CHRISTIAN-ALBRECHTS-UNIVERSITÄT ZU KIEL

Studienarbeit

# KEV - KIELER Environment Visualization

## Beschreibung einer Zuordnung von Simulationsdaten und SVG-Graphiken

Stephan Knauer

1. Juli 2010


Institut für Informatik
Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme

Prof. Dr. Reinhard von Hanxleden

betreut durch:
Dipl. Inf. Hauke Fuhrmann

## Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

_____

**Zusammenfassung**

Der Wunsch, den aktuellen Zustand von Soft- und Hardwaresystemen visuell darzustellen, ist die Grundidee von KIELER Environment Visualization (KEV). Programme, um spezielle Hardware zu visualisieren und so den Zustand eines Systems anzuzeigen, gibt es in zahlreichen Varianten. Ein allgemeines System zur Darstellung von Simulationsdaten gibt es bisher jedoch nicht. Die Grundidee von KEV ist daher diese Lücke zu schließen und eine allgemeine Schnittstelle zur Visualisierung anzubieten. Hierzu verwendet KEV zur Visualisierung Scalable Vector Graphics (SVG)-Graphiken. Bei SVG-Graphiken handelt es sich um ein Format für Vektorgraphiken nach der Spezifikation des Word Wide Web Consortium (W3C). Als Austauschformat für Daten zwischen einer Simulation und KEV verwendet KEV das JavaScript Object Notation (JSON)-Format [8]. Mittels des Eclipse-Plugin-Mechanismus können auf einfache Art und Weise die unterschiedlichsten Hard- und Softwaresysteme mit KEV verbunden werden und mittels JSON-Strings kommunizieren. Um den Zustand eines Systems zu visualisieren, muss nun noch eine SVG-Graphik des Systems erstellt und mit einer entsprechende Mapping-Datei verknüpft werden. Die Mapping-Datei beschreibt dabei das Verhalten der SVG-Graphik auf bestimmte Eingaben von der Simulation. Damit kann KEV den momentanen Zustand von Hard- und Softwaresystemen darstellen und durch sich kontinuierlich verändernde Daten den Ablauf dieser Systeme animieren.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| **EMF** | Eclipse Modeling Framework |
| **IBM** | International Business Machines |
| **JAVA** | objektorientierte Programmiersprache |
| **JSON** | JavaScript Object Notation |
| **KEV** | KIELER Environment Visualization |
| **KIEL** | Kiel Integrated Environment for Layout |
| **KIELER** | Kiel Integrated Environment for Layout for the Eclipse Rich client platform |
| **KIEM** | KIELER Execution Manager |
| **RCA** | Rich Client Application |
| **RCP** | Rich Client Platform |
| **SVG** | Scalable Vector Graphics |
| **W3C** | Word Wide Web Consortium |
| **XML** | eXtensible Markup Language |
| **XSD** | XML Schema Definition |

# 1 Einführung

Um einen Einblick in ein geschlossenes System (z.B. eine Fahrstuhlsteuerung) zu bekommen, kann man es zum Einen auseinander bauen um es zu untersuchen, zum Anderen kann man es aber auch simulieren um das Verhalten zu untersuchen. Letzteres ist günstiger und effizienter, da es ein reales System je nach Größe nicht immer und überall zu Testzwecken zur Verfügung steht. Wenn man das System jedoch einmalig in Form einer geeigneten Graphik darstellt und Regeln angibt, die das Verhalten des Systems für bestimmte Eingabewerte definieren, so ist es möglich eben dieses System jederzeit an nahezu jedem beliebigen Ort zu testen. Es stellt sich daher die Frage, wie man ein System allgemein beschreiben und darstellen kann um anschließend das Verhalten visuell darstellen zu können. Genau an diesem Punkt setzt dieses Arbeit an und stellt mit dem KEV-Projekt eine Möglichkeit vor die eben geschilderten Probleme zu lösen. Eine weitere Motivation dieser Arbeit stellt die Tatsache dar, dass Lösungen dieser Art bislang nur von kommerziellen Tools (z.B. Adobe Flash [1]) bekannt sind.

Allgemein lässt sich sagen, dass KEV ein System in Form einer geeigneten Graphik darstellt und diese Graphik durch eine Reihe definierter Regeln verändert. Diese Veränderung geschieht mit Hilfe einer Zuordnung von Eingabewerten auf vordefinierte Animationen. Dadurch lassen sich bestimmte graphische Elemente in dieser Graphik in Abhängigkeit von konkreten Eingabewerten gezielt verändern und das Ergebnis visuell darstellen. Animationen können z.B. Rotationen, Verschiebungen, Textausgaben, Farbänderungen etc. sein. Die Zuordnung geschieht mittels einer *Mapping*-Datei und kann mit Hilfe eines Mapping-Editors oder beliebigen Texteditors erstellt werden, da es in einem für den Menschen leicht lesbarem Format (eXtensible Markup Language (XML)) gespeichert wird. Dies hat den Vorteil, das es einfach und schnell anpassbar ist und man nicht auf den in dieser Arbeit implementierten Mapping-Editor angewiesen ist.

## 1.1 Aufgabenstellung

Die drei wesentlichen Aufgabenschwerpunkte dieser Arbeit sind

1. die Vervollständigung der bereits begonnen Portierung der Vorgängerversion von KEV,
2. die Neuerstellung eines geeigneten Mappings zur Verknüpfung von Simulationsdaten und graphischer Darstellung und
3. das Ersetzen der vorhandenen Simulation der Modellbahn der Universität Kiel (siehe Kapitel 6) bzw. deren Anpassung an das neue KEV.

In Kapitel 2 werden die zugrunde liegenden Technologien von KEV kurz beschrieben und die für diese Arbeit wichtigen Punkte herausgestellt. Anschließend wird in Kapitel 3 die Bedeutung dieser Arbeit im Rahmen des KIELER-Projekts des Lehrstuhls für Eingebettete Systeme und Echtzeitsysteme dargestellt. Des Weiteren wird in Kapitel 3 beschrieben, wie die in Kapitel 2 eingeführten Technologien eingesetzt wurden um die Aufgabenstellung umzusetzen. In Kapitel 4 werden dann die bereits vorhanden Animationen detailliert beschrieben und deren Verhalten erklärt. Die Erstellung eines Beispielmappings wird dann anschließend in einzelnen Schritten anhand von Abbildungen in Kapitel 5 erklärt. Kapitel 6 gibt einen kleinen Einblick in die hier unter Punkt 3 beschriebene Fallstudie zur Modellbahnsimulation. Als weiteres werden in Kapitel 7 ein Ausblick auf mögliche Erweiterungen dieser Arbeit vorgestellt sowie mögliche Anwendungsszenarios von KEV beschrieben und in Kapitel 8 schlussendlich ein Fazit gezogen.

## 1.2 Verwandte Arbeiten

Ein zu KEV ähnliche Arbeit wurde an der Universität Osnabrück von Dorothee Langfeld im Rahmen einer Diplomarbeit [10] entwickelt. Dort geht es um die Visualisierung von Geoinformationen. Es wird jedoch nicht wie bei KEV die Eclipse Rich Client Platform, sondern eine eigenständige Web-Applikation zur Manipulation von SVG-Dateien verwendet. Ähnlich zu KEV wird ein Mapping verwendet, welches durch sogenannte „Konfigurationsdateien“ realisiert wird.

Im kommerziellen Bereich gibt es als konkretes Beispiel eine von der Firma ILOG (Tochterunternehmen der Firma IBM [7]) entwickelte Komponentenbibliothek für die ebenfalls kommerziellen Produkte *Adobe Flex* und *Adobe Air*. Diese Komponente heißt *IBM ILOG Elixir* [6] und dient ebenfalls der Datenvisualisierung mittels der zwei eben genannten Produkte.

# 2 Verwendete Technologien

Dieses Kapitel gibt einen Überblick über die im Rahmen dieser Studienarbeit verwendeten Technologien. Es ist für eine breite Leserschaft gedacht und dient somit als kleine Einführung zum besseren Verständnis der nachfolgenden Kapitel.

## 2.1 Interne Technologien

Bei den internen Technologien handelt es sich um Projekte welche am Lehrstuhl entwickelt wurden und im Zusammenhang mit dieser Studienarbeit verwendet wurden.

### 2.1.1 Kiel Integrated Environment for Layout for the Eclipse Rich client platform (KIELER)

Das KIELER Projekt stellt die Grundlage für diese und einer Reihe anderer Arbeiten am Lehrstuhl Echtzeitsysteme und Eingebettete Systeme des Instituts für Informatik der CAU zu Kiel dar. Es ist ein eigenständiges, auf der Eclipse Platform basierendes Programm, welches durch diverse Projekte in Form von sog. Eclipse-Plugins erweitert wird. KIELER selbst stellt dabei eine Erweiterung des Kiel Integrated Environment for Layout (KIEL) Projekts dar. Die Erweiterung bezieht sich hierbei auf die Portierung des KIEL Projekts auf die Eclipse Rich Client Platform. Die Portierung hat den großen Vorteil, KIELER in der Zukunft auf einfache Art und Weise erweitern und gleichzeitig die große Open-Source-Community von Eclipse nutzen zu können, um KIELER einer breiteren Masse an Benutzern zur Verfügung zu stellen.

### 2.1.2 KIELER Execution Manager (KIEM)

Wie unter Punkt 2.1.1 bereits erwähnt, besteht KIELER neben der eigentlichen RCP aus einer Reihe von Unterprojekten, zu welchen auch KIEM zählt. Der KIELER Execution Manager ist die Schnittstelle zwischen Simulation und KEV. Generell dient KIEM als Schnittstelle zwischen Erzeugern (Producer) und Beobachtern (Observer). Zum Austausch von Daten verwendet der Execution Manager das Datenaustauschformat JSON. KIEM verwaltet Erzeuger (Producer) von Daten z.B. Simulationen und Beobachter (Observer) z.B. KEV und ermöglicht die Ausführung in diskreten Zeitabschnitten (Ticks).

## 2.2 Externe Technologien

KEV verwendet auch eine ganze Reihe externer Technologien, welche allesamt nicht kommerziell sind. An dieser Stelle werden daher im Folgenden die fünf relevantesten näher erläutert.

### 2.2.1 Eclipse Rich Client Platform (RCP)

Als Eclipse Rich Client Platform wird die minimale Menge an Plugins bezeichnet, die benötigt wird um eine Rich Client Application (RCA) zu erstellen. Dieses Applikation weist die typische Eclipse Erscheinung sowie das Eclipse typische Verhalten auf. Es dient als Grundlage für eigenständige Applikationen und kann durch Plugins zu nahezu jeder beliebigen Anwendung erweitert werden.

### 2.2.2 Eclipse Plugins

Das von Eclipse verwendete Plugin-Konzept dient der einfachen Erweiterbarkeit bestehender Programme, bzw. der Erstellung neuer eigenständiger Programme, auf Grundlage der Eclipse RCP. Plugins können zur Laufzeit von Eclipse nachgeladen werden und verwendet werden. Es wird das Prinzip des sog. *lazy bindings* verwendet, was Eclipse dazu veranlasst die entsprechenden Plugins erst dann zu laden, wenn sie wirklich benutzt werden.

### 2.2.3 Scalable Vector Graphics (SVG)

Scalable Vector Graphics [14] ist eine vom W3C empfohlene Spezifikation zur Beschreibung zweidimensionaler Graphiken in Form von XML. Der Vorteil skalierbarer Vektorgraphiken gegenüber Rastergraphiken wie z.B. BMP, JPEG, PNG, etc. ist der, dass SVG-Dateien ohne Qualitätsverlust beliebig vergrößerbar sind. Das hängt damit zusammen, dass anstelle konkreter Farbinformationen für einzelne Pixel (wie es bei Rastergraphiken der Fall ist) nur die Information zur Erstellung geometrischer Objekte gespeichert werden.

Im Zusammenhang mit KEV hat die Beschreibung von SVG-Graphiken mittels XML den Vorteil, dass XML-Dateien auf einfache Art und Weise programmatisch veränderbar sind.

Obwohl die SVG-Spezifikation [14] Animationen vorsieht, konnten diese für KEV nicht verwendet werden, da die Animationen nicht über den *Execution Manager* gesteuert werden können, sondern mittels eines eigenständigen Batik-Threads ausgeführt werden. Ein weiteres Problem, welches

bei Implementierung der *MovePath-Animation* aufgetreten ist, ist die unterschiedliche Positionierung von SVG-Elementen. Ein SVG-Rechteck hat z.B. explizit die Positionskoordinaten der oberen linken Ecke relativ zum Ursprungskoordinatensystem der SVG-Graphik. Für ein SVG-Element, wie z.B. ein Stern oder ein Dreieck, welches durch einen Pfad beschrieben wird, gibt es hingegen diese Positionsangabe nicht. Dort wird der Ursprung implizit mit dem ersten Move-To-Befehl festgelegt. Verändert man jedoch nur diesen, so stimmen die restlichen Parameter, die den Pfad beschreiben, nicht mehr überein und man erhält ein deformiertes SVG-Element. Abhilfe schafft in diesem Fall nur eine Verschiebung des gesamten Elements mittels einer Translationsmatrix, welche vorher berechnet wurde. Eine Begründung, warum dieses inkonsistente Verhalten existiert und sowohl Rechteck, Kreis und Ellipse nicht auch einfach nur mittels eines Pfades beschrieben werden, wie sämtliche anderen komplexen Elemente auch, gibt es nicht. Die Vermutung liegt jedoch nahe, dass es daran liegt, dass Rechteck, Kreis und Ellipse spezielle geometrische Objekte sind, die häufig Verwendung finden und somit eine Sonderstellung einnehmen.

**Wichtige Hinweise zur Erstellung einer SVG-Graphik**

Bei der Erstellung einer SVG-Graphik ist darauf zu achten, dass fertig positionierte Elemente zum Schluss *gruppiert* werden und ihnen einen entsprechende *id* zugeteilt wird, damit sie später korrekt animiert werden können. Ist dies nicht der Fall kann es sein, dass ein Element bereits ein `transform`-Attribut enthält, welches später von der Animation überschrieben wird. Somit können unerwartete Animationen entstehen, welche die Fehlersuche unnötigerweise erschweren.

