

Lively Analyse

# Spezialgebiet Morph

Aaron Müller

Florian Eitel

29. November 2007

Hochschule Heilbronn

## Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>1</b>
<b>2</b>	<b>Grundaufbau von Morph</b>	<b>1</b>
2.1	Aufbau und Komponenten . . . . .	1
2.2	Von Widgets zu SVG . . . . .	2
2.3	Stile und Darstellung . . . . .	4
2.4	Submorph-Verwaltung . . . . .	5
<b>3</b>	<b>Spezialthemen</b>	<b>5</b>
3.1	Zeit und Timer . . . . .	5
3.2	Morphtransformation . . . . .	6
3.3	Event-Handling . . . . .	8
3.4	MVC-Konzept . . . . .	9
<b>4</b>	<b>Verbesserungen</b>	<b>11</b>
<b>5</b>	<b>Anhang</b>	<b>12</b>
5.1	Beispiel . . . . .	12
5.2	Generator . . . . .	15

Der Lively Kernel ist ein Research-Projekt, das von Sun Microsystems am 8.10.2007 als OpenSource-Projekt veröffentlicht wurde. Die Analyse des Lively Kernels fand im Rahmen der Vorlesung Modellierung von Softwarearchitekturen im WS07/08 statt.

## 1 Einführung

Das Morph-Konzept ist das Kernstück von Lively und wurde grob von dem Projekt Self<sup>1</sup> übernommen. Auf Ebene der Anwendungsprogrammierer entspricht ein Morph der JComponent von SWING in Java oder dem GtkWidget von GTK. Von Morph leiten sich alle SubMorphs wie ButtonMorph oder WindowMorph ab und verdecken die eigentliche Logik vor dem Programmierer.

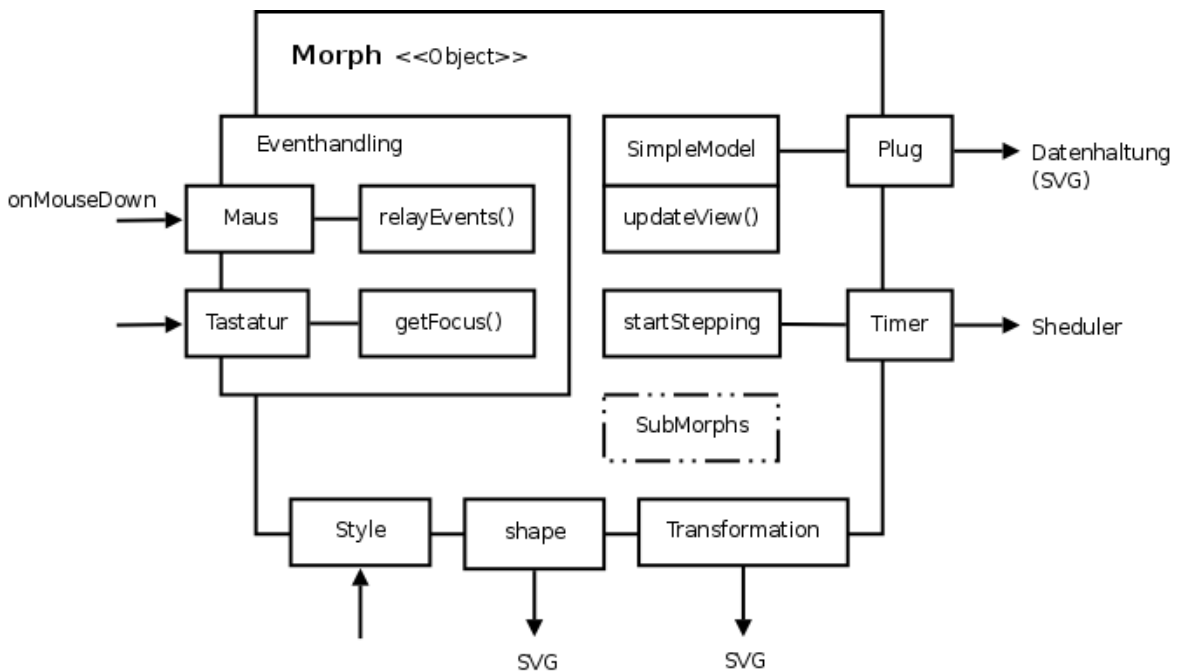
Auf SVG-Ebene entspricht ein Morph einem Knoten, also einem Container für Eigenschaften oder weitere Submorphs (mehr dazu im entsprechenden Kapitel).

Mit der Morph-Umgebung lassen sich Widgets (kleine Programme die einen Zustand anzeigen) oder ganze Anwendungen schreiben.

## 2 Grundaufbau von Morph

### 2.1 Aufbau und Komponenten

Der Morph selbst wird von keiner Klasse abgeleitet, er erhält nur die Eigenschaften die in *Object* definiert wurden. Subklassen wie ButtonMorph oder TextMorph erben natürlich von Morph.



<sup>1</sup><http://research.sun.com/self/>

Das Morph-Objekt hat ein Attribut *shape*, über das es direkt mit der Shape-Ebene (des lokalen Morphs) interagieren kann um spezielle Formen oder Eigenschaften zu definieren, die noch nicht vorgesehen wurden. Standard-Eigenschaften wie *setFill()* oder *setBorder()* können direkt im Kontext des Morphs angesprochen werden. Diese Methoden nutzen wiederum das Shape-Interface für die Manipulation. Um das Aussehen zu vereinheitlichen, wurden Stile eingeführt, die dem Morph diese Eigenschaften nachträglich geben können (siehe Kapitel Stile und Aussehen).

