DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg
Campus Friedrichshafen

TANNER

Darstellung von Vektorgrafiken in modernen Browserumgebungen

Projektarbeit T2000

im Studiengang **TIT09**
an der DHBW Ravensburg
Campus Friedrichshafen

Florian Peschka

7. Oktober 2011

Bearbeitungszeitraum	KW 14 / 2011 - KW 36 / 2011
Matrikelnummer	6192194
Partnerunternehmen	TANNER AG, Lindau
Betreuer im Partnerunternehmen	Bernd Lehmanski

Erklärung

gemäß §*5(2)* der Studien- und Prüfungsordnung DHBW Technik vom 18. Mai 2009.

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel

Darstellung von Vektorgrafiken in modernen Browserumgebungen

selbständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt, keine anderen als die angegebenen Hilfsmittel benutzt und wörtliche sowie sinngemäße Zitate als solche gekennzeichnet habe.

Florian Peschka

Kurzfassung

Vektorbasierte Grafikformate sind schon seit Jahren wichtige Instrumente, um visuelle Informationen auszutauschen. Gerade zur Darstellung industrieller Maschinen ist es enorm wichtig, keine Details aufgrund von Kompression oder Skalierbarkeit einzubüßen, wie es bei den traditionellen Rasterformaten der Fall ist. Bis vor wenigen Jahren waren diese Formate jedoch hauptsächlich in Prozesse integriert, die ausschließlich offline arbeiteten.

Mit der Entwicklung von *SVG* und *WebCGM* wurden vektorbasierte Formate auch für den Einsatz in Webseiten und Browsern fit gemacht. Diese beiden vom *W3C* spezifizierten Formate sind für diesen Einsatz entwickelt worden und ermöglichen es, voll skalierbare Grafiken auf Webseiten zu präsentieren.

Diese Projektarbeit beschäftigt sich mit dem Einsatz dieser Formate in modernen Browserumgebungen und inwiefern welches Format geeignet ist, in online verfügbaren Ersatzteilkatalogen Grafiken darzustellen. Es wird dargestellt, wie sich die Formate grundlegend unterscheiden und welche Auswirkungen diese Unterschiede auf den Einsatz der Formate für Applikationen in einem technisch-redaktionellen Umfeld haben.

Dafür wird eine Beispielapplikation entwickelt, die einen solchen Ersatzteilkatalog mit allen benötigten Funktionen darstellt. Für den Einsatz des Katalogs ist es von entscheidender Bedeutung, den Aufwand für Endbenutzer so gering wie möglich zu halten, gerade im Hinblick auf die Installation oder Konfiguration des Systems.

Die beiden Formate werden verglichen und eine Empfehlung wird ausgesprochen, welches Format für Kataloge in Browserumgebungen besser geeignet ist. Dabei ist festzuhalten, dass dies nur die Entwicklung eines solchen Systems betrifft – die Eignung für technische Grafiken im Allgemeinen soll nicht Hauptbestandteil dieser Arbeit sein.

Vielmehr geht es um den direkten Einsatz in einem funktionierenden System für Ersatzteilkataloge und wie sich die Formate in den bereits bestehenden Prozess der Erstellung der Kataloge einbinden lassen. Dabei muss beachtet werden, welche Anpassungen nötig wären, um die einzelnen Formate komplett zu unterstützen.

Die Arbeit wurde vor dem Hintergrund erstellt, die Zukunftsfähigkeit des SVG-Formates festzustellen und mögliche Alternativen darzulegen, die weiterhin die Unterstützung vektorbasierter Grafiken in Ersatzteilkatalogen *ohne Mehraufwände für Benutzer* sichern.

Zusätzlich wird ein beispielhafter Einsatz der Funktionsbibliothek und die damit verbundenen benötigten Anpassungen eines Generatorprozesses eines Kunden der *TANNER AG* dargestellt und welche damit verbundenen Prozesse von der Grafikerstellung bis hin zum fertigen Katalog angepasst werden müssen.

Abstract

Vector based graphic formats have been important instruments to transport visual information for years. Especially for displaying industrial machines it is important that no details are lost due to compression or scaling, as is the case when using raster based formats. Until a few years ago these formats have been included solely in processes which worked offline.

By developing *SVG* and *WebCGM*, the *W3C* has made vector based formats applicable for use in web pages and browsers. These two formats have been developed with the exact goal to bring scalable graphics to the web.