**Das SVG-Ursprungskoordinatensystem**

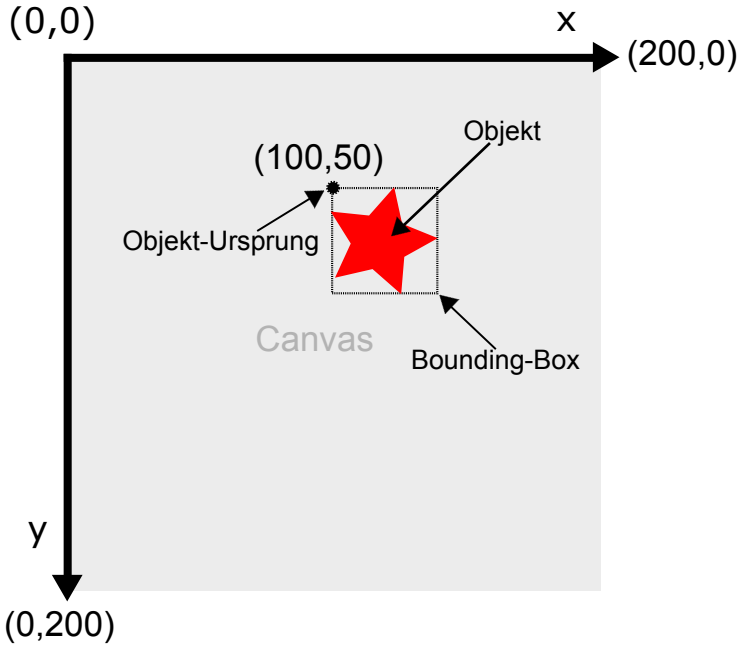



Abbildung 2.1: SVG-Ursprungskoordinatensystem

Das Koordinatensystem einer SVG-Graphik hat ihren Ursprung in der linken oberen Ecke. Die *Move-Animation* bezieht sich auf genau dieses Ursprungskoordinatensystem bei der Angabe der `xRange, yRange` Parameter. Die *Rotate-Animation* bzw. die *MovePath-Animation* hingegen beziehen sich bei der Angabe des `anchorPoint`-Attributs auf die obere linke Ecke ihrer Bounding-Box (siehe Abbildung 2.1).

### 2.2.4 Eclipse Modeling Framework (EMF)

Das Eclipse Modeling Framework (EMF) ist ein Unterprojekt von Eclipse zur Modellbasierten Softwareentwicklung. Mittels EMF wird JAVA-Quellcode aus sog. Modellen generiert. Diese Modelle können mittels des graphischen EMF-Editors erstellt werden oder aber aus einer XML Schema Definition (XSD)-Datei, aus UML-Diagrammen oder aus annotierten JAVA-

Schnittstellen generiert werden. Diese Modelle können dann in Form von XML-Dokumenten persistent gespeichert werden. Aus dem generierten JAVA-Quellcode können dann Instanzen von dem Modell gebildet werden und dieses anschließend programmatisch verändert werden. Darüber hinaus ist es möglich den generierten Quellcode um eigene Methoden zu ergänzen oder um generierte Methoden zu verändern. Nur das Modell zu erstellen und anschließend den Quellcode durch EMF erzeugen zu lassen kann in vielen Fällen eine Erleichterung sein, da sich der Programmieraufwand erheblich verringert. Ein weiterer Vorteil ist, dass der generierte Quellcode syntaktisch immer korrekt ist und somit eine weitere Fehlerquelle bei der Codeerzeugung eliminiert wird. Für weitere Informationen sei an dieser Stelle auf das EMF-Buch [13] von Dave Steinberg et al. verwiesen.

### 2.2.5 JavaScript Object Notation (JSON)

*JavaScript* Object Notation, kurz JSON [8], ist eine Untermenge der Programmiersprache *JavaScript* des sogenannten ECMA-262 Standards [4]. JSON dient als schlankes Austauschformat für Daten, welches sowohl für den Menschen als auch für Maschinen in leicht lesbarer Form vorliegt. Für viele Programmiersprachen gibt es mittlerweile eine JSON-Implementierung, weshalb die Wahl auf das JSON-Format nicht schwer viel. Darüber hinaus hat JSON bei der *JavaScript*-Programmierung den Vorteil, dass ein JSON-String mittels der `eval()`-Methode direkt in ein *JavaScript*-Objekt umgewandelt werden kann.

# 3 Umsetzung der Aufgabenstellung

Im Folgenden werden ein kurzer Überblick über die Stellung des KEV-Projekts im Rahmen anderer interner Projekte sowie Details zur Umsetzung des Mappings gegeben.

## 3.1 Zusammenhang zwischen KEV und den verwendeten Technologien

Das am Institut für eingebettete Systeme der Christian-Albrechts-Universität Kiel entwickelte Projekt mit dem Namen „Kiel Integrated Environment for Layout (KIEL)“ war genauso wie der Vorgänger von KEV eine eigenständige JAVA-Applikation. Die Entwicklungsumgebung Eclipse, welche im Rahmen des KIEL-Projekts verwendet wurde, bietet neben der Entwicklungsumgebung mit der Eclipse RCP auch das Grundgerüst für eingenständige Programme, die auf den Eclipse-Plugin-Mechanismus aufbauen. Da das alte KIEL nicht auf Eclipse basierte und diesen Plugin-Mechanismus zur einfachen Erweiterbarkeit daher nicht unterstützte, wurde KIEL zu KIELER erweitert und die Funktionalität von KIEL auf Grundlage der Eclipse RCP neu implementiert. In diesem Rahmen wurde auch das bereits vorhandene KEV-Projekt als Eclipse-Plugin neu implementiert. KEV wurde zur Visualisierung von Simulationsdaten entwickelt. Die Visualisierung geschieht hierbei durch Veränderung eines SVG-Dokumentes. Aus dem bereits vorhanden KEV-Plugin wurden für diese Arbeit einige Teile (insbesondere die Implementierung der graphischen Schnittstelle mit dem Batik SVG Framework) übernommen und daraus ein neues Plugin mit gleichem Namen erstellt. Dieses ist die Fortsetzung und Weiterentwicklung des bereits begonnenen KEV-Plugins. Der grundlegende Unterschied zur alten Version von KEV ist das sog. Mapping, was zur Verknüpfung von Simulationsdaten und Animationen von SVG-Elementen wird. Das neue Mapping wurde in Form eines EMF-Modells neu entwickelt und ist Grundlage dieser Studienarbeit.

Der wichtigste Punkt von KEV ist die Verknüpfung von SVG-Elementen mit vordefinierten Animationen. Diese Animationen werden zur Manipulation der jeweiligen SVG-Elemente verwendet, um diese auf Grund spezieller Eingabewerte zu verändern und diese Veränderung anschließend visuell darstellen zu können. Dies kann z.B. eine Animation zur Rotation eines SVG-Elementes oder eine Animation zur Veränderung der Farbe sein. Um eine leichte Erweiterbarkeit dieses Mappings zu gewährleisten wurde EMF verwendet. Auf diese Weise lassen sich schnell und bequem neue Animationen hinzufügen um das Mapping somit einfach erweiterbar zu machen.

Die Verwendung von KEV ist eng an ein anderes Unterprojekt von KIELER gebunden, dem KIELER Execution Manager (KIEM). Dieser stellt die Schnittstelle zwischen KEV und Simulation dar und verbindet diese indem er den Datenfluss von Simulation zum KEV-Plugin steuert. Der Austausch von Daten erfolgt hierbei in dem offenen JSON-Format[8], welches ein leichtgewichtiges Datenaustauschformat ist. JSON hat gegenüber XML den Vorteil, dass es weniger „aufgebläht“ ist. Dies ist insbesondere dann von Interesse, wenn das Mapping umfangreich ist und in jedem Schritt eine große Menge an Daten übermittelt wird. Für eine flüssige Animation ist es für KEV daher wichtig, die empfangenen Daten möglichst schnell und effizient auswerten zu können, wobei JSON hierfür gegenüber XML besser geeignet ist.

### 3.1.1 Neuerstellung des Mappings mittels EMF

EMF erzeugt mittels des EMF-Modells drei unterschiedliche JAVA-Packages.

1. Das Package *de.cau.cs.kieler.xkev.mapping* welches die Interface-Klassen der Animationen enthält.

2. Als zweites Package wird *de.cau.cs.kieler.xkev.mapping.impl* erstellt, in welchem sich die konkreten Implementierungen der einzelnen Animation befinden. Dieses ist das für den Programmierer relevante Package, da hier die konkreten Methoden für neue Animationen implementiert werden müssen.

3. Schließlich wird noch das *de.cau.cs.kieler.xkev.mapping.util* Package erzeugt, welches einige Hilfsklassen beinhaltet, die für die Animationen jedoch nicht von Belang sind.

Abbildung 3.1 zeigt die durch EMF generierten Packages und Klassen sowie das für die Verknüpfung von Animationen und SVG-Elementen relevante *de.cau.cs.kieler.mapping.animations* Package.

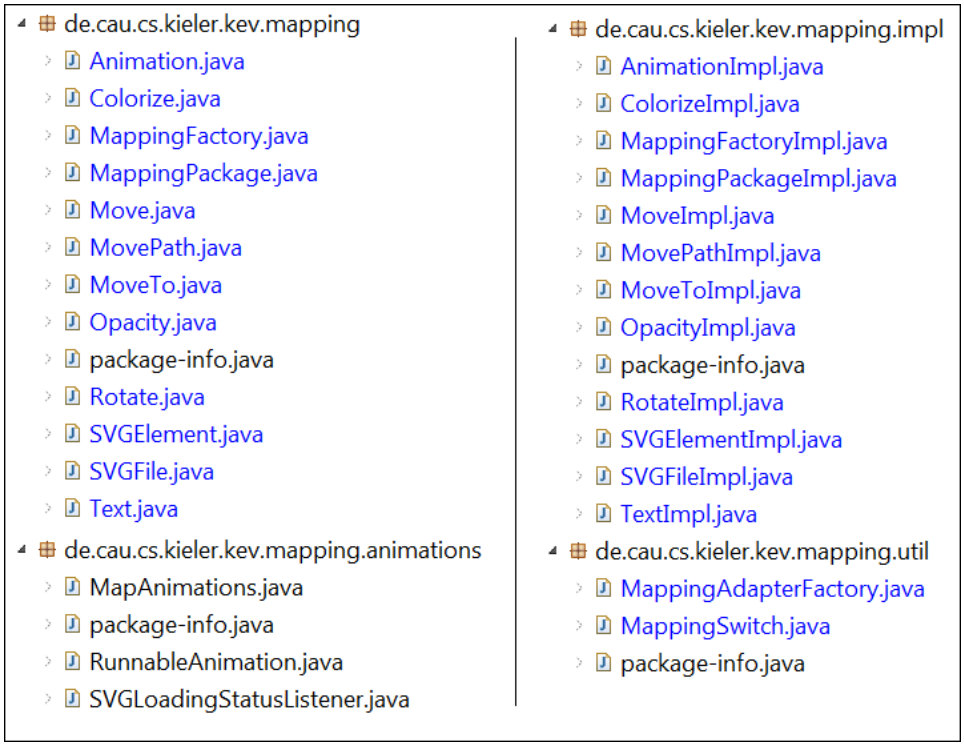



Abbildung 3.1: Packageübersicht des Mappings

### 3.1.2 Aufbau und Beschreibung des EMF-Modells für das Mapping

Jede Mapping-Datei verweist auf genau ein `SVGFile`, welches wiederum eine beliebige Anzahl von zu transformierenden `SVGElement`en enthalten kann. Jedes `SVGElement` kann nun eine beliebige Anzahl an `Animation`s enthalten. `Animation` ist als Interface definiert, welches es zu implementieren gilt. So ist eine Modularität gewährleistet, damit später auf einfache Art und Weise neue konkrete Animationen hinzugefügt werden können. Die Abbildung 3.2 zeigt das graphische EMF-Modell des Mappings.

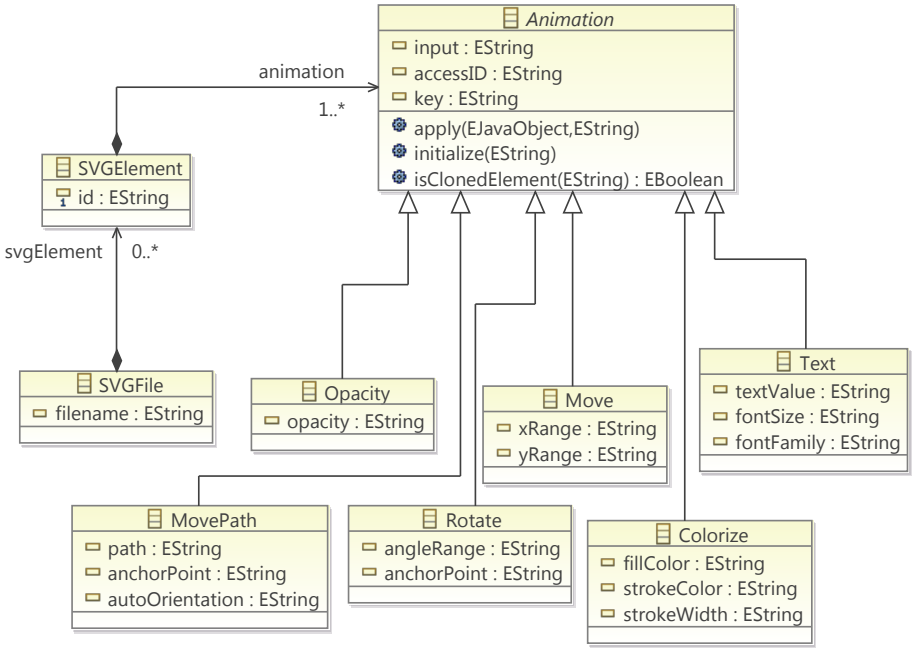



Abbildung 3.2: EMF-Modell des im KEV-Plugin verwendeten Mappings

Um eine neue Animation zu KEV hinzuzufügen, muss als erstes im Mapping-Modell eine neue Klasse mit dem Namen der Animation erstellt werden. Diese leitet sich direkt von der Klasse `Animation` ab. Im nächsten Schritt werden dann die Attribute vom Typ `EString` für die neue Animation festgelegt. Anschließend muss dann das EMF-Modell nur noch mittels Generator-Modell neu generiert werden. Der Programmierer hat dann die Aufgabe die zwei Methoden `apply() und initialize()` zu implementieren. Die dritte Methode (`isClonedElement()`) ist für alle Animationen gleich und wurde somit nur einmalig in der `AnimationImpl`-Klasse implementiert.