Um nicht nur Aussehen und Logik, sondern auch die Datenhaltung sauber zu trennen, enthält jeder Morph die Möglichkeit ein Model anzulegen oder ein fremdes zu nutzen. Damit die GUI (also die Morphs) bei einer Änderung der Daten die Anzeige auffrischen, hat jeder Morph die Methode *updateView()* um sich selbst zu aktualisieren (Observer-Pattern).

Ein Morph kann sich entweder selbst bewegen oder wird von seiner Umwelt bewegt (die Maus verschiebt ein Fenster). Hierzu gibt es mehrere Möglichkeiten der Transformation (verschieben, drehen, skalieren), die über SVG realisiert wurden.

Eine weitere Eigenschaft ist das Timer-Konzept. Hierzu kann eine Aktion die regelmäßig ausgeführt werden soll vom Morph in den Scheduler verlagert werden. Aus dem Morph heraus kann diese wiederkehrende Aktion gestartet und abgebrochen werden.

Um als vollwertiges Fenstersystem zu zählen, muss ein Widget auch auf externe Events reagieren können. Wenn der Morph auf ein bestimmtes Event das ihn betrifft (z.B. das Klicken mit der Maus auf einen Morph) reagieren will, kann er sich die Events zu sich selbst umleiten und Funktionalität zuweisen. Selbiges gilt bei Tastatureingaben.

Morphs können ineinander geschachtelt werden, um komplexere Morphs (wie z.B. einen Slider) zu erzeugen. Hierzu wird einfach im Konstruktor des Morphs die benötigten Morphs instantiiert und dem eigenen Morph hinzugefügt.

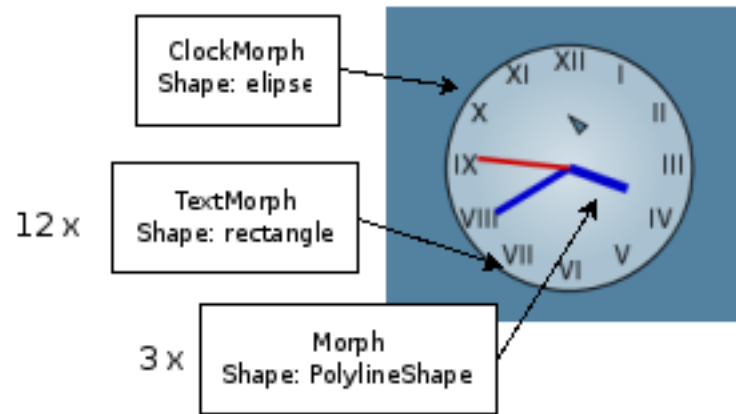
## 2.2 Von Widgets zu SVG

Nachfolgend wird beschrieben, wie ein neues Widget entsteht. Dazu greife ich die unten stehende Zeile aus dem ClockMorph heraus um daran am Beispiel zu erklären.

```

1  this . setNamedMorph (
2    " hours " ,
3    Morph . makeLine ( [ pt ( 0 , 0 ) , pt ( 0 , - radius * 0 . 5 ) ] , 4 , Color . blue )
4  );
```

Der Morph dient auf SVG-Ebene als eine Art Container in den man verschiedene Shapes verankern kann. Der Morph ist für das Verschieben, die Größenänderung, aber auch für die Submorphverwaltung zuständig. Der Morph hat ein eigenes Koordinatensystem über das alle Shapes und Submorphs adressiert werden. Wenn der Morph transformiert wird ändern sich so auch automatisch alle Unterobjekte. Die Transformation wird im entsprechenden Kapitel näher erläutert.



*Morph.makeLine* erstellt einen Morph, der durch ein *RectShape* repräsentiert wird. Jeder Morph wird mindestens durch einen Shapetype repräsentiert, damit man den Morph auswählen und mit ihm interagieren kann.

Außerdem wird im Beispiel des ClockMorph dem so erstellten Morph ein PolylineShape hinzugefügt. Man erkennt den Zusammenhang gut wenn man sich das generierte SVG anschaut:

```

1 <g lively:type="Morph" id="16" transform="matrix(0.539506 ,
2   0.841982 , -0.841982 , 0.539506 , 0 , 0)"
3   lively:property="hours">
4
5   <polyline points="0,0 0,-25" lively:type="polyline"
6     fill="none" stroke-width="4" stroke="rgb(0,0,204)" />
7
8   <g lively:type="Submorphs" />
9 </g>

```

*PolylineShape* ist eine Unterklasse von *Shape*. *Shape* wiederum von *DisplayObject*. *DisplayObject* sorgt für die Erstellung des eigentlichen SVG-Baumes. *Shape* und *DisplayObject* sind beides Mixins<sup>2</sup>. Als Beispiel wird *Shape* in *PolyShape* per Mixin eingefügt.

```
1 Shape.mixInto(PolylineShape);
```

Also gibt die Funktion *Morph.makeLine()* ein Morphobjekt zurück, welches dann für die Darstellungs des Zeigers der Uhr dient. Um dem Morph einen eindeutigen Namen zu geben, um auf ihn später wieder zu referenzieren oder ihn aus dem SVG-Baum zu holen, gibt es die *setNamedMorph()*-Funktion von *Morph*.

```
1 setNamedMorph("hours", Morph.makeLine(/* parameter */))
```

Dazu wird mit *setAttributeNS(Namespace.LIVELY, "property", name)* die Eigenschaft des Names zum SVG hinzugefügt. Also z.B. *lively:property="hours"*.