This report deals with the application of these formats in modern browser environments and the question about which format is the best on to be used in online spare parts catalogues. The differences between the formats are shown and how they influence the application of a format in a technical-editorial environment.

Therefore a demo application is developed which depicts a spare parts catalogue with all necessary functions. It's very important that using this catalogue causes the user as vew effort as possible as far as installing or configuring the system is concerned.

Both formats are compared with each other to determine which format is the best fit for online catalogues. The target is the development of such a system, but the general suitability of a format to display technical graphics is not be the central part of this report.

Instead, the focus is on application of a format in a spare parts catalogue system, how this format can be integrated within an already existing process of creating the catalogue and what necessary changes have to be made in order to integrate the format.

The report is based on the decision wether the *SVG* formats sustainability is given or not and which alternatives do exist in order to use vector based graphic formats in online spare parts catalogues *without much additional effort for users.*

In addition the application of the created library is demonstrated by modifying and adapting processes of a *TANNER AG* customer. All related business and programming process steps are analysed and optimised: from the creating the graphics up to completing the catalogue.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Allgemeines . . . 1
1.2 Vektorgrafiken . . . 1
1.3 Computer Graphics Metafile . . . 2
1.4 Scalable Vector Graphics . . . 3

2 Zielsetzung **4**
2.1 Allgemeines . . . 4
2.2 Definition der Anforderungen . . . 4

3 Lösungsansatz **6**
3.1 WebCGM . . . 6
3.2 SVGWeb . . . 6
3.3 JavaScript-Bibliotheken . . . 8

4 Aufbau einer Beispielapplikation **10**
4.1 WebCGM . . . 10
4.2 SVG . . . 10
4.3 jQuery . . . 12
4.4 SVGWeb . . . 13
4.5 Umsetzung . . . 16
4.5.1 Aufbau einer Funktionsbibliothek . . . 16
4.5.2 Hervorheben von Elementen . . . 17
4.5.3 Navigation innerhalb der Grafik . . . 20
4.5.4 Verlinken auf andere Grafiken . . . 23
4.5.5 Einbinden von Rastergrafiken . . . 24
4.5.6 Usability . . . 25
4.6 Anpassung und Zukunftsfähigkeit . . . 28

5 Generator **33**
5.1 Allgemeines . . . 33
5.2 Beschreibung . . . 33
5.3 Anpassungen und Verbesserungen . . . 35

6 Diskussion **40**
6.1 Analyse der Lösungen . . . 40
6.1.1 WebCGM . . . 40
6.1.2 SVGWeb . . . 41

6.2 Schlussfolgerung 42
6.2.1 Proprietäre ETKs 42
6.2.2 SVGWeb zur Anzeige in Browserumgebungen 42
6.2.3 SVG zur Anzeige vektorbasierter Grafiken 42
6.3 Ausblick 43

Literaturverzeichnis **44**

Abbildungsverzeichnis **45**

Listing-Verzeichnis **46**

1 Einleitung

1.1 Allgemeines

Im Rahmen der technischen Dokumentation für industrielle Maschinen spielen Grafiken eine entscheidende Rolle. Gerade für komplizierte Geräte, Bau- und Ersatzteile sowie Bedienungsanleitungen zum Auf- und Umbau eben dieser Geräte ist es von entscheidender Wichtigkeit, dass eine visuelle Repräsentation des Gegenstandes vorliegt. Um dies zu gewährleisten, muss es Techniken geben, mit welchen sich komplexe Gegenstände möglichst exakt und zugleich schnell begreifbar darstellen lassen.

Zu diesem Zweck sind Umgebungen für Computer Aided Design (CAD) entwickelt worden, mit deren Hilfe sich jedwede Art von (technischen) Gegenständen rendern, modellieren und animieren lässt. Diese Grafiken sind dreidimensional, was sich zum direkten Vorführen besonders gut eignet, da sie leicht in jede Richtung gedreht, vergrößert und einzelne Gruppen von Bauteilen ausgeblendet oder hervorgehoben werden können. All dies ermöglicht es dem Anwender, Maschinen virtuell auseinander zu bauen, um sich deren Zusammensetzung und Funktionsweise im Detail anzusehen.