Wichtig bei der Implementierung der Methoden `apply()` und `initialize()` ist, dass der EMF-Tag (@generated) entweder gelöscht oder auf (@generated NOT) gesetzt wird, damit bei der Codegenerierung die eigene Implementierung nicht überschrieben wird.

# 4 Das Mapping und die einzelnen Animationen

## 4.1 Das Mapping

Das Mapping selbst wird mit Hilfe einer Mapping-Datei durch den von EMF generierten *Mapping-Editor* erstellt. Es ist jedoch auch möglich einen XML-Editor oder einen einfachen Texteditor zu verwenden, da die Mapping-Datei im XML-Format gespeichert wird. Die Erstellung im XML-Editor ist gerade bei größeren Dateien sehr aufwendig, da die Syntax genau eingehalten werden muss und sich schnell Fehler ergeben können. Dies wird durch die Verwendung des Mapping-Editors verhindert. Der Vorteil bei der Verwendung eines XML-Editors liegt jedoch bei der späteren Editierung einzelner Attribute, was mit dem XML-Editor um einiges schneller geht als mit dem *Mapping-Editor*. Gegenüber einem einfachen Texteditor bietet der Eclipse-XML-Editor den Vorteil des *syntax highlightings*, wobei Elemente, Attribute, Kommentare etc. farbig hervorgehoben werden. Beide Editoren, sowohl der Mapping-Editor als auch der XML-Editor, haben ihre Vor- und Nachteile und es liegt letztlich an den Vorlieben des Benutzers selbst, welchen er für seine Arbeit bevorzugt. Die Nutzung beider Möglichkeiten erleichtert jedoch deutlich die effiziente Erstellung von Mapping-Dateien. Der Mapping-Editor eignet sich besonders für die initiale Erstellung einer Mapping-Datei, während sich der XML-Editor zur schnellen Anpassung der Datei im weiteren Verlauf besser eignet. Zum direkten Vergleich ist in Abbildung 4.2 eine Mapping-Datei jeweils im *Mapping-Editor* und im Eclipse-XML-Editor dargestellt.

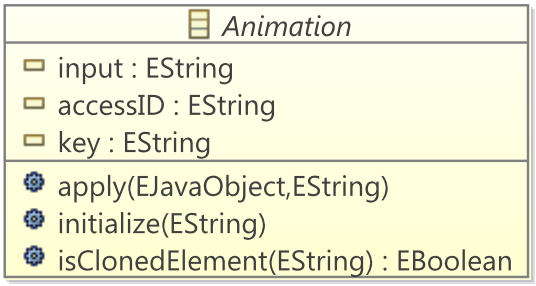



Abbildung 4.1: Klassendiagramm des Animation Interfaces

Alle Animationen haben die gemeinsamen Attribute `input`, `key` und `accessID`. Des Weiteren muss jede Animation die Methoden `apply()` und `initialize` implementieren (vgl. Abbildung 4.1).

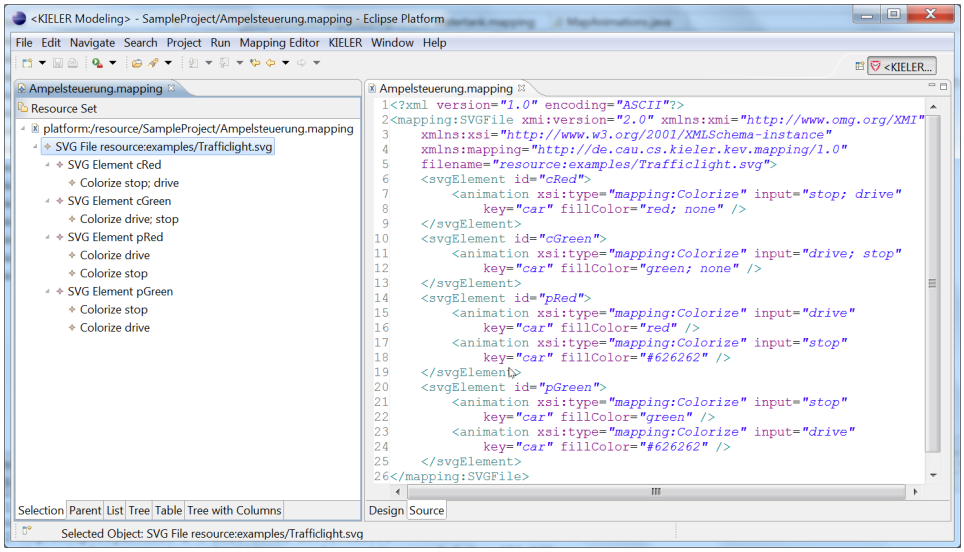



Abbildung 4.2: Darstellung einer Mapping-Datei sowohl im generierten *Mapping-Editor* als auch im Eclipse-XML-Editor

## 4.2 Gültige Werte und ihre Schreibweise

Eingabewerte können entweder Einzelwerte oder eine kommaseparierte Liste von Werten `(wert1,wert2,wert3)` sein. Die kommaseparierte Liste von Werten ist als Kurzschreibweise gedacht, um jeweils nur eine Animation eines jeden Typs für verschiedene Eingabewerte definieren zu müssen (vgl. Listing 4.3). Dies spart außerdem Ressourcen, da somit nur eine Instanz eines Typs einer Animation angelegt werden muss.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2 <mapping:SVGFile xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:mapping="
      http://de.cau.cs.kieler.xkev.mapping/1.0"
3 filename="resource:examples/Ampel.svg">
4   <!-- Listenschreibweise -->
5   <svgElement id="cRed">
6     <animation xsi:type="mapping:Colorize" input="on,off" fillColor="red
          ,#696969"/>
7   </svgElement>
8   <svgElement id="cGreen">
9     <animation xsi:type="mapping:Colorize" input="on,off" fillColor="green
          ,#696969"/>
10  </svgElement>
11  <!-- Einzelelementschreibweise -->
12  <svgElement id="pRed">
13    <animation xsi:type="mapping:Colorize" input="on" fillColor="red" />
14    <animation xsi:type="mapping:Colorize" input="off" fillColor="#696969"
          />
15  </svgElement>
16  <svgElement id="pGreen">
17    <animation xsi:type="mapping:Colorize" input="on" fillColor="green"/>
18    <animation xsi:type="mapping:Colorize" input="off" fillColor="#696969"
          />
19  </svgElement>
20 </mapping:SVGFile>
```

Quellcode 4.3: Unterschied zwischen Listen- und Einzelwertschreibweise

### Der leere `input`

Ein besonderer Fall tritt ein, wenn bei einer Animation das `input`-Attribut weggelassen wird oder es dem leeren String (`""`) entspricht. Dann interpretiert KEV diese Animation als Defaultanimation, die immer dann ausgeführt wird, wenn keine andere Animation gleichen Typs mit einem gültigen Wert für `input` existiert. Die leere Eingabe entspricht also dem „sonst“-Fall.

**Beispiel:** Angenommen es existieren zwei Animation $A, D$. Die Animation $A$ wird genau dann ausgeführt, wenn der aktuelle JSON-Wert $j$ einem der gültigen Eingabewerte (`"on, an, true"`) für das `input`-Attribut entspricht. Ist $j$ nicht in diesem Bereich, so wird die Defaultanimation $D$ ausgeführt.

$$j \in \texttt{input} \Rightarrow \text{wende Animation } A \text{ an}$$
$$j \notin \texttt{input} \Rightarrow \text{wende Animation } D \text{ an} \qquad (\text{„sonst“-Fall})$$

**Die Steuer-/Sonderzeichen**

Bei der Erstellung einer Mapping-Datei gibt es drei Sonderzeichen zu beachten:

| Sonderzeichen | Erklärung |
|---|---|
| $*NAME* | Der Dollar-Operator legt fest, dass es sich bei dem Wert *NAME* um ein JSON-Key handelt, dessen Wert anstelle $*NAME* für das Mapping verwendet werden soll. |
| _ | Der Unterstrich darf nicht das erste Zeichen eines Wertes des `key`-Attributs sein, da dieses Zeichen intern verwendet wird, um geklonte Elemente zu adressieren. |
| . | ID's von SVG-Elementen dürfen, ebenso wie JSON-Keys keinen Punkt im Namen enthalten. Der Punkt wird intern von KEV dazu verwendet, verschachtelte JSON-Objekte zu adressieren. Beispiel: Sei folgendes JSON-Objekt `{ "signal": { "present" : true, "value" : "A" }}` gegeben. Dann lässt sich der Wert des Signals durch die Adressierung `signal.value` auslesen, das Ergebnis wäre `"A"`. |
| $x..y$ | $x, y \in \mathbb{N}_0$ – Zwei aufeinander folgende Punkte zwischen zwei natürlichen Zahlen $x$ und $y$ definieren einen Wertebereich. Beispiel: `input="1..10"` bedeutet, dass alle Werte $x >= 1$ und $y <= 10$ gültige Eingabewerte sind. |

## 4.3 Beschreibung der einzelnen Animationen

Dieser Teil befasst sich mit den einzelnen Animationen, die in KEV implementiert wurden. Diese werden im Folgenden jeweils einzeln näher beschrieben und ihre Besonderheiten bei der Implementierung herausgestellt. Insbesondere sollen hier die jeweiligen animationsspezifischen Attribute erläutert werden. Die möglichen Eingabewerte für die Mapping-Attribute entsprechen den durch die SVG-Spezifikation[14] bzw. der CSS2-Spezifikation[15] festgelegten Angaben. Im Folgenden wird daher nicht

mehr auf die jeweiligen Eingabewerte der speziellen Attribute eingegangen, es sei denn, sie weichen von den oben genannten Spezifikationen ab.

**Allgemeine Attribute:**

| Attributname | Erklärung |
|---|---|
| `input` | Enthält gültige Eingabewerte, welche für die Animation auf das jeweilige SVG-Element angewendet werden soll. |
| `key` | Legt einen von der SVG-Element-ID unterschiedlichen JSON-Key zur Überprüfung der `input`-Werte fest. |
| `accessID` | Sofern es sich bei dem JSON-Key um ein Array handelt, legt dieses Attribut fest, von welcher Position im Array der Eingabewert gelesen werden soll. |

### 4.3.1 Die Colorize-Animation

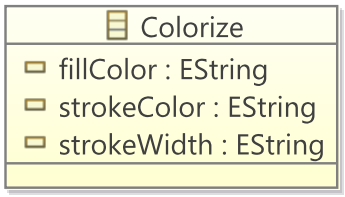
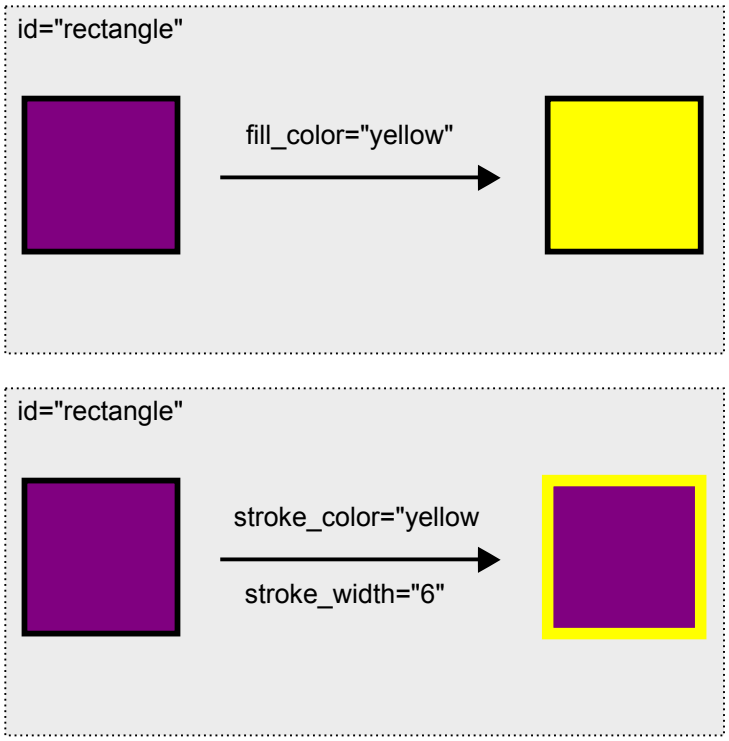



Abbildung 4.4: Klassendiagramm der *Colorize-Animation*

Die *Colorize-Animation* dient sowohl zum Färben von Flächen als auch zur Veränderung der Rahmenfarbe und -dicke eines SVG-Elements. Gültige Werte für das Attribut `color` sind sämtliche nach der SVG-Spezifikation[14] erlaubten Farbwerte. Im Folgenden sei hier nur auf die Wichtigsten verwiesen.

Abbildung 4.5 zeigt das Beispiel einer *Colorize-Animation* mit dem Auszug aus einer entsprechenden Mapping-Datei. Das `fillColor`-Attribut wird angewendet, sobald der JSONKey *FillColor* den Wert *red* oder *blue* annimmt. Für das `strokeColor`- und `strokeWidth`-Attribut ändert sich die Rahmenfarbe und -dicke genau dann, wenn der JSON-Key *StrokeColor* den Wert *on* bzw. *off* annimmt.