Der erstellte Morph wird dann in der Methode *domAddMorph()* eingefügt, je nach dem ob der Morph im Vordergrund oder Hintergrund sichtbar sein soll. Das Prinzip

<sup>2</sup><http://en.wikipedia.org/wiki/Mixin>

ist, das alle Submorphs in einem Array liegen. Dieses Array wird von vorne nach hinten durchgegangen und alle Submorphs dargestellt. D.h. wenn ein Morph im Vordergrund sein soll, muss er am Ende des Arrays hinzugefügt werden. Ansonsten wird er evtl. von einem anderen Morph überdeckt.

Im nachfolgenden wird `layoutChanged()` aufgerufen was dann evtl. Transformationen die im Cache liegen umwandelt und ins SVG schreibt. Darauf wollen wir nicht näher eingehen. Interessant ist nur noch wie eine Transformationen in SVG abgebildet wird:

```
1 <g lively:type="Morph" id="16" transform="matrix(0.539506,0.841982,
2   -0.841982, 0.539506, 0, 0)" lively:property="hours">
```

Die Transformation wird nur im Morph abgebildet. Alle Unterelemente bleiben unberührt da diese wieder auf den darüber liegenden Morph beziehen.

`ApplyTransform` wird rekursiv nach oben im der Morphherarchie weitergeleitet.

Jeder Morph hat als Attribute im SVG seinen `lively:type="Morph"`, seine `id="16"`, die Transformmatrix `transform="matrix(0.5..., 0.8..., -0.8..., 0.5..., 0, 0)"` und evtl. noch Attribute wie der Name `lively:property="hours"`.

Da Morph nur ein Container ist, werden alle Methoden wie `setFill` oder auch `setBorderColor` an die Shape weitergeleitet die das Morph repräsentiert.

Mit `setShape` kann der Shape verändert werden. Z.B. wenn ein Fenster von Viereckig auf Rund wechselt.

## 2.3 Stile und Darstellung

Im `ClockMorph` wird `this.linkToStyles(['clock'])`; ein neuer Stil für die Uhr gesetzt. Es wird anschließend in der Morphherarchie gesucht ob schon ein Stil mit diesem Namen angelegt wurde. Da `clock` neu ist wird der Aufruf rekursiv über `WorldMorph.current().styleNamed(name)`; weitergegeben. Hier sind drei DisplayThemes vorgeben die jeweils andere Stildefinitionen haben. (Siehe `Widgets.js` ab Zeile 2000). Exemplarisch `DisplayTheme: lively (Standard), Style: clock`:

```
1 clock: {
2   borderColor: Color.black,
3   borderWidth: 1,
4   fill: RadialGradient.makeCenteredGradient(
5     Color.primary.blue.lighter(2), Color.primary.blue.lighter())
6 }
```

Das `DisplayTheme` wird in der `WorldMorph.initialize` gesetzt:

`this.setDisplayTheme(this.displayThemes['primitive'])`; Zu guter Letzt wird über die Methode `applyStyle(spec)` der Style für die Clock übernommen.

Dieses Konzept trennt somit das eigentliche Aussehen vom Morph, ähnlich wie CSS und HTML. Das Aussehen - also der Stil - kann an einer Stelle global definiert und geändert werden und beeinflusst alle Morphs, die diesen Stil verwenden. Ein Umgestalten der Anzeige ist mit dieser Technik ohne großen Aufwand möglich.

## 2.4 Submorph-Verwaltung

```

Morph  addMorph(morph)
Morph  addMorphAt(morph, position)
Morph  addMorphFront(morph)
Morph  addMorphBack(morph)
Morph  addMorphFrontOrBack(m, front)
void    domAddMorph(m, isFront)
Morph  removeMorph(m)
void    removeAllMorphs()
Morph  hasSubmorphs()
Morph  remove()
void    withAllSubmorphsDo(func, argOrNull)
void    topSubmorph()
void    shutdown()

```

Man kann ein Submorph mit allen `addMorph*` Methoden hinzufügen. Wobei all diese Morphs auf `addMorphFrontOrBack(m, front)` zugreifen. `front` gibt dabei an, ob der Morph im Vorder- oder Hintergrund stehen soll (s.o.).

Die Methode `remove()` löst ein Submorph von seinem übergeordneten Morph ab und gibt es zurück.

`topSubmorph()` gibt den ersten Submorph im Array zurück.

`shutdown()` wird als eine Art Destruktor benutzt.

Jedes Widget wie z.B. `ClockMorph` ist ein Morph was wiederum Submorphs haben kann wie z.B. die Zeiger oder die Zahlen auf der Uhr.

## 3 Spezialthemen

### 3.1 Zeit und Timer

Wie man am `ClockMorph` sieht, ist es möglich eine Art Thread im Hintergrund laufen zu lassen. Im Fall von `ClockMorph` werden einfach immer die Zeiger weiterbewegt.

Dieser Mechanismus wird durch die Methode in der `ClockMorph` gestartet:

```

1 startSteppingScripts: function() {
2     this.startStepping(1000, "setHands"); // once per second
3 }

```

Die Methode `startStepping` erstellt eine Aktion die folgendermaßen aufgebaut ist:

```

1 var action = { actor: this, scriptName: scriptName,
2     argIfAny: argIfAny, stepTime: stepTime, ticks: 0 };

```

`this` ist der `ClockMorph`, `scriptName` ist in dem Fall die Methode die aufgerufen werden soll. Wir übergeben den Namen dieser Methode im Methodenaufruf. In unserem Fall ist das `setHands`. `stepTime` sind die übergeben 1000.