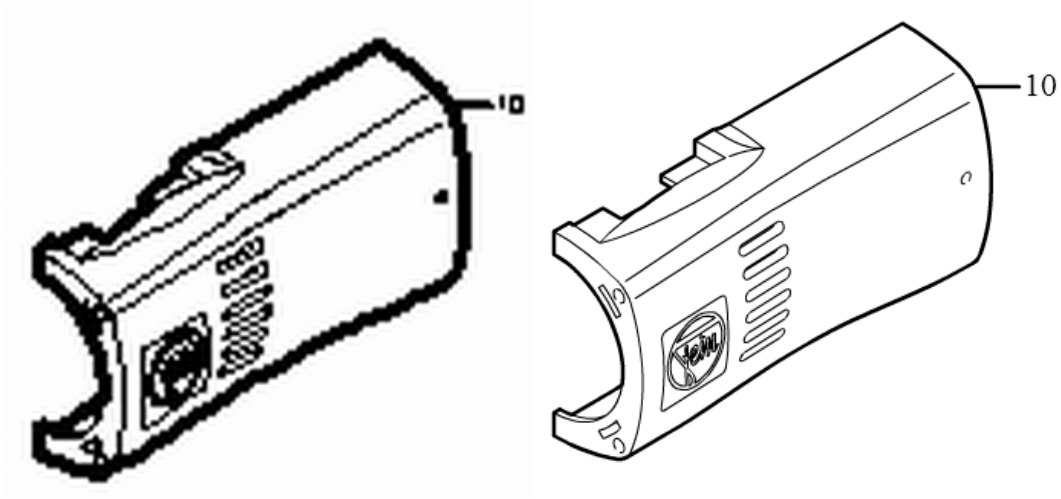
1.2 Vektorgrafiken

Um die Vorteile dieser 3D-Grafiken auch in gedruckter Form oder in Dokumenten nutzen zu können, können aus den oben genannten CAD-Dateien so genannte Vektorgrafiken erzeugt werden. Diese bilden den Gegenpol zu den wohlbekannten Rastergrafiken (z. B. Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG) oder Portable Network Graphics (PNG)). Diese Formate haben eines gemeinsam: Sie speichern die Daten, aus welcher die Grafik besteht, als Raster von Pixeln. D. h., dass beim Rendern der Grafik durch den Rechner des Betrachters die Pixel der Grafik genau auf die Pixel des Bildschirms angepasst werden, sodass ein Pixel der Grafik genau einem Pixel des Bildschirms entspricht.

Im Gegensatz dazu speichern Vektorformate (z.B. Computer Graphics Metafile (CGM), Encapsulated PostScript (EPS) oder Scalabe Vector Graphic (SVG)) nicht die exakte Pixelposition, sondern vielmehr Pfade (Vektoren) und deren Verhältnisse zueinander. So ist ein Strich in einer Rastergrafik lediglich eine Aneinanderreihung von Pixeln, die sich aus der Entfernung als Strich darstellen, wohingegen ein Strich in einer Vektorgrafik tatsächlich ein eigenes Element „Strich“ (`<line>`) ist.

Dadurch entsteht ein enormer Vorteil: Vektorgrafiken können in beliebig hoher Auflösung betrachtet werden, ohne dass durch das Vergrößern Unschärfen entstehen, wie sie bei Rastergrafiken schon bei kleinsten Zoomstufen auftreten. Vektoren beschreiben die Grafik lediglich, wodurch Strichstärken immer die gleiche Pixeldicke auf dem Bildschirm des Betrachters haben.

So lassen sich auch kleinste Details (z.B. die Seriennummer eines Bauteils) direkt sichtbar in der Grafik speichern, ohne dass sie dadurch eine enorme Dateigröße erhält. Würde man dies in einer Rastergrafik versuchen, so wäre sie leicht mehrere Gigabyte groß, damit sich das Detail noch lesbar darstellen lässt (s. Abb. 1).



(a) Zoom um 300% bei Raster-Grafiken (b) Zoom des gleichen Bildes als Vektor-Grafik

Abbildung 1: Vergleich der Zoomstufen

1.3 Computer Graphics Metafile

Einer der ersten Standards zu einem Datentyp für vektorbasierte Grafiken ist die Computer Graphics Metafile (CGM), welche 1987 durch das American National Standards Institue (ANSI) und die Internationale Organisation für Normung (ISO) in der Norm ISO/IEC 8632-1 festgehalten wurde. Für dieses Format wurde das eindeutige Ziel gesetzt, aktuelle Anforderungen der Industrie zu berücksichtigen und dadurch das Austauschen von Grafiken zwischen verschiedenen Firmen, Organisationen, Programmen und Betriebssystemen zu erleichtern [ISO99, Abschnitt 0.1].