```
1  <svgElement id="rectangle">
2    <animation xsi:type="mapping:Colorize" input="violet, yellow"
         fillColor="#800080, yellow" key="FillColor" />
3    <animation xsi:type="mapping:Colorize" input="on,off" key="StrokeColor
         " strokeColor="yellow, black" strokeWidth="6, 3" />
4  </svgElement>
```

Abbildung 4.5: Die *Colorize-Animation* mit dem relevanten Teil aus einer entsprechenden Mapping-Datei

**Spezifische Attribute:**

| Attributname | Erklärung |
|---|---|
| `fillColor` | Setzt die zu verändernde Füllfarbe eines SVG-Elementes. (Siehe „Erlaubte Werte für Farben.“) |
| `strokeColor` | Setzt die zu verändernde Rahmenfarbe eines SVG-Elementes. (Siehe „Erlaubte Werte für Farben.“) |
| `strokeWidth` | Bestimmt die Rahmendicke eines SVG-Elementes in Pixeln. |

**Erlaubte Werte für Farben:**

none: Wenn *none* als Parameter verwendet wird, so ist die Fläche transparent gefüllt und die ursprüngliche Farbe ist sichtbar.

$\#xxyyzz$: Wobei $x, y, z \in [0..9, A..F]$ die Farbwerte für die Farben Rot, Grün und Blau (RGB) darstellen.

*Farbname*: Es können auch sämtliche in der SVG-Spezifikation definierten Farbnamen verwendet werden z.B. *lightgoldenrodyellow = #FAFAD2*.

**Besonderheiten bei der Implementierung**

Die *Colorize-Animation* verändert das SVG-`style`-Attribut, welches neben Füllfarbe (`fill`) und der Rahmenfarbe (`stroke`) noch elementspezifische Parameter enthalten kann. Für weitere Informationen sei an dieser Stelle auf die SVG-Spezifikation[14] verwiesen. Stimmt nun ein `input`-Wert mit dem JSON-Wert, welcher unter der `id` bzw. dem `key` gespeichert ist, überein, so wird der aktuelle Farbwert des `fill`-Attributes mit dem unter `fillColor` angegebenen Farbwert ersetzt.

### 4.3.2 Die Move-Animation

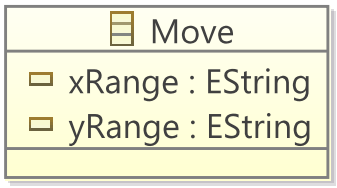
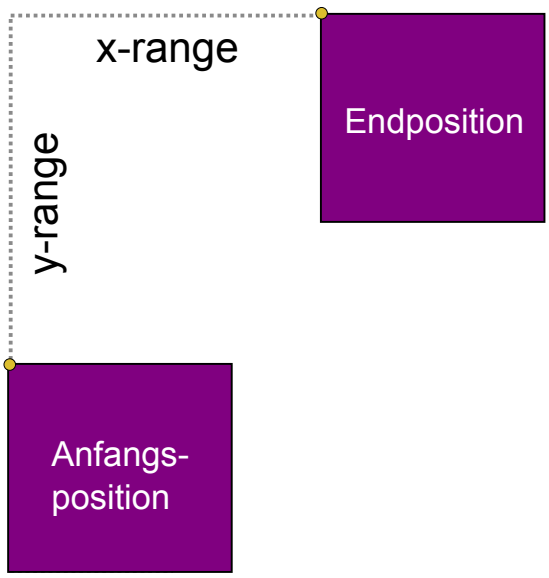



Abbildung 4.6: Klassendiagramm der *Move-Animation*

Die *Move-Animation* verschiebt ein SVG-Element in x- bzw. y-Richtung in Bezug auf das SVG-Koordinatensystem. Eingabewerte werden hierbei auf den in den Attributen `xRange` bzw. `yRange` angegebenen Wertebereich übertragen (interpoliert).




```
1  <svgElement id="rectangle">
2    <animation xsi:type="mapping:Move" input="1..50" xRange="20..70" />
3    <animation xsi:type="mapping:Move" input="51..100" yRange="60..30" />
4  </svgElement>
```

Abbildung 4.7: Die *Move-Animation* mit dem relevanten Teil aus einer entsprechenden Mapping-Datei

**Spezifische Attribute:**

| Attributname | Erklärung |
|---|---|
| `xRange` | Der Inputwert/Inputwertebereich wird auf den unter `xRange` angegebenen Wertebereich verteilt und beschreibt somit eine Verschiebung des SVG-Elementes auf der x-Achse. Mögliche gültige Werte sind Einzelwerte, kommaseparierte Werte (z.B. 12.90, 34, 36) und Wertebereiche (z.B. 100..10). Wobei ausschließlich Zahlenwerte zulässig sind. |
| `yRange` | Der `yRange` verhält sich analog zum `xRange`, beschreibt jedoch anstelle der x-Achse eine Verschiebung des SVG-Elementes auf der y-Achse. |

#### Hinweis

Die Werte für die beiden Wertebereiche sind als absolute Werte zu verstehen. Sie beziehen sich auf das Ursprungskoordinatensystem der SVG-Graphik (vgl. Abbildung 2.1).

#### Besonderheiten bei der Implementierung

Bei der *Move-Animation* wird das SVG-Element mittels einer Translation (Verschiebung) auf den durch die Mapping-Attribute `xRange/yRange` festgelegten Wert/Wertebereich in $x$- bzw. $y$-Richtung verschoben. Dies wird durch das SVG-*transform*-Attribut und den *translate*$(x, y)$-Befehl realisiert.

### 4.3.3 Die Text-Animation

Die *Text-Animation* dient zum Anzeigen von generiertem Text in einem bestimmten SVG-Element. Darüber hinaus enthält die Animation Attribute zum Verändern der Textgröße (`fontSize`) sowie der Schriftart (`fontFamily`).

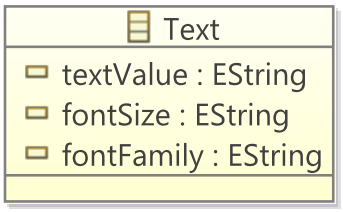



Abbildung 4.8: Klassendiagramm der *Text-Animation*

Das Beispiel in Abbildung 4.9 zeigt, wie sich ein Textfeld bei bestimmten Eingabewerten eines Zählers (`key="counter"`) verändert. Der aktuelle Wert aus dem JSON-Key *counter* wird auf den Wertebereich von 0 bis 100 gemappt, wobei in dem Beispiel vier Wertebereiche unterschieden werden, in denen sich der Text verändert. Sofern der Wert von *counter* auf einen Inputbereich mappt, wird der aktuelle Wert des JSON-Keys *currentValue* als Text ausgegeben. Wird kein `input`-Attribut angegeben, so wird die jeweilige Animation bei jedem Schritt ausgeführt und der Text wäre jeweils der aktuelle JSON-Wert von *currentValue*.

**Spezifische Attribute:**

| Attributname | Erklärung |
|---|---|
| `textValue` | Gibt den anzuzeigenden Text an. |
| `fontSize` | Legt die Schriftgröße fest. |
| `fontFamily` | Bestimmt die zu verwendende Schriftart. |

**Hinweis**

Gerade im Fall der *Text-Animation* kann es sinnvoll sein, den $-Operator zu verwenden (vgl. 4.2) . Es kann z.B. vorkommen, dass aktuelle Statusinformationen einer Simulation angezeigt werden sollen. Ohne den $-Operator ist es jedoch nur möglich vordefinierte statische Texte auszugeben. Lässt man allerdings das `input`-Attribut weg und verweist mit dem $-Operator im `textValue`-Attribut auf den Wert eines JSON-Keys, so wird in jedem Schritt der jeweils aktuelle Wert dieses JSON-Keys ausgegeben. Beispiel:
`<animation xsi:type="mapping:Text" textValue="$statusValue"/>`

**Besonderheiten bei der Implementierung**

Jedes *text*-Element einer SVG-Graphik enthält ein Kindelement, dass den darzustellenden Text beinhaltet. Die *Text-Animation* überschreibt genau dieses *text*-Element und verändert das *style*-Attribut des SVG-Elementes, sofern ein Wert für das `fontSize`- bzw. `fontFamily`-Attribut vorhanden ist, indem es die Attribute *font-style* bzw. *font-family* mit den Werten aus der Mapping-Datei überschreibt.


```
1   <svgElement id="centerText">
2     <animation xsi:type="mapping:Text" input="100..70" key="counter"
          textValue="$currentValue" font_size="24" fontFamily="Arial" />
3     <animation xsi:type="mapping:Colorize" input="100..70" key="counter"
          textValue="$currentValue" fillColor="green" strokeColor="white" />
4     <animation xsi:type="mapping:Text" input="69..40" key="counter"
          textValue="$currentValue" font_size="24" fontFamily="Arial" />
5     <animation xsi:type="mapping:Colorize" input="69..40" key="counter"
          textValue="$currentValue" fillColor="white" strokeColor="white" />
6     <animation xsi:type="mapping:Text" input="39..20" key="counter"
          textValue="$currentValue" font_size="14" fontFamily="Arial" />
7     <animation xsi:type="mapping:Colorize" input="39..20" key="counter"
          textValue="$currentValue" fillColor="white" strokeColor="white" />
8     <animation xsi:type="mapping:Text" input="19..0" key="counter"
          textValue="$currentValue" font_size="24" fontFamily="Times New
          Roman" />
9     <animation xsi:type="mapping:Colorize" input="19..0" key="counter"
          textValue="$currentValue" fillColor="red" strokeColor="white" />
10  </svgElement>
```

Abbildung 4.9: Die *Text-Animation* mit dem relevanten Teil aus einer entsprechenden Mapping-Datei

### 4.3.4 Die MovePath-Animation

Bei der *MovePath-Animation* handelt es sich um eine Kombination von *Move-Animation* und *Rotate-Animation*. Die *MovePath-Animation* bewegt ein SVG-Element anhand eines vordefinierten SVG-Pfades. Zusätzlich zur Bewegung entlang des Pfades gibt es die Möglichkeit das SVG-Element mittels des `autoOrientation`-Attributes am Pfad auszurichten. Die Schwierigkeit bei dieser Animation lag in der Berechnung der einzelnen Pfadpunkte und der automatischen Ausrichtung um den gegebenen `anchorPoint`.

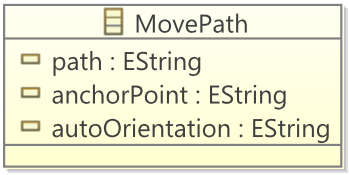
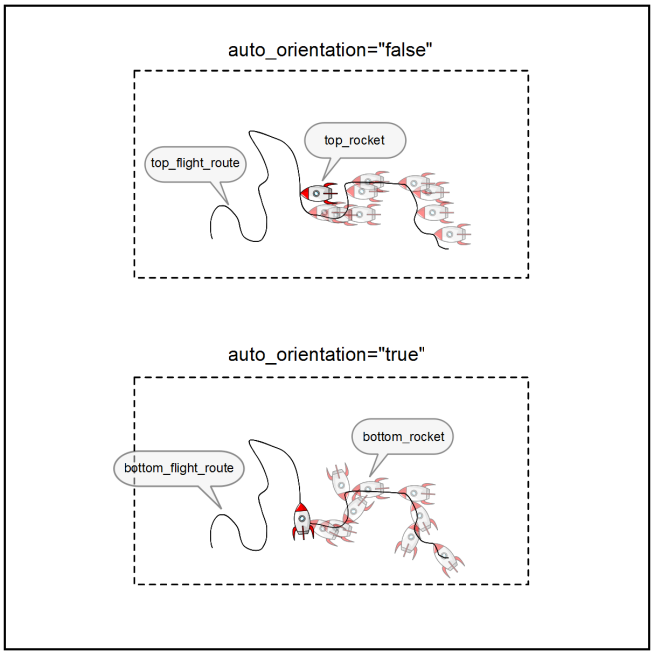



Abbildung 4.10: Klassendiagramm der *MovePath-Animation*

**Spezifische Attribute:**

| Attributname | Erklärung |
|---|---|
| `path` | ID des zu verwendenden SVG-path-Elementes. |
| `anchorPoint` | Gibt den Ankerpunkt $(x, y)$ relativ zur linken oberen Ecke der Bounding-Box des zu animierenden SVG-Elementes an. Eingabewerte für $x$ bzw. $y$ können sowohl Ganzzahl als auch Gleitkommazahlen (durch . dargestellt) sein. Beispiel: `anchorPoint="0.67,12.3983"`. |
| `autoOrientation` | Kann als mögliche Parameter *nur* `true` oder `false` enthalten. |