Diese Aktion wird anschließend in den SVG-Baum geschrieben:

```

1 <lively:action><![CDATA[{"scriptName": "setHands",
2   "stepTime": 1000, "ticks": 0}]]></lively:action>

```

Anschließend wird in der aktuellen Welt `startStepping` aufgerufen. Es wird dann, sollte es bereits eine vorherige Action in `scheduledActions` geben, diese gelöscht. Außerdem wird die Aktion in die Liste `scheduleActions` eingereiht. `scheduleActions` ist ein Array in dem Format `[nächster tick, Aktion]`. Nächster Tick beinhaltet die Zeit zu der die Action zum nächsten Mal aufgerufen wird. Zum Abschluss wird die Methode `kickstartMainLoop` aufgerufen:

```

1 kickstartMainLoop: function() {
2   this.mainLoop = window.setTimeout(this.mainLoopFunc, 10);
3 }

```

`window.setTimeout` ist eine interne Javascript Methode die in diesem Fall mit 10 Milisekunden Verzögerung aufgerufen wird. Für die Variable `mainLoopFunc` ist die Methode `doOneCycle` definiert, was den Scheduler repräsentiert.

Hier wird dann für alle Funktionen, die an der Reihe sind, die vorher festgelegt Methode aufgerufen. In unserem Beispiel ist dies `setHands`

```

1 var func = action.actor[action.scriptName];
2 func.call(action.actor, action.argIfAny);

```

Wenn die Aktion regelmäßig auftritt, wird sie wieder in die Liste der `scheduleActions` eingereiht. Anschließend wird von allen wartenden Actions die Zeit ermittelt bei der eine Action das nächste mal die Aufmerksamkeit erfordert.

Nach Beendigung eines Ablaufes wird, sollte es noch wartende Aktionen geben, wieder die `doOneCycle` aufgerufen. Dazu wird wieder `window.setTimeout` verwendet und als Verzögerung die Zeit bis zur nächsten Aktion eingesetzt.

```

1 this.mainLoop = window.setTimeout(this.mainLoopFunc,
2   timeOfNextStep - this.lastStepTime);

```

## 3.2 Morphtransformation

Jeder Morph kann transformiert (verschoben), rotiert und skaliert werden. Dies ist für ein Fenstersystem essenziell und wird auch oft angewandt. Anhand des `ClockMorph`-Beispiels wird das Prinzip und die verwendeten Techniken erläutert.

Auf Morph-Ebene kann auf Funktionen `setRotation()`, `setTranslation()` und `setScale()` zugegriffen werden, die am Uhr-Beispiel den Stundenzeiger korrekt rotiert:

```

1 this.getNamedMorph('hours').setRotation(hour/12*2*Math.PI);

```

Der Morph mit dem eindeutigen Namen `hours` wird über die `setRotation` entsprechend rotiert. Auf Morph-Ebene ist hier nun schon alles erledigt. Schauen wir in die Implementierung der `setRotation()`-Funktion in `Morph`, wird klar, dass die eigentliche Rotation nur weiterdelegiert wird.

```

1 setRotation: function(theta) {
2     this.rotation = theta;
3     this.cachedTransform = null;
4 }.wrap(Morph.onLayoutChange('rotation')),

```

Die `wrap()`-Methode kommt von Prototype und führt aspektorientierte Programmierung ein. Sie dient als Wrapper für Änderungen am SVG-Baum. Wenn beispielsweise ein Morph rotiert wird, werden über den Aufruf von `onLayoutChange()` die Originalmethode aufgerufen und anschließend der SVG-Baum aktualisiert. Es wird also der Aspekt der Auffrischung des SVG-Baumes zur Rotationsmethode hinzugefügt.

```

1 onLayoutChange: function(fieldName) {
2     return function(/* arguments */) {
3         this.changed();
4         var args = $A(arguments);
5         var proceed = args.shift();
6         var result = proceed.apply(this, args);
7
8         // SVG und Anzeige auffrischen
9         this.recordChange(fieldName);
10        this.layoutChanged();
11        return result;
12    }
13 }

```

Die `onLayoutChange()`-Methode nimmt alle übergebenen Argumente entgegen (der erste Parameter entspricht der Originalfunktion die in `proceed` abgelegt wird). Die Originalfunktion wird mit den übrigen Parametern evaluiert und anschließend der SVG-Baum aktualisiert. So lässt sich das Aktualisieren des SVG-Baumes von den Transformationsfunktionen abkapseln und später gegebenenfalls durch eine andere Aktualisierungsfunktion austauschen. Die eigentlichen Transformationen wie rotieren, verschieben oder skalieren übernehmen die DOM-SVG Methoden die im Browser implementiert sind und für die JavaScript ein Interface anbietet. Verwendet werden die Objekte *SVGMatrix* und *SVGTransform*.

Das Transform-Objekt ist eine Instanz von `SVGTransform`, welches mit `Canvas.createSVGTransform()` erzeugt wird. Es stehen nun einige Möglichkeiten für die Transformation zur Verfügung. Verwendet werden hauptsächlich folgende Methoden:

```

SVGMatrix  getMatrix()
void       setMatrix(SVGMatrix matrix)
void       setRotate(float angle, float cx, float cy)
void       setScale(float sx, float sy)
void       setTranslate(float tx, float ty)

```

Mit der ersten Methode lässt sich dann ein `SVGMatrix`-Objekt holen, auf dem hauptsächlich folgende Methoden angewendet werden können:

```

SVGMatrix translate(float x, float y)
SVGMatrix rotate(float angle)
SVGMatrix scale(float scaleFactor)

```

Diese Grundmethoden werden in der Transform-Klasse gekapselt und auf einmal ausgeführt.

```

1 matrix.translate(delta.x, delta.y).
2   rotate(angleInRadians.toDegrees()).
3   scale(scale);

```

Die Nutzung von SVG hat hier einen entscheidenden Vorteil: Alle Transformationen übernimmt der Browser und müssen nicht mehr im Lively-Kernel selbst untergebracht werden. Dies spart viele Zeilen Code.

