgt; }
get top() { return this.v_top; }
get btm() { return this.v_btm; }
```

Nun können die Attribute des Bühnenobjektes noch nicht aktualisiert werden. Wenn die Werte nur bei Spielstart initialisiert werden, könnte man diese Aufgabe dem Konstruktor übertragen. Allerdings kann sich untergewissen Umständen die Bühnengröße im Laufe der Zeit ändern. Daher ist es sinvoll, auch geeignete Setter-Funktionen für die sechs Attribute zu definieren. Und hier zeigt sich der Vorteil, den der Einsatz von Setter-Funktionen mit sich bringt: Die Werte der Attribute hängen voneinander ab. Zum Beispiel ist die Bühnenbreite gleich dem Abstand zwischen rechtem und linken Rand. Bei der Änderung eines Attributwertes kann man nun automatisch davon abhängige Attributwerte ebenfalls modifizieren, sodass die sogenannten Integritätsbedingungen „Breite gleich Abstand zwischen linken und rechtem Rand“ sowie „Höche gleich Abstand zwischen unterem und oberend Rand“ immer erfüllt sind:

```
set width (p_w) { this.v_width  = p_w; this.v_rgt = this.v_lft + p_w; }
set height(p_h) { this.v_height = p_h; this.v_btm = this.v_top + p_h; }

set lft(p_x) { this.v_lft = p_x; this.v_width  = this.v_rgt - p_x; }
set rgt(p_x) { this.v_rgt = p_x; this.v_width  = p_x - this.v_lft; }
set top(p_y) { this.v_top = p_y; this.v_height = this.v_btm - p_y; }
set btm(p_y) { this.v_btm = p_y; this.v_height = p_y - this.v_top; }
```

Bislang enthalten alle Attribute eines Bühnenobjekt den Wert , nachdem es mit Hilfe des Befehls

```
 let buehne = new Stage()
```

erzeugt wurde. Das ist eine ziemlich unbrauchbare Bühne. Mann könnte es nut mittels

```
buehne.width = 500; buehne.height = 400;
```

an die gewünschten Gegebenheiten anpassen. Die Setterfunktionen würden dann nicht nur die Breite und Höhe, sondern auch gleich noch den rechten und den unteren Rand entsprechend anpassen. Dieses Vorgehen ist aber etwas umständlich. Schöner ist es, wenn man die Initialwerte dem neuen Objekt gleich bei dessen Erzeugung übergeben kann:

```
new Stage( { width: 500, height: 400 } )
```

Um dies zu erreichen, übergeben Sie dem Konstrouktor ein Initialisierungsobjekt, dessen Attribute mit Hilfe der zuvor definierten Setter-Funktionen dem Objekt noch während der Erzeugung zugewiesen werden. Ergänzen Sie im Konstruktor den Parameter

```
p_config = {}
```

und fügen Sie dann folgende Schleife[3] als letzten Befehl in den Konstruktor ein:

```
for (let l_key of Object.keys(p_config))
{
  this[l_key] = p_config[l_key];
}
```

Diese Schleife liest alle Attribute aus dem Initialisierungsobjekt `p_config` aus und weist sie dem neu erstellten Objekt `this` unterdemselben Namen als Attribut zu. Wenn das Initialisierungsobjekt korrekt ist und die Attribute mit den Namen `width`, `height`, `lft`, `rgt`, `top` bzw. `btm` enthält, werden die entsprechenden Setterfunktionen aufgerufen. *Achung:* Es wird nicht sichergestellt, dass nur diese sechst Attributwerte im neuen Bühnenobjekt gespeichert werden. Jedes Attribut, das in `p_config`

übergeben wird, wird ins Bühnenobjekt kopiert. Hier ist wieder Programmierdisziplin gefragt.
Ändern Sie nun die Datei `game.js` geeignet ab. Weisen Sie der Variablen `v_stage` ein neu erstelltes Bühnenobjekt zu: `new Stage()`. Da die Bühnengröße innerhalb der Funktion `init` an die Bildschirmgröß angepasst wird, ist es hier nicht notwendig, dem Konstruktor ein Initialisierungsobjekt zu übergeben. Wenn Sie nun die Anwendung wider starten (`grunt` nicht vergessen), sollte die Anwendung wieder funktionieren. (Die Klasse `Stage` wird bereits importiert.)
Nun ist es an der Zeit, die Korrektheit der Setter-Funktionen auszuprobieren: Ersetzen Sie in den ersten beiden Befehlen der Funktionen `init` die Befehle

```
v_stage.rgt = p_stage.width; v_stage.btm = p_stage.height;
```

durch

```
v_stage.width = p_stage.width; v_stage.height = p_stage.height;
```

. Die Anwendung sollte immer noch funktionieren! (*Vorsicht, Falle:* Wenn Sie vergessen `grunt` aufzurufen, funktioniert die App ebenfalls weiterhin, da die Bundle-Datei `app1.bundle.js` nicht neu erstellt wird.)
Erstellen Sie nun die Klasse `Circle` analog. Allerdings gelten nun hinsichtlich der Entscheidung „Attribut speichern oder berechnen“ andere Regeln. Ein Kreis ist beweglich. Entsprechend oft ändern sich seine Position und seine Geschwindigkeit. Auch die Beschleunigung kann sich regelmäßig ändern (z. B., wenn der Benutzer den Ball mit Hilfe eines Eingabegeräts wie Tastatur oder Maus steuert). Und selbst der Radius kann sich häufig ändern (z. B. weil der Ball aufgrund einer Aktion wächst oder schrumpft.)

Bei jeder Änderung die vier Ränder `lft`, `rgt`, `top` und `btm`, die es laut Datenmodell geben muss (siehe Grafik) neu zu berechnen und zu speichern, kostet zu viel Zeit. Besser ist es den Wert eines dieser Attribute nur dann zu berechnen, wenn dies bei der Kollissionserkennung und -behandlung benötigt wird. Schreiben Sie den Konstruktor also so, dass der Radius, die Positionsattribute, die Geschwindigkeitsattribute und Beschleunigungsattribute direkt gepeichert werden:

```
this. r = 0;
...
```

Die Gettermethoden für die Ränder der Bounding Box definieren Sie so, dass die jeweiligen Werte mit Hilfe von Mittelplunkt und Radius ermittelt werden:

```
get lft() { return this.x - this.r; }
...
```

Und die Setter-Methoden definieren Sie so, dass jeweils die [math]x[/math]- bzw. die [math]y[/math]-Position so angepasst wird, dass sich der entsprechende Rand an der gewünschten Position befindet:

```
set lft(p_x) { this.x = p_x + this.r; }
...
```

Nun müssen Sie noch die Methode `update(p_dt = 1/60)` definieren. Kopieren Sie dazu die ersten vier Zeilen aus der Methode `modelUpdates` und fügen Sie diese in die Methode ein. Beachten Sie, dass das Ball-Objekt jetzt nicht mehr `v_ball` heißt, sondern `this`.
Als nächstes sollten Sie die Datei `game.js` anpassen, so dass die neu erstellte Kalsse zum Einsatz kommt. Weisen Sie der Variablen `v_ball` ein neu erstelltes Kreisobjekt zu: `new Circle({r: 75, ...})`. Als Initialisierungsobjekt verwenden Sie das Objekt, das bislang als Kreisobjekt in der Datei `game.js` verwendet wird. Außerdem sollten Sie die ersten vier Anweisungen in der Funktion `modelUpdates` durch einen Methodenaufruf ersetzen:

```
v_ball.update(p_dt);
```

Das heißt, die Funktion `modelUpdates` erledigt ihre Aufgabe „Ball bewegen und Kollissionen erkennen und behndeln“ an sofort nur noch mittels zweier Hilfsfuktionen, der Funktion `collision` und der Methode `Circle.update`. Testen Sie, ob die Web-Anwendung noch funktioniert.
Zu guter Letzt sollten Sie noch eine Unsauberkeit in der Datei `collision.js` bereinigen. In der Kollissionsfunktion wird auf das kreisspezifische Attribut `r` zugegriffen. Doch nicht jedes bewegliche hat einen Radius. Seitdem Sie die Klasse `Circle` definiert haben, hat jedes Kreisobjekt zusätlich die Attribute `lft`, `rgt`, `top` und `btm`. Diese Attribute können nicht nur gelesen, sondern auch geändert werden. Schreiben Sie den Code im Rumpf der Funktion `colission` so um, dass das Attribut `r` gar nicht mehr verwendet wird.

2.2.2 Modularisierung der View-Komponenten

Erstellen Sie eine Klasse `ViewCircle` mit dem Konstruktor `constructor(p_pixi, p_canvas, p_model, p_config = {})` zur Erzeugung eines Viewobjekts für den Ball. In der Datei `game.js` wird dieses Objekt in der Variablen `v_ball_view` gespeichert. Das eigentliche Viewobjekt wird derzeit in der Initialisierungsfunktion `init` erzeugt und initialisiert. Mit Hilfe Ihrer Klasse sollte dazu nur noch ein Befehl notwendig sein:

```
v_ball_view
  = new ViewCircle(p_pixi, p_canvas, v_ball,
                   {
                     border:      5,
                     borderColor: 0xAAAAAA,
                     color:       0xFFAA00
                   }
                  );
```

Ein Viewobjekt muss zwei Attribute anbieten, da die Funktion `viewUpdates` n der Datei `game.js` darauf zugreift: `x` und y. Diese Attribute müssen einen Zugriff auf das zugehörige Pixi-Sprite gewähren, welches innerhalb des Viewobjekt gespeichert wird. Dazu sind je eine Getter- und eine Setter-Methode vorzüglich geeignet.

Wenn Sie in der Klasse `ViewCircle` auch noch eine Update-Methode implementieren – analog zur Update-Methode in der Klasse `MovableCircle` –, können Sie die Funktion `viewUpdates` sogar noch vereinfachen. Sie brauchen nur noch die neu definierte Updatemethode geeignet aufzurufen. (Anmerkung: Nachdem Sie dies gemacht haben, sind die beiden zuvor definierten Attribute `x` und y eigentlich überflüssig.)

3 Quellen

1. Programmierprinzipien
2. Qualität
3. ES6 In Depth: Iterators and the for-of loop

1. **Kowarschick (MMProg)**: Wolfgang Kowarschick; Vorlesung „Multimedia-Programmierung“; Hochschule: Hochschule Augsburg; Adresse: Augsburg; Web-Link; 2018; Quellengüte: 3 (Vorlesung)