```
1  <svgElement id="top_rocket">
2    <animation xsi:type="mapping:MovePath" input="1..200" path="
         top_flight_route" anchor_point="0,5" autoOrientation="false" />
3  </svgElement>
4  <svgElement id="bottom_rocket">
5    <animation xsi:type="mapping:MovePath" input="1..200" path="
         bottom_flight_route" anchor_point="0,5" autoOrientation="true" />
6  </svgElement>
```

Abbildung 4.11: Die *MovePath-Animation* mit dem relevanten Teil aus einer entsprechenden Mapping-Datei[1]

[1] Die Rakete stammt von `http://www.openclipart.org/`

**Besonderheiten bei der Implementierung**

Bei der *MovePath-Animation* wird das SVG-path-Element, welches mittels des `path`-Attributes der Mapping-Datei definiert ist, in genau die Anzahl an Punkten unterteilt, die der Anzahl an verschiedenen Inputwerten entspricht. Dieses Punkte werden dann in einer Liste gespeichert und anschließend die Winkeldifferenz zweier aufeinanderfolgender Punkte berechnet. Diese Winkeldifferenz wird anschließend dazu verwendet, die Ausrichtung des SVG-Elementes dem Pfad anzupassen, sofern das `autoOrientation`-Attribut den Wert `true` hat. Des Weiteren wird in jedem *Tick* der Ausführung das SVG-Element mit dem mittels des `anchorPoint`-Attribut festgelegten Ankerpunkt auf den entsprechenden Pfadpunkt verschoben. Dadurch bewegt sich das Element auf dem Pfad mit jedem Tick von einem Pfadpunkt zum Nächsten und wird ggf. dem Pfad entsprechend ausgerichtet.

### 4.3.5 Die Opacity-Animation

Die *Opacity-Animation* verändert die Deckkraft eines beliebigen SVG-Elementes. Der Wertebereich erstreckt sich von 0 = unsichtbar bis 1 = vollständig sichtbar. Diese Animation kann somit dafür verwendet werden, um Elemente zu verstecken oder sichtbar zu machen oder aber einen bestimmten Grad an Transparenz zu erzielen.

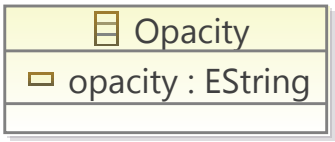
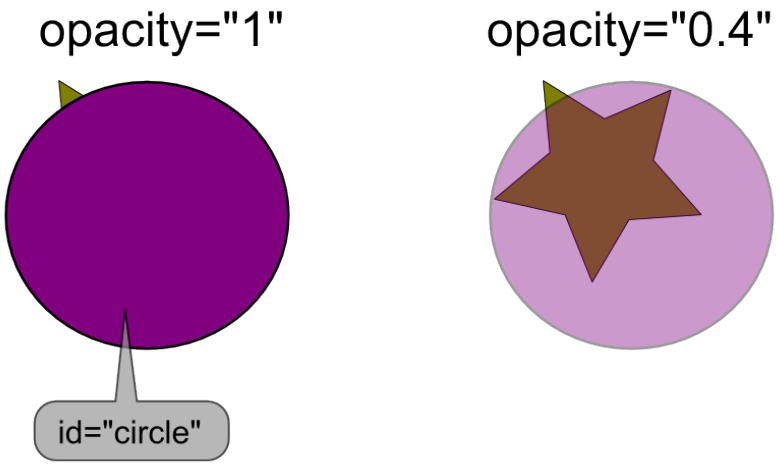



Abbildung 4.12: Klassendiagramm der *Opacity-Animation*

**Spezifische Attribute:**

| Attributname | Erklärung |
|---|---|
| `opacity` | Gibt den zu setzenden Wert für die Deckkraft des zu animierenden SVG-Elementes an. Gültige Werte sind alle Gleitkommazahlen zwischen 0 und 1. |




```
1  <svgElement id="circle">
2    <animation xsi:type="mapping:Opacity" input="off,on" opacity="1,0.4"
         />
3  </svgElement>
```

Abbildung 4.13: Die *Opacity-Animation* mit dem relevanten Teil aus einer entsprechenden Mapping-Datei

#### Besonderheiten bei der Implementierung

Die *Opacity-Animation* verändert das SVG-style-Attribut eines beliebigen Elementes und überschreibt dort den (sofern vorhandenen) *opacity*-Wert mit dem aktuellen Wert aus der Mapping-Datei.

### 4.3.6 Die Rotate-Animation

Um ein SVG-Element um einen beliebigen Punkt mit einem beliebigen Winkel zu drehen, kann die *Rotate-Animation* verwendet werden. Sie rotiert ein Element um einen durch das `anchor_point`-Attribut festgelegten Punkt. Der Winkel wird dabei im Gradmaß angegeben und kann sowohl negative (Rotation gegen den Uhrzeigersinn) als auch positive (Rotation im Uhrzeigersinn) Zahlenwerte enthalten.

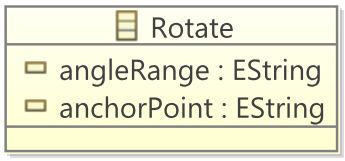
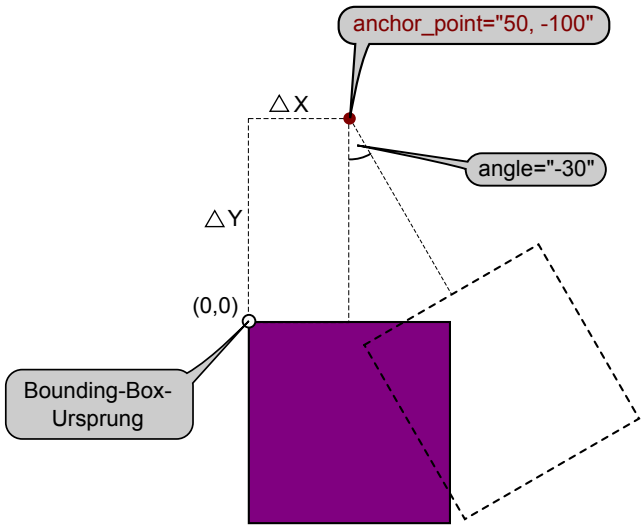



Abbildung 4.14: Klassendiagramm der *Rotate-Animation*