### 3.3 Event-Handling

Lively stützt sich beim Eventhandling wieder auf schon vorhandenes. Hier auf die Event-Funktionen von Javascript.

#### 3.3.1 Maus

Um auf Events zu reagieren, müssen die Events auf Funktionen umgeleitet werden. Im Beispiel werden die Events *onMouseDown* und *onMouseUp* eines *ButtonMorphs* mit der Morph-Methode *relayMouseEvents()* an die Funktionen *decreaseCounter()* und *decreaseCounterUp()* umgeleitet.

```

1 main.addMorph(m = ButtonMorph(Rectangle(10, 60, 170, 30)));
2 m.relayMouseEvents(this, {
3   onMouseDown: "decreaseCounter",
4   onMouseUp: "decreaseCounterUp"
5 });
6 // ...
7 decreaseCounter: function(evt) {
8   this.model.setA(this.model.getA() - 1);
9 },
10 decreaseCounterUp: function(evt) {}

```

Der erste Parameter (in diesem Fall *this* für das aktuelle Widget) gibt den Kontext an, in dem das Event abgefangen werden soll. Der zweite Parameter stellt das Mapping von Event zu Funktion her.

Wenn ein Morph Events entgegennehmen will, aber von einem anderen Morph überdeckt wird, muss der übergeordnete Morph alle Events ignorieren. Dies kann man mit der Methode *ignoreEvents()* veranlassen.

#### 3.3.2 Tastatur

Bei Tastatureingaben verhält es sich etwas anders. Hier muss der Morph den Tastaturfokus zuerst auf sich lenken um dann erst (wie bei der Maus) das Mapping Event nach

Funktion herzustellen. In der vom Tastaturevent aufgerufenen Funktion kann dann auf die in Javascript definierten Konstanten für spezielle Tasten wie *KEY\_ESC* oder *KEY\_ENTER* zurückgegriffen werden.

### 3.4 MVC-Konzept

Lively hat ein interessantes MVC-Konzept umgesetzt, das nicht dem Originalpattern von der GoF entspricht, aber dennoch gut funktioniert. Die meisten Widgets werden von der Klasse *Model* abgeleitet, welche dem Widget bereits eine grundlegende Verhaltensweise vererbt. Die Klasse *Model* übernimmt hierbei die Persistenzschicht für die Datenhaltung auf SVG-Ebene.

Die Klasse *SimpleModel* dagegen stellt die Schnittstelle zwischen Model und dem/-den Morphs dar. Sie erstellt automatisch Zugriffsmethoden für die Daten. Zur Veranschaulichung verwende ich den Testmorph *CountMorph* und das TestWidget aus dem Anhang.

#### 3.4.1 Das Modell im Morph

Morphs können Daten enthalten. Bei einem Slider wäre das die aktuelle Position des Schiebereglers, im *CountMorph*-Beispiel ist es die zu incrementierende Zahl. Im Beispiel wird im Konstruktor ein neues *SimpleModel* angelegt und dem Attribut „model“ zugewiesen. Als Parameter werden die Namen der „Speicherplätze“ im Modell angegeben, hier nur *Count*.

```
1 var model = new SimpleModel( null , "Count" );
```

Diese Speicherplätze macht SimpleModel über Getter/Setter verfügbar (hier *getCount()* und *setCount(value)*), diese müssen nicht mehr erstellt werden. Definiert ist die Konstruktion der Methoden folgendermaßen:

```
1 addVariable: function( varName , initialValue ) {
2     // functional programming is fun!
3     this[ varName ] = initialValue;
4     this[ getter( varName ) ] = function( name ) {
5         return function() { return this[ name ]; }
6     }( varName );
7
8     this[ setter( varName ) ] = function( name ) {
9         return function( newValue , v ) {
10             this[ name ] = newValue;
11             this.changed( getter( name ) , v );
12         }
13     }( varName );
14 },
```

Da in JavaScript jedes Objekt gleichzeitig als Hashmap verwendet und angesprochen werden kann, können so zur Laufzeit neue Methoden im Modell angelegt werden. Die

Funktionen `getter()` und `setter()` formatieren lediglich den String `varName` in Form von `[set|get]varName`.

Interessant wird es in der nächsten Zeile. Hier wird das Modell im DOM-Baum verankert und verfügbar gemacht. Über `modelPlug` kann nachfolgend immer auf das Modell und deren Getter/Setter mit den Methoden `getModelValue()` und `setModelValue()` zugegriffen werden.

```
1 this.modelPlug = this.addChildElement(model.makePlug());
```

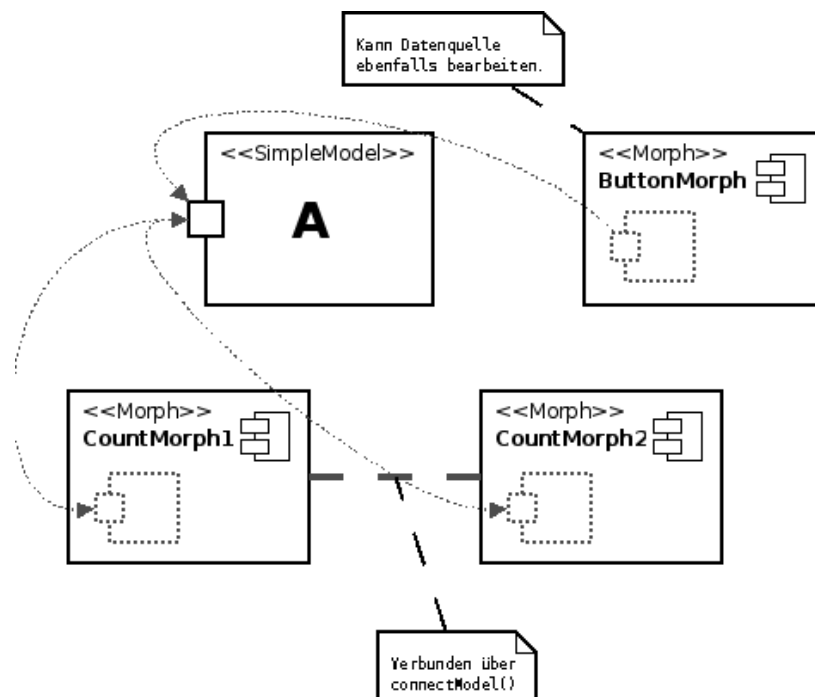
Um beispielsweise auf den Inhalt von `Count` zuzugreifen, verwendet man folgenden Aufruf:

```
1 this.getModelValue('getCount', 0);
```