CGM ist ein Byteformat, d.h. die Daten liegen codiert vor und sind damit nur mit geeigneten Programmen les- und editierbar, die diesen Bytecode decodieren können. Dadurch ist der Einsatz von CMG-Grafiken vor allem dort häufig anzutreffen, wo technische Zeichnungen erstellt, bearbeitet und zwischen mehreren Parteien ausgetauscht werden müssen. Durch den sehr strikten Standard ist CGM ein enorm robustes Format, das strengen Regeln genügen muss und dadurch gerade für den Austausch geeignet ist.

Das World Wide Web Consortium (W3C) hat den Standard des CGM auch in einen für Webseiten und Internettechnologie brauchbaren Dialekt spezifiziert, das *WebCGM*. Dieser Standard definiert, welchen bestimmten Regeln CGM-Grafiken folgen müssen, um über Browser, Plugins oder spezielle Viewer einsehbar zu sein. Es wurde dabei besonderer Wert darauf gelegt, einerseits dem komplexen Standard des CGM gerecht zu werden, andererseits aber auch eine gewisse Einfachheit sicherzustellen, sodass die Implementierung von Viewern in Browsern und Webseiten den Entwicklern möglichst einfach ermöglicht wird.

1.4 Scalable Vector Graphics

Um eine Alternativen zu dem bereits etablierten CGM und dessen Umsetzung für Webtechnologien WebCGM zu schaffen, spezifizierte das W3C im Jahr 2001 die Scalable Vector Graphics (SVG). Auch dieses Dateiformat hat die Aufgabe, vektorbasierte Grafiken zu speichern. Das Hauptaugenmerk lag bei diesem Standard jedoch darauf, dass das Format nicht in Bytecode, sondern als textuelle Repräsentation der Grafik vorliegt. Die Struktur dieses Textes ist an die Extensible Markup Language (XML) gebunden, dessen Definition ebenfalls vom W3C stammt[1].

Während bei (Web)CGM die Bytestruktur bestimmt, wie Linien aussehen und wo sie positioniert sind, wird dies bei SVG-Grafiken durch XML-Tags (z.B. `<line>` für eine Linie) erledigt. Dadurch ist eine SVG-Grafik gleichzeitig ein normales Text-Dokument, das mit jedem Text-Editor geöffnet und bearbeitet werden kann. Die Codierung der Grafik ist für Menschen les- und vor allem verstehbar. Da eine SVG-Grafik durch die XML-Basis zusätzlich gestützt wird, ist sie auch durch den Mechanismus der Validierung gegen eine Document Type Definition (DTD) leicht auf Fehler zu überprüfen, noch bevor sie geladen und angezeigt wird.

Da SVG vom *W3C* entwickelt wurde, liegt die Verwendung des Formates im Web nahe. Daher bot sich die XML-basierte Umsetzung an, da auch das Standard-Format des Web, die Hypertext Markup Language (HMTL), einen XML-ähnlichen Dialekt besitzt[2]. So lässt sich eine SVG-Grafik z.B. als reiner Text in ein HTML-Dokument einbinden oder auch als Referenz auf eine *.svg-Datei.

Allgemein wird SVG eher für kreative Grafiken eingesetzt, da es weitreichende Unterstützung für Animationen, spezielle grafische Effekte wie Shader, Schatten, Übergänge etc. und die Möglichkeit der Einbindung von dynamischen und interaktiven Elementen via JavaScript bietet [Wor03, Abschnitt 1.1].

[1] http://www.w3.org/TR/2008/REC-xml-20081126/

[2] Der XML-Dialekt von HTML wird als XHTML bezeichnet. Die „normalen“ HTML-Dialekte sind stärker durch die Standard Generalized Markup Language (SGML) beeinflusst. Zwar basiert XML selbst auch auf SGML, jedoch erweitert es dieses um einige sehr strenge Regeln, denen die HTML-Dialekte nicht immer gerecht werden.

2 Zielsetzung

2.1 Allgemeines

Das Ziel dieser Projektarbeit soll sein, die Zukunftsfähigkeit der vorgestellten Techniken zu evaluieren. Dabei soll nicht die prinzipielle Eignung zur Darstellung technischer vektorbasierter Grafiken im Vordergrund stehen, sondern vor allem die wirtschaftlich sinnvolle Verwirklichung der Darstellung solcher Grafiken in modernen Browserumgebungen. Hierfür muss geprüft werden, ob und wie sich die Grafiken in bestehende Web-Systeme einbinden lassen und inwiefern die Grafiken in Echtzeit im Browser manipuliert werden können, z. B. in Größe, Position oder Hervorhebung und Ausblendung von einzelnen Teilen der Grafik.