```
1  <svgElement id="rectangle">
2    <animation xsi:type="mapping:Rotate" input="1..30" anchorPoint="50,
         -100" angle="-45..0" />
3  </svgElement>
```

Abbildung 4.15: Die *Rotate-Animation* mit dem relevanten Teil aus einer entsprechenden Mapping-Datei

**Spezifische Attribute:**

| Attributname | Erklärung |
|---|---|
| `angleRange` | Gibt den Winkelbereich an, um den sich das SVG-Element drehen soll. Nur Zahlenwerte sind erlaubt. Negative Zahlenwerte beschreiben eine Rotation entgegen des Uhrzeigersinns; positive Werte eine Rotation im Uhrzeigersinn. |
| `anchorPoint` | Legt den Ankerpunkt fest, um den sich das SVG-Element drehen soll. Der Ursprung $(0, 0)$ ist die obere linke Ecke der Bounding-Box des zu drehenden Elementes. Ohne Angabe des Ankerpunktes wird explizit der Wert $(0, 0)$ festgelegt. |

**Hinweis**

Die *Rotate-Animation* verwendet bei der Positionierung des Ankerpunktes relative Werte. Der Ursprung stimmt auch wie bei der *MovePath-Animation* nicht im mit dem SVG-Ursprungskoordinatensystem überein, sondern liegt in der oberen linken Ecke der Bounding-Box des SVG-Elementes.

**Besonderheiten bei der Implementierung**

Bei der *Rotate-Animation* wird, wie auch schon bei der *Move-Animation* und *MovePath-Animation*, das SVG-*transform*-Attribut verändert, um das gewünschte Ergebnis zu erreichen. Hierzu wird das *transform*-Attribut mit dem SVG-Rotationsbefehl (*rotate($\theta, x, y$)*) überschrieben. Die Werte für $x$ und $y$ werden, sofern angegeben, direkt aus dem `anchorPoint`-Attribut übernommen. $\theta$ ist hierbei der Winkel in Grad.

# 5 Der Mapping-Editor

Dieses Kapitel erklärt die Funktionsweise und die Verwendung des durch EMF generierten Mapping-Editors. Am konkreten Beispiel einer Ampelsteuerung wird nun die Erstellung der entsprechenden Mapping-Datei

1. anhand des mitgelieferten Editors, welcher von EMF aus den Werten des Metamodells generiert wurde und

2. mittels des Eclipse internen XML-Editors

erläutert. Der Vorteil bei EMF ist, dass die durch den generierten Editor gespeicherten Daten im XML-Format vorliegen und somit neben dem Mapping-Editor auch auf einfache Art und Weise per XML- oder Texteditor angepasst werden können. In den folgenden Beispielgraphiken (Abbildungen 5.3 bis 5.8) ist die Mapping-Datei sowohl im Mapping-Editor als auch in dem von Eclipse mitgelieferten XML-Editor dargestellt.

## 5.1 Erstellen einer Mapping-Datei

Um eine neue Mapping-Datei anzulegen, wird der *New-Wizard* von Eclipse aufgerufen und eine neue Datei vom Typ *Mapping Model* erzeugt (siehe Abbildung 5.1).

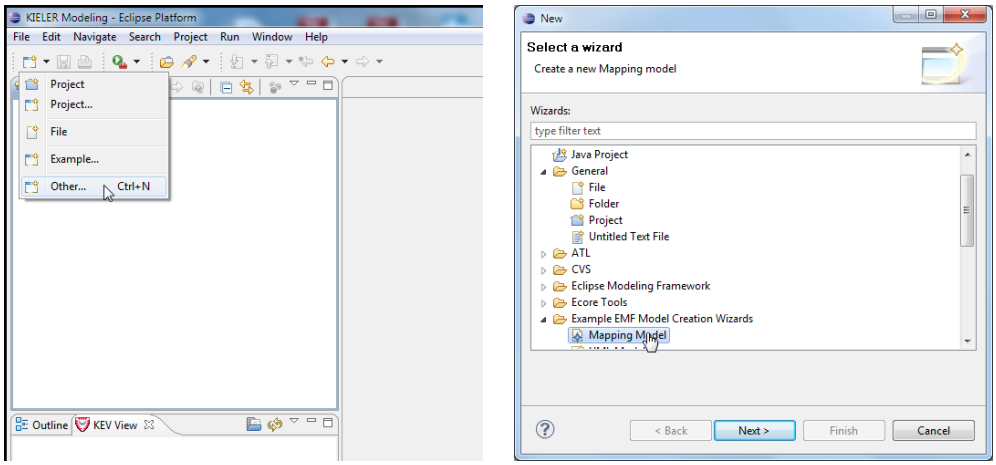



Abbildung 5.1: Anlegen einer neuen Mapping-Datei – Teil 1

Als nächstes muss ein Name für die Mapping-Datei angegeben und ein existierendes Projekt zum Speichern der neuen Datei ausgewählt werden. Im

folgenden Dialog muss für das *Model Object*-Feld das *SVG-File*-Objekt ausgewählt werden. Abbildung 5.2 veranschaulicht dieses Vorgehen.

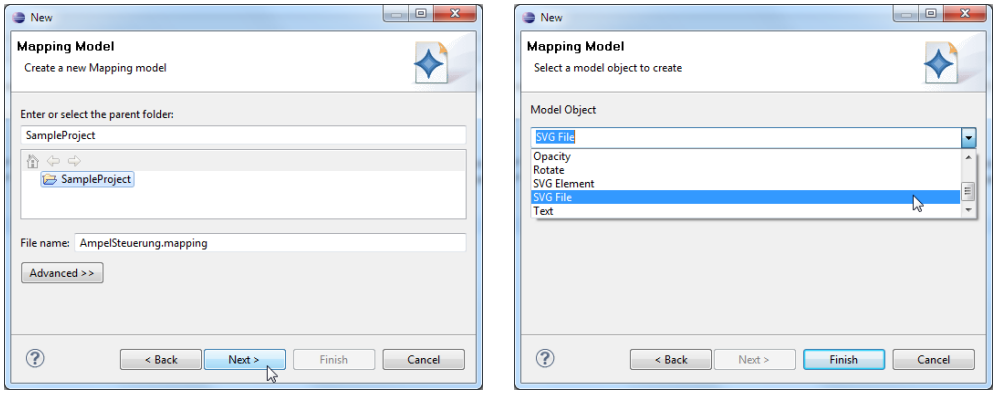



Abbildung 5.2: Anlegen einer neuen Mapping-Datei – Teil 2

Nach dem Erstellen der Mapping-Datei – hier im Beispiel die Datei *Ampelsteuerung.mapping* – öffnet ein Doppelklick auf diese den Mapping-Editor. Die Abbildung 5.3 zeigt die Mapping-Datei sowohl im Mapping-Editor als auch im XML-Editor.

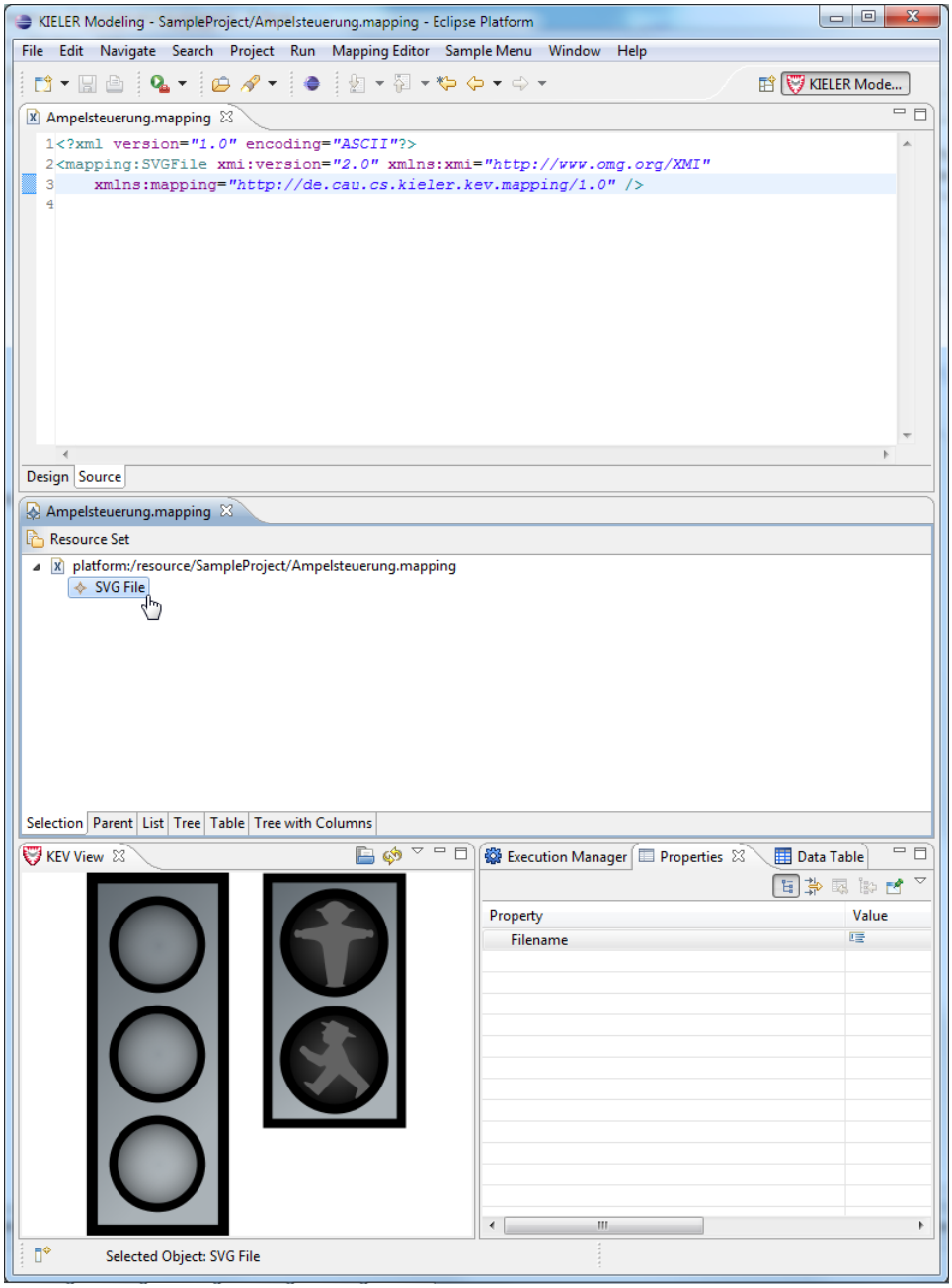



Abbildung 5.3: Die Mapping-Datei im Eclipse-XML-Editor (Oben) und im Mapping-Editor (Mitte) direkt nach der Erstellung

Im Mapping-Editor sieht man, dass das *SVG-File* das Wurzelelement der Mapping-Datei darstellt. Jede Mapping-Datei kann auf genau ein solches *SVG-File*-Objekt verweisen. Ein Rechtsklick auf diesem Objekt zeigt das Kontextmenü von Eclipse, indem sich ein neuer Eintrag (*New Child*) befindet.

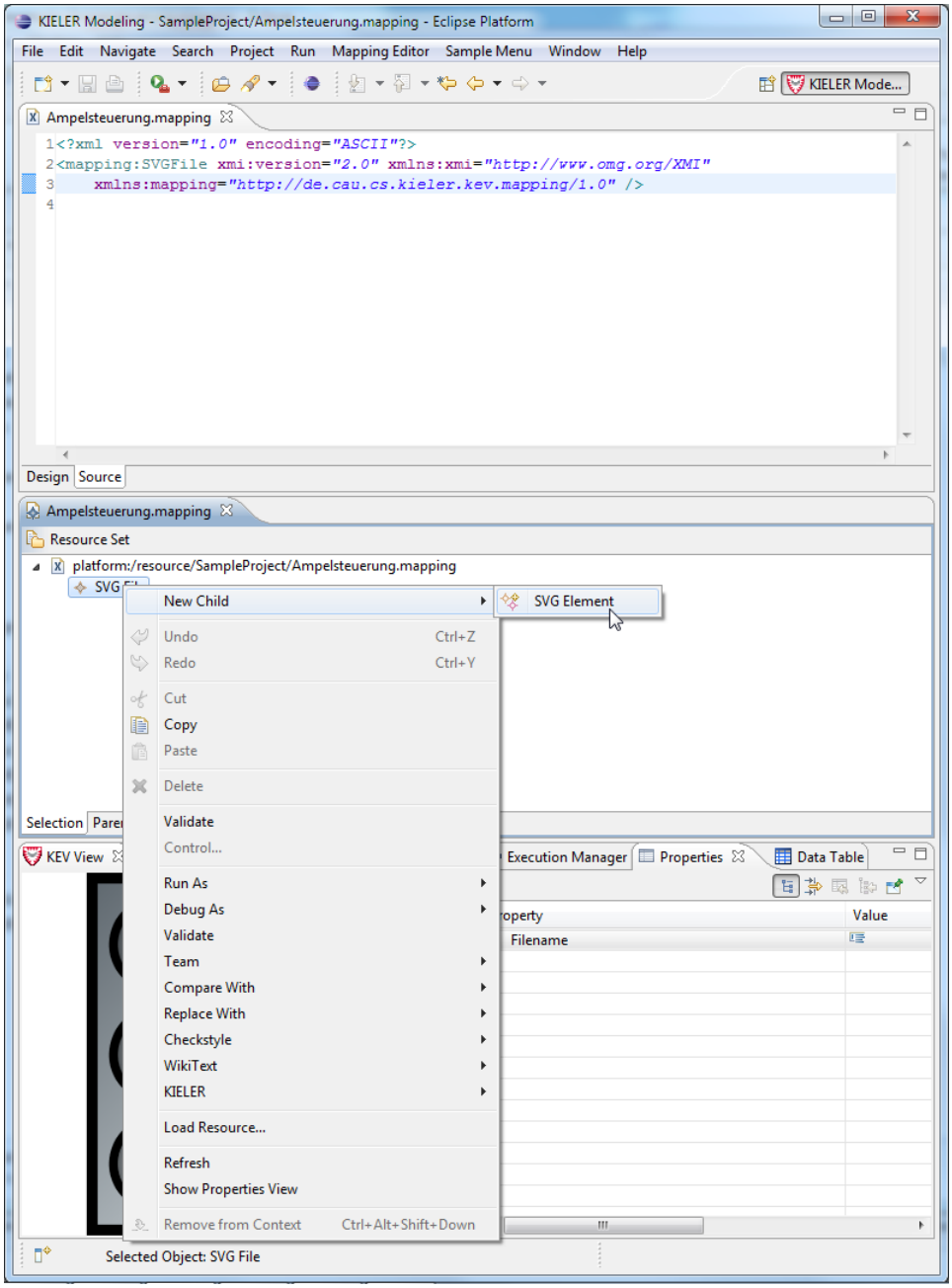



Abbildung 5.4: Hinzufügen eines SVG-Elementes, welches später animiert werden soll.

Wählt man diesen aus, wie in Abbildung 5.4 dargestellt, so fügt man einen Unterknoten vom Typ *SVG-Element* zum Mapping hinzu. Diesem neuen Unterknoten muss man dann noch mittels der Eclipse Property-View eine ID zuweisen. Diese ID muss mit der ID des Objekts in der SVG-Datei übereinstimmen.

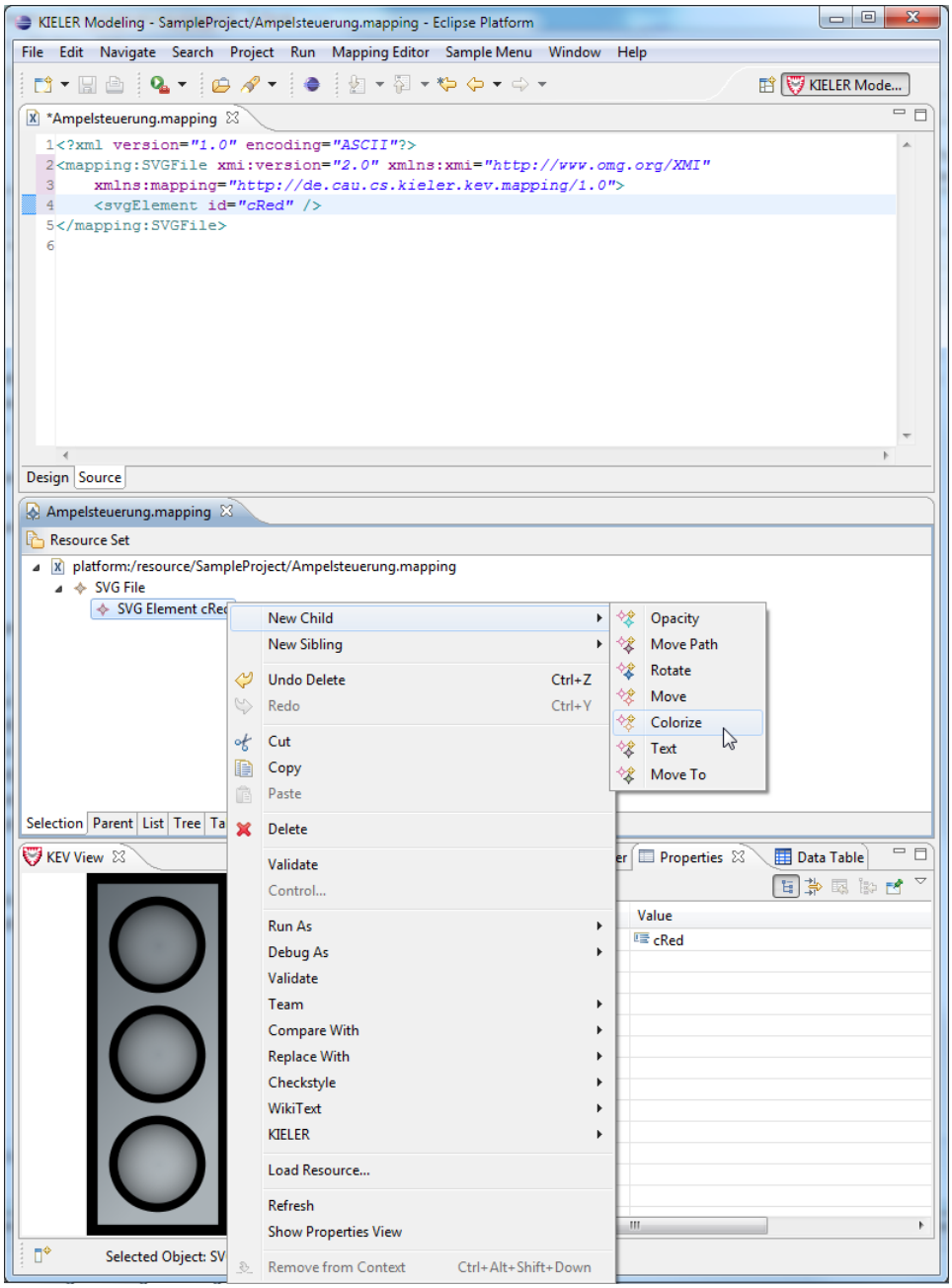



Abbildung 5.5: Verknüpfung eines SVG-Elementes mit einer Animation

Ein weiterer Rechtsklick auf den Unterknoten ermöglicht diesmal unter dem Punkt *New Child* dem *SVG-Element* eine Animation hinzuzufügen (siehe Abbildung 5.5).

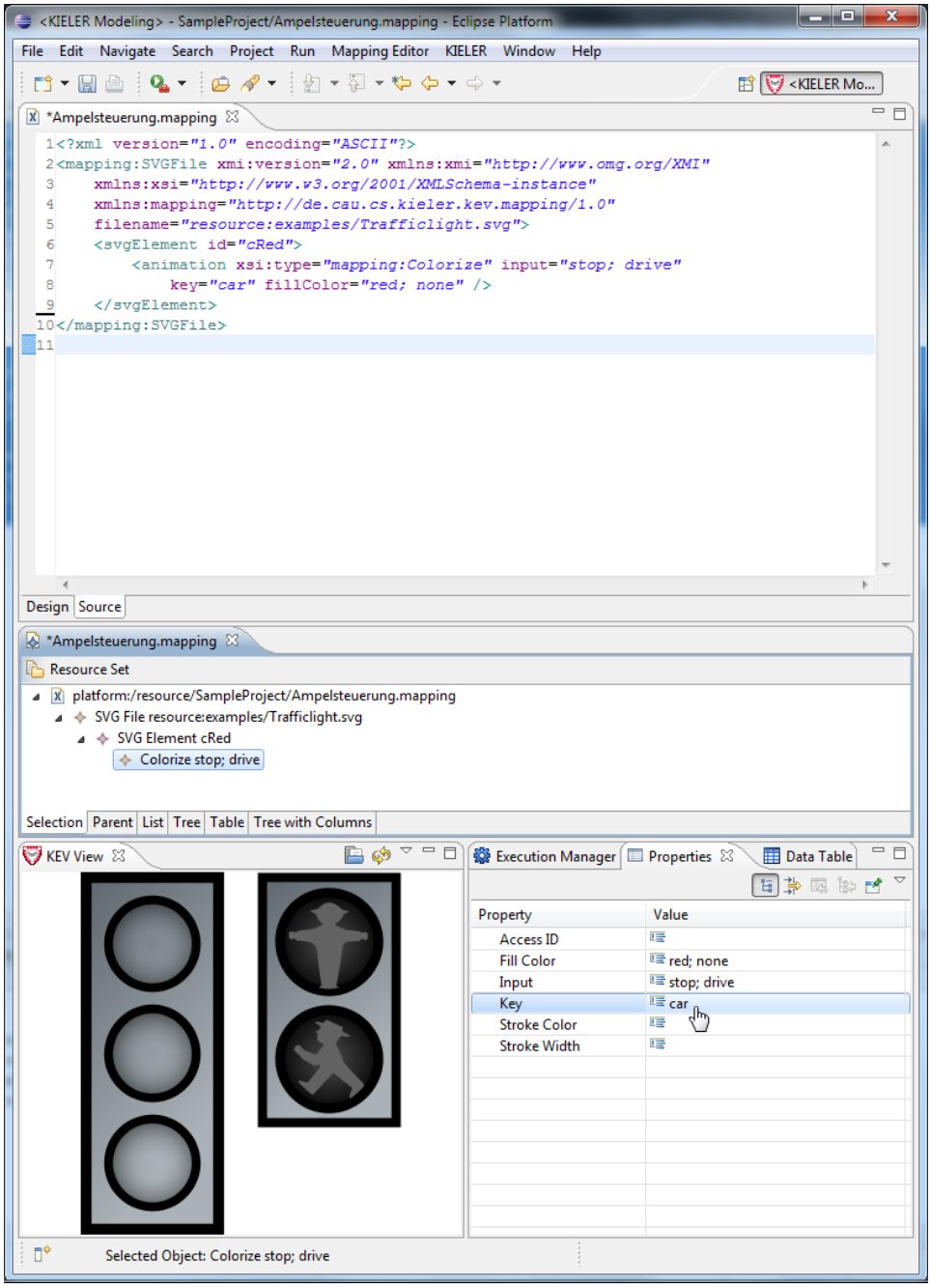



Abbildung 5.6: Darstellung eines SVG-Elements im Editor mit den entsprechenden Attributwerten in der Eclipse-Property-View

Mittels der Eclipse-Property-View können die jeweiligen Attribute für alle Elemente verändert werden. In Abbildung 5.6 ist dies für das Beispiel der Ampelsteuerung dargestellt.

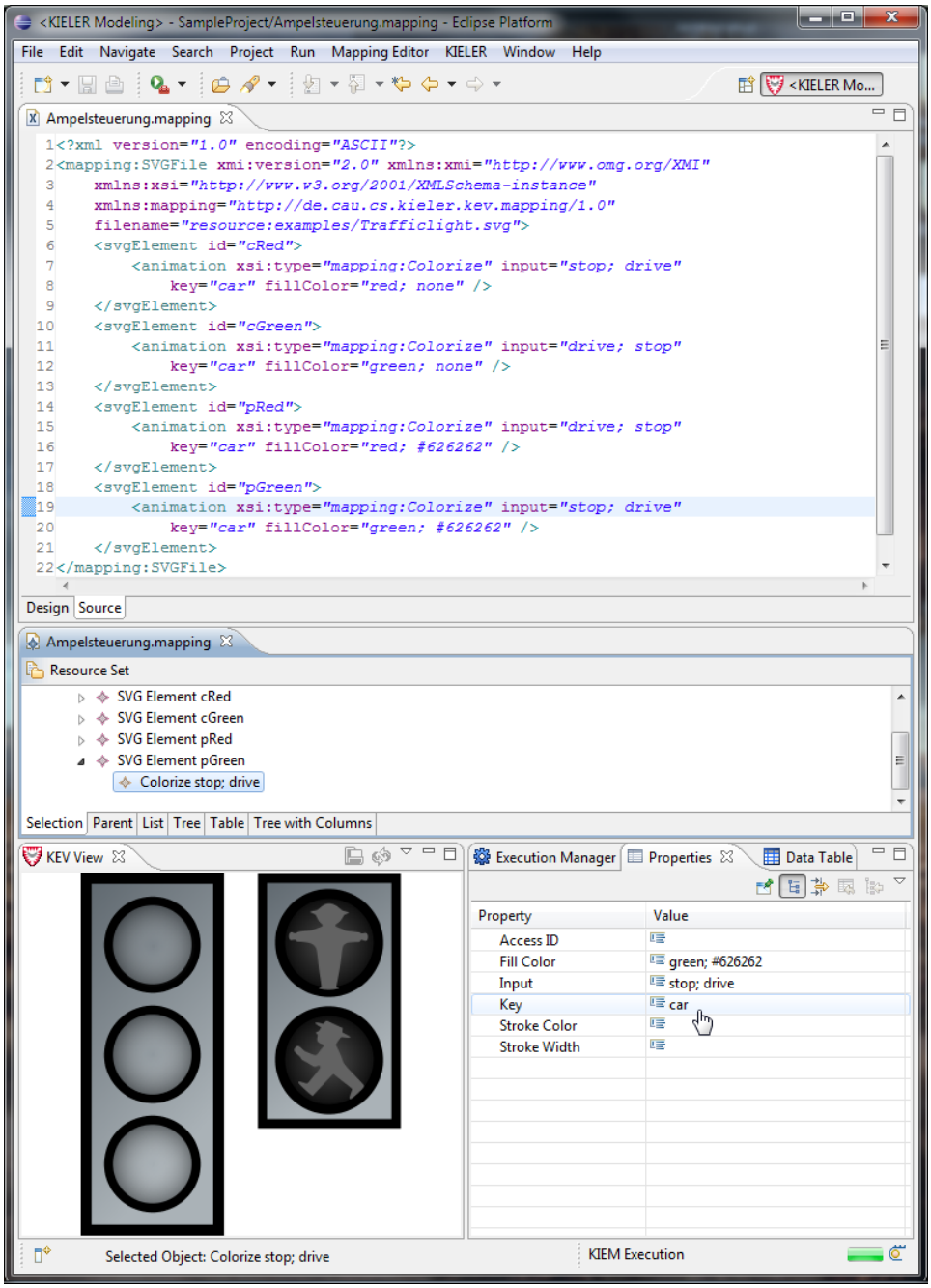



Abbildung 5.7: Das fertige Mapping für die Ampelsteuerung sowohl im Mapping-Editor als auch im XML-Editor.

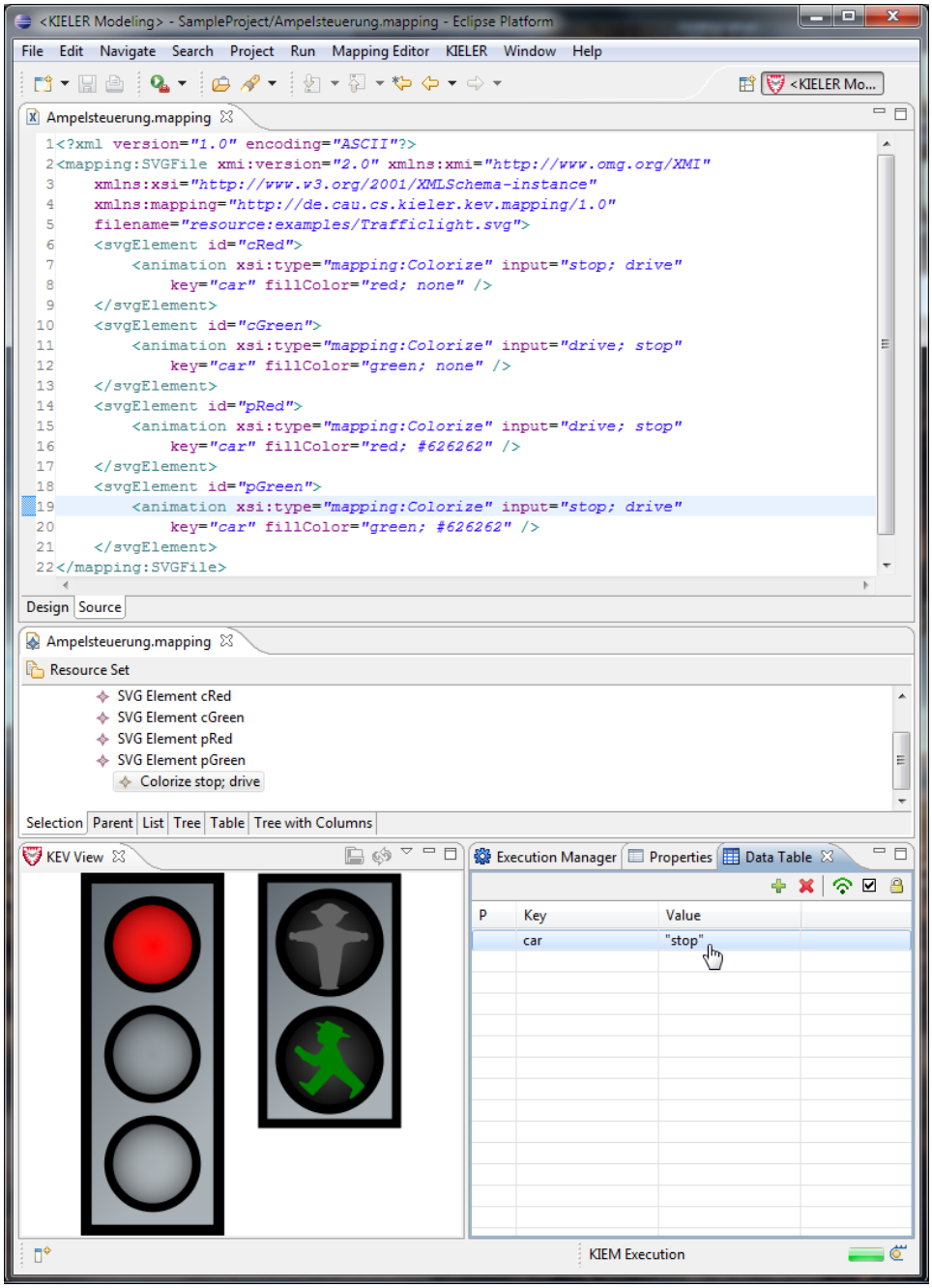