Die Methode kann deshalb direkt mit `this` angesprochen werden, da die beiden Methoden in der Klasse `Morph` definiert wurden und diese auf den `modelPlug` zugreifen. (Der zweite Parameter der Methode ist ein Defaultwert, der zurückgegeben wird wenn der `modelPlug` nicht gesetzt ist oder der "Speicherplatz" `Count` nicht definiert ist. In fast allen untersuchten Morphs werden Wrapper für diese Methoden geschrieben, um einfacher darauf zuzugreifen.

### 3.4.2 Modell-Sharing

Mehrere Morphs können sich das gleiche Modell teilen um den gleichen Datenbestand auszulesen oder zu verändern. Hierzu stellt die Morph-Klasse die Methode `connectModel()` bereit. Im Beispiel greifen zwei `CountMorphs` auf den selben Datenbestand `A` zu.



Der Morph *CountMorph* besitzt durch das Modell die Methoden `getCount()` und `setCount()` über die er selbst den Zähler auslesen und setzen kann (siehe vorhergehender Abschnitt). Ziel ist es nun, die beiden Methoden der *CountMorphs* auf den selben Datenbestand anzuwenden.

```

1 var model = new SimpleModel( null, 'A' );
2 var m;
3
4 main.addMorph(m = CountMorph( Rectangle( 10, 10, 80, 40 ) ));
5 m.connectModel( { model: model, getCount: "getA", setCount: "setA" } );
6 main.addMorph(m = CountMorph( Rectangle( 100, 10, 80, 40 ) ));
7 m.connectModel( { model: model, getCount: "getA", setCount: "setA" } );

```

Zuerst wird wieder ein neues *SimpleModel* mit dem “Speicherplatz” A angelegt. Dies soll nun als zentrale Datenquelle für beide Morphs verwendet werden. Dazu wird die *connectModel()*-Methode der jeweiligen Morph-Klasse verwendet. Der erste Parameter verweist auf das zu verwendende Modell, die restlichen Parameter erzeugen die Umleitungen für die Getter/Setter-Methoden.

Wenn sich nun ein *CountMorph* seiner *setCount()*-Methode bedient, wird *setA()* des neuen Modells verwendet um die Daten in A zu speichern. Selbiges gilt für *getCount()*. Im Beispiel zeigen jeweils *getCount()* auf *getA()* und *setCount()* auf *setA()*. Damit alle beteiligten Morphs von der Änderung des Datenbestandes mitbekommen, muss im Morph die Methode *updateView()* definiert werden, wie immer aufgerufen wird, wenn sich etwas ändert.

```

1 updateView: function( aspect, controller ) {
2     var p = this.modelPlug;
3     if ( p ) {
4         if ( aspect == p.getCount || aspect == 'all' )
5             // Anzeige lokal aendern
6             this.changeAppearanceFor( this.getCount () );
7         return this.getCount ();
8     }
9 },

```

Wie im oben dargestellten Diagramm schon angedeutet, können Modelle auch von ausserhalb angesprochen werden. Sie müssen also nicht zwangsläufig in Morphs mit *modelPlug* eingebunden werden. Die Komponente benötigt nur eine Referenz auf das Modell, um auf *getA()* bzw. *setA()* zuzugreifen. Dies ist im Beispiel mit einem Button realisiert, der den Wert von A um 1 verringert.

## 4 Verbesserungen

Javascript ist eine sehr flexible Sprache, die für eine solche Implementierung gut gewählt wurde. Allerdings verleitet diese Sprache auch zum schlampigen Programmieren. Der Code selbst hat sehr starke Abhängigkeiten. Der Versuch von uns die Klasse *Morph* in einer extra Datei auszulagern (oder zimmindest alle Methoden der Klasse an eine

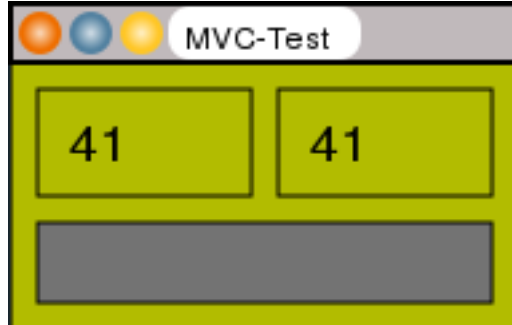
Stelle zu schreiben) ist uns nicht gelungen. Einzelne Teile der Klasse Morph benötigen einzelne Teile anderer Klassen, die aber selbst wieder Teile von Morph benötigen (als ob man zwei Käbme ineinanderstecken würde). Eine Kapselung der Klassen würde die Übersicht und der Komplexität verbessern.