Abbildung 5.8: Die Ampeldarstellung nachdem der JSON-Key `car` mit Hilfe der Eclipse-Data-Table-View auf „stop“ gesetzt wurde.

Im Beispiel dieses Mappings werden jeweils die SVG-Elemente *pRed* und *cGreen* sowie *pGreen* und *cRed* in der SVG-Datei verändert. Hierbei stehen *pRed, pGreen* für das rote und grüne Licht der Fußgängerampel und *cRed, cGreen* für die entsprechenden Lichter der Auto-Ampel (siehe Ampel in der KEV-View).

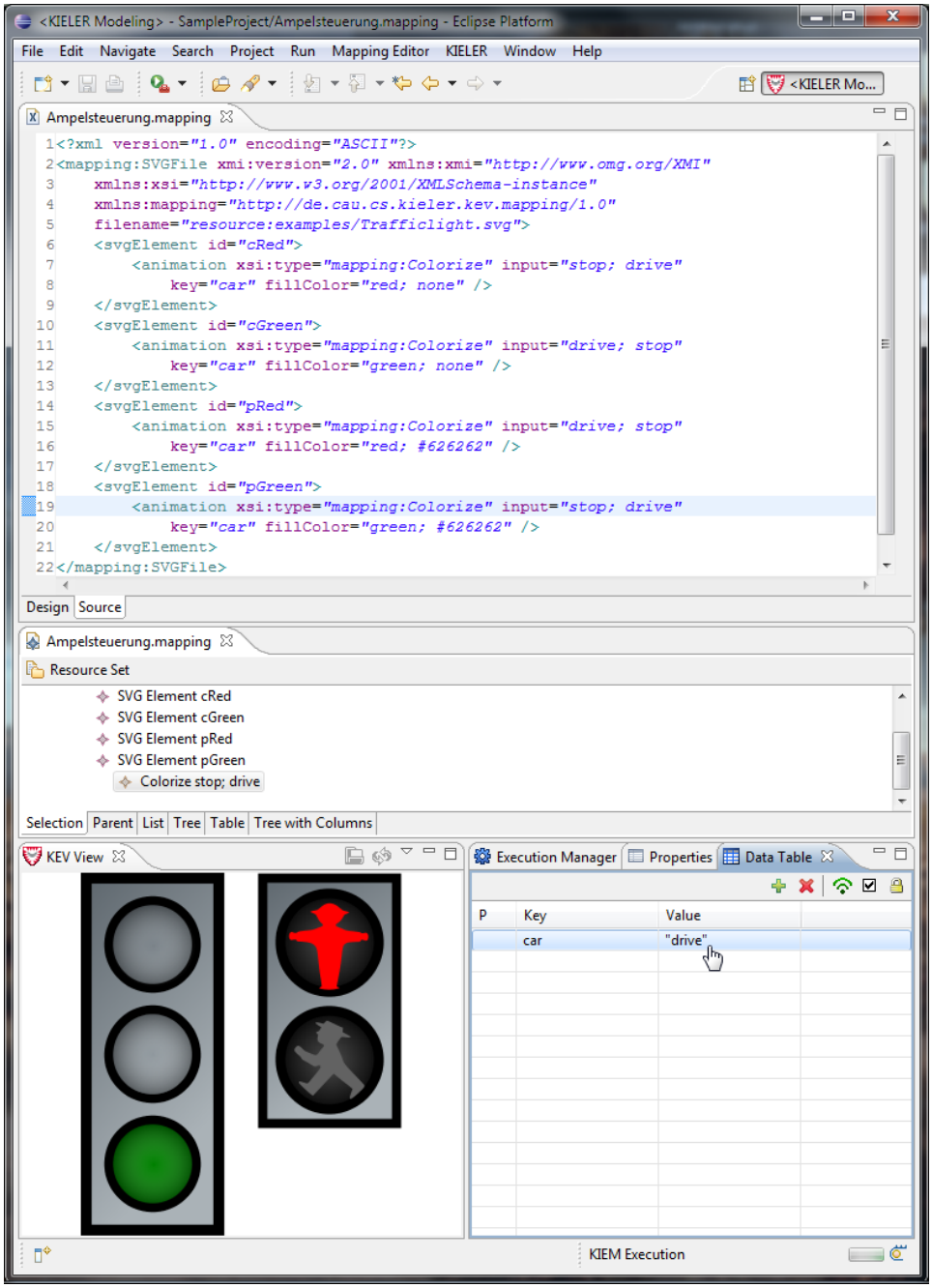



Abbildung 5.9: Die Ampeldarstellung nachdem der JSON-Key `car` mit Hilfe der Eclipse-Data-Table-View auf „drive“ gesetzt wurde.

Die beiden Abbildungen 5.8 und 5.9 zeigen die Animation der Ampel für den Wert von `car="stop"` (5.8) bzw. `car="drive"` (5.9). Das Verändern des JSON-Wertes für das `key`-Attribut *car* bewirkt den paarweisen Wechsel der Ampelfarben *cRed, pGreen* bzw. *cGreen, pRed*.

# 6 Fallstudie zur Modellbahnsimulation

Eine weitere Studienarbeit mit dem Titel „Modellbasierte Umgebungssimulation für verteilte Echtzeitsysteme mit flexiblem Schnittstellenkonzept - Fallstudie einer Bahn-Anlage“[12] am Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme des Instituts für Informatik der CAU zu Kiel hatte sich bereits mit dem Vorgänger des KEV-Plugins auseinandergesetzt. Damals hieß die Kieler Visualisierungsumgebung noch *ModelGUI* und war eine eigenständige und vom damaligen KIEL-Projekt unabhängige Applikation. Die damalige Studienarbeit nutzte den KEV-Vorgänger zur Simulation der Modellbahnanlage der Universität Kiel. Die Simulation der Anlage auch mit KEV darstellen zu können war eine der Hauptaufgaben der vorliegenden Arbeit. Das neue mittels EMF realisierte Mapping wurde daher auch im Hinblick auf die Durchführbarkeit der Modellbahnsimulation entwickelt. Abbildung 6.1 zeigt die Eisenbahnanlage selbst sowie das entsprechende Modell in Form einer SVG-Graphik mit den entsprechenden Zuständen der Gleise, Weichen und Züge. Die aktuellen Positionsdaten der Züge sowie die Zustände der Weichen und Schranken kommen direkt von dem Eisenbahnmodell. Für die genauen Abläufe sowie der Ansteuerung der Modellbahn sei an dieser Stelle auf die oben erwähnte Studienarbeit zu diesem Thema verwiesen.

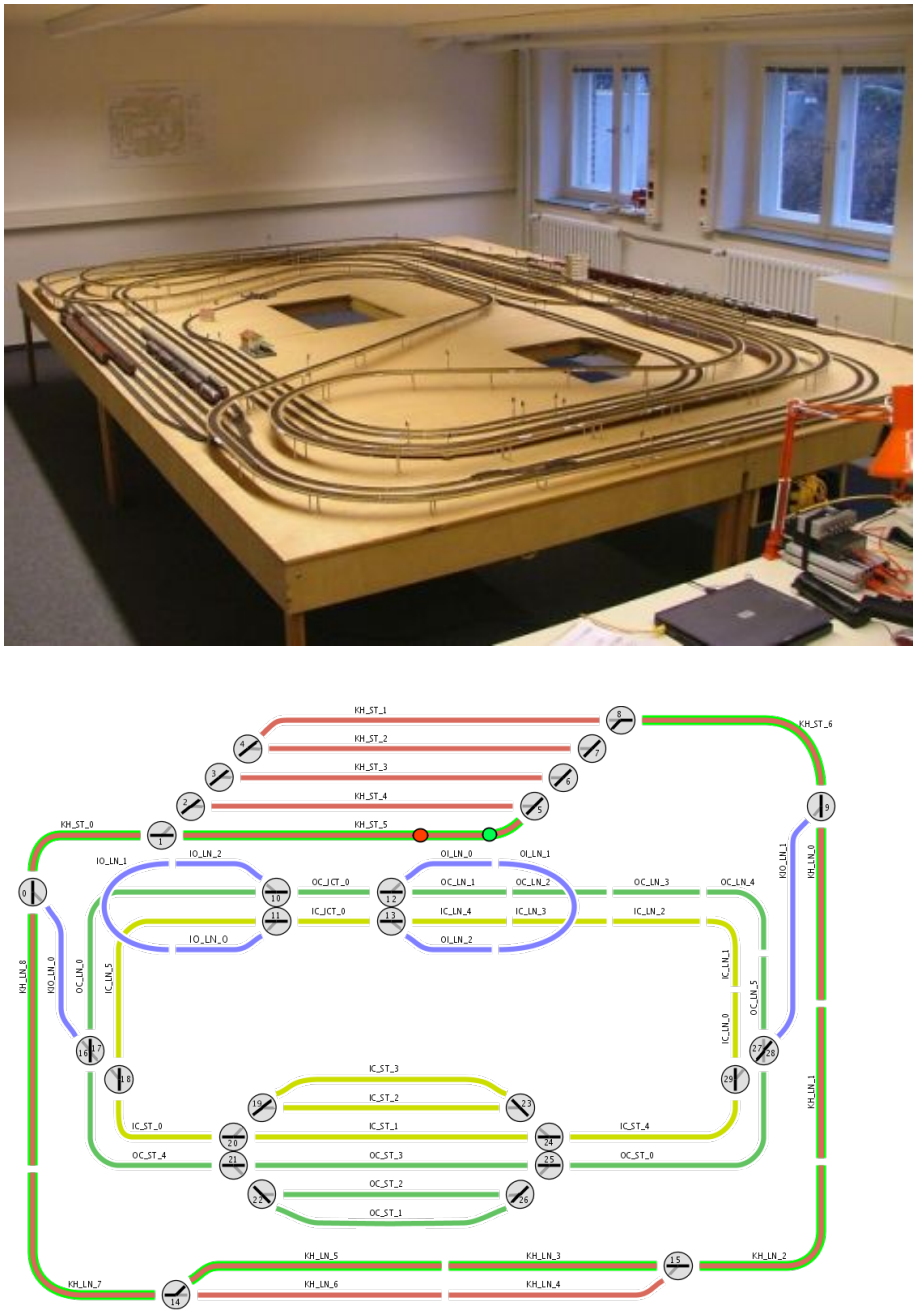



Abbildung 6.1: Modellbahnanlage der Universität Kiel[11] und das Modell als SVG-Graphik

# 7 Ausblick auf mögliche Erweiterungen von KEV

Im folgenden Abschnitt werden einige Anwendungsszenarios vorgestellt sowie ein kurzer Ausblick auf die Erweiterbarkeit von KEV gegeben. Außerdem sollen an dieser Stelle mögliche Erweiterungen/Verbesserungen des durch EMF generierten Mapping-Editors diskutiert werden.

## 7.1 Mögliche Szenarios

Die Verwendung von KEV ist praktisch für sämtliche technischen Geräte denkbar, die ihre aktuellen Zustandsdaten mittels einer Schnittstelle (welche vorzugsweise JSON als Austauschformat verwendet) ihrer Umwelt mitteilen können. Dadurch wäre dann eine graphische Simulation beziehungsweise eine aktuelle Statusansicht des jeweiligen Gerätes in Form einer – dem Gerät entsprechenden – SVG-Graphik möglich. So kann man sich z.B. einen Fahrstuhl-Monteur vorstellen, der sich gerade am Notebook mittels einer geeigneten graphischen Darstellung der realen Fahrstuhlumgebung den aktuellen Systemzustand anzeigen lässt und dadurch den Fehler schnell, effizient und kostengünstig finden und beheben kann.

## 7.2 Implementierung eines neuen Mapping-Editors

Da der mitgelieferte bzw. von EMF – aus dem Metamodell – generierte Mapping-Editor doch recht rudimentäre Funktionen bietet, wäre eine mögliche Verbesserung die Neuimplementierung des Mapping-Editors. Dieser könnte z.B. eine Kombination aus Texteditor und der für das Mapping vorgesehenen SVG-Graphik sein. Denkbar wäre dann eine Animation zu einem SVG-Element hinzuzufügen, indem dieses direkt in der Graphik ausgewählt wird. Dadurch könnte die Benennung aussagekräftiger IDs entfallen, weil ja nun eine direkte Verbindung zwischen SVG-Element und Animation mittels der SVG-Graphik besteht. Des Weiteren könnte ein SVG-Element mit Hilfe von *Drag & Drop* verschoben werden, um z.B. Start- und Endpunkt eines Wertebereichs zu definieren. Dies würde die Erstellung eines Mappings erheblich beschleunigen und vereinfachen. Auch wären kleine Symbole vorstellbar, welche die jeweiligen Animationen repräsentieren und, sofern sie mit einem SVG-Element verknüpft sind, direkt an diesem angezeigt werden können. Damit hätte der Benutzer direkt eine Übersicht darüber, welches Element bereits mit welcher Art von Animation verknüpft ist.

Eine weitere sinnvolle Erweiterung von KEV wäre ein SVG-Editor, mit dem man eine neue SVG-Graphik aus bereits vorgefertigten Komponenten