Man könnte die Performance an vielen Stellen verbessern. Oft wird zu viel neu gezeichnet, was eigentlich nicht erneuert werden muss. Eine weitere Möglichkeit wäre die SVG-Engine auszutauschen. SVG gibt es in den Browsern (zumindest in den guten) schon länger, aber wurde nie wirklich auf Performance getrimmt, da es noch viel zu wenig genutzt wird. Man könnte versuchen die SVG-Engine eines Vektorgrafikprogrammes für das Rendering zu verwenden. Voraussetzung dafür ist natürlich, dass das Vektorprogramm eine Schnittstelle zu Javascript anbietet.

## 5 Anhang

### 5.1 Beispiel

Um das Konzept der Morphs, Widgets und deren Komponenten besser zu verstehen, haben wir ein dokumentiertes Beispiel geschrieben, welches die Hauptbestandteile eines Morphs implementiert. es wird ein Morph *CountMorph* angelegt, welcher seine Daten (den aktuellen Zählerstand) in einem eigenen Modell speichert. Bei Klick auf den Morph wird der Zählerstand um 1 inkrementiert. Um das Prinzip des Stepping-Timers zu demonstrieren, inkrementiert die Komponente zusätzlich den Zählerstand in regelmäßigen Abständen.



Das TestWidget erstellt zwei Instanzen des CountMorphs und verbindet die Datenquelle. Das steppingScript wird für eine der Morphs gestartet, für den anderen nicht. Ein weiterer Button manipuliert den Zählerstand.

```

1
2 /**
3  * CountMorph
4  *
5  * Ein simpler Morph um die die Grundlagen und das MVC-Konzept zu
6  * demonstrieren. Der Morph stellt einen Counter dar, der bei einem
7  * Klick im 1 inkrementiert.
8  */
9 CountMorph = HostClass.create('CountMorph', Morph);
10 Object.category(CountMorph.prototype, "core", function() { return {

```

```

11  /**
12  * Konstruktor der automatisch aufgerufen wird. Es wird zuerst
13  * der Konstruktor von Morph aufgerufen (aehnlich wie super() in
14  * Java) und danach ein TextMorph (als Label) erstellt. Eine
15  * neue Instanz von SimpleModel wird erstellt und als Datenhaltung
16  * getCount() und setCount() angelegt. Anschliessend wird ein neuer
17  * "plug" in den SVG-Baum eingehaengt. Ueber diesen lassen sich
18  * nachfolgend auf das Model zugreifen.
19  *
20  * @param Rectangle initialBounds Koordinaten fuer die Position
21  * @return CountMorph Der neu erstellte Morph
22  */
23  initialize: function(initialBounds) {
24      CountMorph.superClass.initialize.call(this, initialBounds, "rect");
25      var counter = new TextMorph(Rectangle(0, 0, 50, 20), "0").beLabel();
26      counter.ignoreEvents();
27      counter.setFontSize(20);
28
29      var model = new SimpleModel(null, "Count");
30      this.modelPlug = this.addChildElement(model.makePlug());
31
32      // Morph einbinden und die Count-Variable initialisieren
33      this.setNamedMorph("counter", counter);
34      this.setModelValue('setCount', 0);
35      this.changeAppearanceFor(0);
36      return this;
37  },
38
39  /**
40  * Diese Methode wird nur lokal ausgefuehrt, wenn sich etwas am Model
41  * aendert. Es wird nur der Text auf dem Label auf den aktuell uebergebenen
42  * Wert gesetzt.
43  *
44  * @param int value Der zu setzende Wert
45  */
46  changeAppearanceFor: function(value) {
47      this.counter.setTextString(""+value);
48  },
49
50  /**
51  * Wird vom Controller aufgerufen, wann immer sich etwas am Model aendert.
52  * Es wird zudem ueberprueft ob der "plug" schon gesetzt ist und ob
53  * das Updaten der grafischen Oberflaeche diesen Morph betrifft.
54  *
55  * @param String aspect Den Bereich, der erneuert werden soll
56  * @param controller Die Controller-Instanz
57  * @return int Den Wert des Models
58  */
59  updateView: function(aspect, controller) {
60      var p = this.modelPlug;
61      if (p) {
62          if (aspect == p.getCount || aspect == 'all')
63              this.changeAppearanceFor(this.getCount());
64          return this.getCount();
65      }
66  },
67
68  startSteppingScripts: function() {
69      this.startStepping(1000, "count");
70      console.log("counting ...");
71  },
72
73  /**
74  * Getter- und Setter-Methoden, die fuer das Model notwendig sind. Es wird
75  * direkt das Model geaendert, es befinden sich keine lokalen Daten im Morph.
76  */