(auch wieder SVG-Graphiken) z.B. mittels *Drag & Drop* auf einfache und schnelle Art und Weise erstellen kann. Denkbar wäre in diesem Zusammenhang z.B. das Anlegen einer Datenbank, in der häufig verwendete Standard-SVG-Komponenten wie z.B. Skalen, Messkomponenten, Eingabefelder etc. abgelegt werden können, um den Benutzer die Arbeit bei der Erstellung neuer SVG-Graphiken zu erleichtern. Die Verwendung einer Datenbank für eine solche Bibliothek mit Standard-SVG-Komponenten bietet sich gerade deswegen an, weil SVG-Dateien nur aus Text bestehen.

Als dritten Punkt gäbe es noch die Möglichkeit, zusätzlich zur KEV-*Observer*-Komponente eine *Producer*-Komponente zu implementieren, die es ermöglicht auf Benutzereingaben zu reagieren und der Simulation somit ein „Feedback“ zu geben. In diesem Zusammenhang denkbar wären da zum Beispiel (z.B.) einzelne SVG-Elemente, die als Button dienen könnten um bestimmte Abläufe innerhalb der Simulation zu steuern. Auch die Aktivierung/Deaktivierung einzelner SVG-Elemente während der laufenden Simulation wären dadurch möglich, womit sich das Verhalten zur Laufzeit aktiv beeinflussen ließe.

Neben den oben genannten möglichen Erweiterungen spielen gerade auch die Benutzer für die Weiterentwicklung ein große Rolle, da durch die Benutzung von KEV sich im Laufe der Zeit sicherlich noch einige Verbesserungsvorschläge seitens der Benutzer zu diesem Thema ergeben werden, die momentan aufgrund mangelnder Erfahrung noch gar nicht existieren.

## 7.3 Erweiterung des Mappings

Das Mapping von KEV wurde bewusst so gestaltet, dass sich neue Animationen jederzeit einfach hinzufügen lassen. Dank EMF lassen sich nach Anpassung des EMF-Modells auf Knopfdruck die entsprechenden JAVA-Klassen generieren, so dass nur noch die animationsrelevanten Methoden `apply()` und `initialize()` sowie ggf. Hilfsmethoden vom Programmierer implementiert werden müssen. Hierdurch ist eine schnelle und einfache Erweiterbarkeit des Mappings gewährleistet.

# 8 Fazit

Die Portierung der alten Model-GUI hin zum neuen KEV-Plugin gab die Möglichkeit das alte Mapping-Konzept zu überarbeiten und generischer zu machen. Es wurde gezeigt, dass die Verwendung von EMF als Grundlage dem Programmierer eine Reihe von Aufgaben abnimmt, so dass sich dieser nur noch um die Entwicklung und Implementierung von Code für neue Animation kümmern muss. Da das Mapping die zentrale Schnittstelle zwischen Animation (SVG-Graphik) und Simulation darstellt, ist die leichte Erweiterbarkeit ein wichtiger Aspekt. Dank den durch EMF generierten Editor und der Speicherung der Mapping-Dateien im XML-Format ist die Erstellung eines entsprechenden Mappings einfach realisierbar. Einen komfortablen Editor zu erstellen, war nicht die Aufgabe der vorliegenden Ausarbeitung, weshalb bei der Erstellung auch nur der durch EMF generierte Editor verwendet wurde. Des Weiteren wurde gezeigt, dass der mitgelieferte Editor zwar gut zum Erstellen von Mapping-Dateien geeignet ist, nicht jedoch zum schnellen Editieren, wofür – dank XML – ein einfacher Text-Editor besser geeignet wäre.

Die leichte Erweiterbarkeit von Animationen mittels EMF kombiniert mit der Erweiterbarkeit der Eclipse-IDE lässt die Möglichkeit offen, welche Einsatzgebiete sich die Open-Source-Gemeinschaft – für KIELER im Allgemeinen und das KEV-Plugin im Speziellen – noch ausdenken wird.

# A Literaturverzeichnis


[1] ADOBE SYSTEMS GMBH: *Adobe Flash Platform*. http://www.adobe.com/de/flashplatform/

[2] APACHE SOFTWARE FOUNDATION: http://www.apache.org/

[3] APACHE SOFTWARE FOUNDATION (Hrsg.): *Batik SVG Toolkit*. http://xmlgraphics.apache.org/batik/

[4] ECMASCRIPT LANGUAGE SPECIFICATION (ECMA-262): http://www.ecma-international.org/publications/standards/Ecma-262.htm

[5] EMF (ECLIPSE MODELING FRAMEWORK): http://www.eclipse.org/modeling/emf/

[6] IBM ILOG ELIXIR V2.5: *Datenvisualisierung für Adobe Flex und Adobe AIR*. http://wwis-dubc1-vip60.adobe.com/de/products/flex/ibmilogelixir/

[7] INTERNATIONAL BUSINESS MACHINES (IBM): http://www.ibm.com/

[8] JSON (JAVASCRIPT OBJECT NOTATION): http://www.json.org/

[9] KIELER: *Kiel Integrated Environment for Layout for the Eclipse RCP*. http://www.informatik.uni-kiel.de/rtsys/kieler/

[10] LANGFELD, Dorothee: *Entwicklung einer SVG Web Mapping Applikation zur Visualisierung von Geoinformationen.* Diplomarbeit, Institut für Informatik, Universität Osnabrück, 2006. http://www.informatik.uni-osnabrueck.de/prakt/pers/dipl/dlangfel.pdf

[11] MODELLBAHNANLAGE DER UNVERSITÄT KIEL: http://www.informatik.uni-kiel.de/~railway/

[12] MOTIKA, Christian: *Modellbasierte Umgebungssimulation für verteilte Echtzeitsysteme mit flexiblem Schnittstellenkonzept – Fallstudie einer Bahn-Anlage.* Studienarbeit, Institut für Informatik, Christian-Albrechts-Universität zu Kiel, 2007. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-st.pdf

[13] STEINBERG, David ; BUDINSKY, Frank ; PATERNOSTRO, Marcelo ; MERKS, Ed: *EMF: Eclipse Modeling Framework (2nd Edition) (Eclipse)*. 2nd Revised edition (REV). Addison-Wesley Longman, Amsterdam, 2009. – ISBN 9780321331885

[14] SVG WORKING GROUP (W3C.ORG): `http://www.w3.org/TR/SVG11/`

[15] W3C.ORG: *Cascading Style Sheets Level 2 Specification*. `http://www.w3.org/TR/CSS2/`