```

```

77     getCount: function() {
78         if (this.modelPlug) return this.getModelValue('getCount', 0);
79     },
80     setCount: function(value) {
81         if (this.modelPlug) return this.setModelValue('setCount', value);
82     },
83
84     /**
85      * Mausfunktionen fuer das Eventhandling. handlesMouseDown muss true
86      * zurueckgeben um die Events abzufangen. Da das Objekt nicht verschoben
87      * werden soll, muss die vererbte Funktion aus Morph mit einer leeren
88      * Funktion ueberschrieben werden. onMouseDown() erhoehrt den Wert des
89      * Models um 1 und erneuert die Anzeige.
90      */
91     handlesMouseDown: function(evt) { return true; },
92     onMouseMove: function(evt) { },
93     onMouseDown: function(evt) {
94         this.count();
95     },
96
97     /**
98      * Methode um das Modell um eins zu erhoehen. Diese wird vom steppingScript
99      * und von den Maushandlern verwendet.
100     */
101     count: function() {
102         this.setCount(this.getCount()+1);
103         this.changeAppearanceFor(this.getCount());
104     }
105 }});
106
107
108 /**
109  * Eine simple Anwendung um den erstellten Morph zu testen. Es wird demonstriert
110  * wie sich zwei Morphs das selbe Model teilen koennen und wie ein "externer"
111  * Morph ebenfalls auf dieses Modell zugreifen und die Daten veraendern kann.
112  */
113 TestWidget = Class.extend(Model);
114 Object.extend(TestWidget.prototype, {
115     /**
116      * Konstruktor fuer das TestWidget. Er erstellt zwei CountMorphs die
117      * miteinander verbunden werden. Ein weiterer ButtonMorph greift ebenfalls
118      * auf das Modell der beiden Zaehler zu.
119      *
120      * @param Canvas world           Hauptfenster
121      * @param Rectangle location     Die Koordinaten an denen das Fenster
122      *                                platiert werden soll.
123      * @return Cancas                Das Hauptfenster mit eingefuegtem Morph
124      */
125     initialize: function(world, location) {
126         // Der Konstruktor der Klasse Model wird aufgerufen und ein neues
127         // Panel angelegt.
128         TestWidget.superClass.initialize.call(this);
129         var main = PanelMorph(pt(190, 100));
130
131         // Ein neues Model mit dem "Speicherplatz" A (dieser erhaelt dann setA()
132         // und getA() Accessoren.
133         var model = new SimpleModel(null, 'A');
134         this.model = model;
135         main.connectModel({model: model});
136
137         // Zwei CountMorphs werden angelegt und die Methoden getCount() und setCount()
138         // auf die gemeinsame Datenquelle A gemappt.
139         var m;
140         main.addMorph(m = CountMorph(Rectangle(10, 10, 80, 40)));
141         m.connectModel({model: model, getCount: "getA", setCount: "setA"});
142         main.addMorph(m = CountMorph(Rectangle(100, 10, 80, 40)));

```

```

143     m.connectModel({model: model, getCount: "getA", setCount: "setA"});
144
145     // Den Couter starten
146     m.startSteppingScripts();
147
148     // Ein ButtonMorph bei dem die onMouseDown()-Funktion mit relayMouseEvents()
149     // auf die Funktion decreaseCounter() umgeleitet wird..
150     main.addMorph(m = ButtonMorph(Rectangle(10, 60, 170, 30)));
151     m.relayMouseEvents(this, {
152         onMouseDown: "decreaseCounter",
153         onMouseUp: "decreaseCounterUp"
154     });
155
156     return world.addMorphAt(WindowMorph(main, "MVC-Test"), location);
157 },
158
159 /**
160  * Durch die Umleitung des Mausevents kann hier eine Aktion fuer den ButtonMorph
161  * bei Klick auf den Button gesetzt werden. Dabei wird der Inhalt des Models A
162  * um 1 dekrementiert. Der onMouseUp-Event muss ueberschrieben werden, um das
163  * Mausevent wieder freizugeben (wenn dies nicht gemacht wird, werden alle
164  * zukuenftigen onMouseDown-Events auf diese Funktion umgeleitet.
165  */
166     decreaseCounter: function(evt) {
167         this.model.setA(this.model.getA() - 1);
168     },
169     decreaseCounterUp: function(evt) {}
170 });
171
172 console.log("loaded CustomWidgets.js");

```

## 5.2 Generator

Für die grobe Analyse und für die Programmierung der Widgets benötigten wir eine API um schnell zu sehen welche Attribute und Methoden eine Klasse besitzt. Da der Code allein leider sehr zerpfückt ist, eignet er sich weniger als schnelles Nachschlagewerkzeug.

Deshalb haben wir die vorhandenen Funktionen des internen Klassenbrowsers genutzt um uns den kompletten Code zur Laufzeit auszugeben. Diese Ausgabe haben wir mit einem kleinen Ruby-Script zuerst in XML umgewandelt und anschließend daraus HTML-Seiten erzeugt. (Ursprünglich wollten wir die XML-Datei für die Generierung von Beziehungen und Abhängigkeiten von Klassen verwenden, doch dies erwies sich als zu zeitaufwendig).

# ClockMorph

## Methods

- [attrNames](#)
- [become](#)
- [bind](#)
- [bindAsEventListener](#)
- [curry](#)
- [defer](#)
- [delay](#)
- [fnkNames](#)
- [functionNames](#)
- [initialize](#)
- [localFunctionNames](#)
- [localattrNames](#)
- [localfnkNames](#)
- [logCalls](#)
- [logErrors](#)
- [makeNewFace](#)
- [methodize](#)
- [mixInto](#)
- [reshape](#)
- [setHands](#)
- [startSteppingScripts](#)
- [wrap](#)

## Code

### attrNames

## Attributes

ATTRIBUTE\_NODE  
 CDATA\_SECTION\_  
 COMMENT\_NODE  
 DOCUMENT\_FRAG  
 DOCUMENT\_NODE  
 DOCUMENT\_POSI  
 DOCUMENT\_POSI  
 DOCUMENT\_POSI  
 DOCUMENT\_POSI  
 DOCUMENT\_POSI  
 DOCUMENT\_POSI  
 DOCUMENT\_POSI  
 DOCUMENT\_POSI  
 DOCUMENT\_TYPE  
 ELEMENT\_NODE  
 ENTITY\_NODE  
 ENTITY\_REFERENC  
 NOTATION\_NODE  
 PROCESSING\_INSP  
 TEXT\_NODE  
 attributes  
 baseURI  
 childNodes  
 className

Einzelne Methoden einer Klassen können aufgeklappt werden um den eigentlichen Code der Funktion zu sehen.

## onMouseMove

## onMouseUp

```
        ButtonMorph.prototype.onMouseUp = function (evt) {  
var newValue = this.isToggle() ? !this.getValue() : false;  
this.setValue(newValue);  
this.changeAppearanceFor(newValue);  
}
```

## restorePersistentState