All dies geschieht vor dem Hintergrund des Auftrags, einen Ersatzteil-Katalog (ETK), in welchem vektorbasierte Explosionsgrafiken eingesetzt werden, zu realisieren, der für Endanwender von technischen Maschinen als Bezugsquelle für Ersatzteile dient. In diesem Katalog sollen die Explosionsgrafiken angezeigt und über eine dynamische Auswahl einzelne Teile hervorgehoben werden können. So können Benutzer ein bestimmtes Teil ihrer Maschine gezielt auswählen, woraufhin sie die exakte Kennnummer des Ersatzteils erhalten und dieses nachbestellen können.

ETKs sind oft über Firmenwebseiten zugänglich, wodurch die Anforderung entsteht, dass die Grafiken mit einem möglichst geringen Aufwand für den Endanwender sichtbar gemacht werden sollen. Gleichzeitig muss jedoch beachtet werden, durch welche Prozesse eine Grafik so weit aufbearbeitet werden muss, dass sie sich für die Darstellung in einem solchen Katalog überhaupt eignet.

Als Ergebnis wird eine Aussage erwartet, aus welcher hervorgeht, wie man einen solchen Katalog und allgemein vektorbasierte Grafiken in Browserumgebungen zukunftssicher gestalten kann, und eine Empfehlung, auf welche Technologie man zur Darstellung von vektorbasierten Grafiken setzen sollte.

2.2 Definition der Anforderungen

Für den direkten Kundeneinsatz sind die folgenden Anforderungen definiert. Sie sollen bei der Auswahl der Technologie berücksichtigt werden. Die allgemeine Aussage über die Eignung einer Technologie für den Einsatz zur Darstellung von Vektorgrafiken in modernen Browserumgebungen bleibt davon jedoch unberührt.

Implementierung in Erstellungsprozesse Es muss gewährleistet sein, dass die Erstellung der Grafik aus CAD-Daten in einen Prozess eingebunden werden kann, der besagte Daten möglichst aufwandsarm portieren und erstellen kann. Dabei muss auch die Manipulation der Grafiken im Nachhinein bedacht werden.

Eignung für technische Grafiken Das Format der Grafiken muss für die Darstellung von komplexen technischen Zeichnungen geeignet sein. Dazu gehören unter anderem die grundsätzliche Skalierbarkeit der Grafik sowie grundlegende geometrische Formen wie Linien, Rechtecke, Kreise, die beliebig positioniert, eingefärbt und manipuliert werden.

Manipulation der Grafiken Um dem Endanwender die Möglichkeit zu geben, bestimmte Bauteile einfach hervorzuheben, muss es möglich sein, die Grafiken während der Anzeige im Browser zu verändern, um so z. B. Bereiche anders einzufärben oder ein/auszublenden. Dies muss ohne ein erneutes Laden der Grafik möglich sein, um die Last des Servers gering zu halten.

Integration im Browser Die ETKs werden vorrangig in Browserumgebungen eingesetzt, daher muss es ohne Aufwand möglich sein, die Grafiken in eine Browserumgebung einzubinden. Dies muss so ablaufen, dass für den Endanwender kein Mehraufwand dadurch entsteht, wenn er die Grafiken betrachten will.

Usability für Endanwender Für den Endandwender muss die Bedienung des ETKs als Gesamtkonzept stimmig sein, d. h. die Grafiken müssen durch dynamische Manipulation der Inhalte an die Bedürfnisse jedes Benutzers anpassbar sein. Dies muss vor allem auch dann möglich sein, wenn die Grafik bereits im Browser des Anwenders geladen wurde.

3 Lösungsansatz

3.1 WebCGM

Der WebCGM-Standard ist ein Profil des CGM-Standards, welcher vom W3C entwickelt wurde, um die Stärken von CGM auch für Anwendungen im Web zugänglich zu machen. Dabei wurde besonders Wert darauf gelegt, die Definition so strikt wie möglich zu halten und gleichzeitig den Entwicklern möglichst viele der bereits bekannten Methoden von CGM weiterhin zu bieten [Wor10, Abschnitt "Abstract"].

Das Profil wurde erstmals vom W3C 1999 in der Spezifikation 1.0[3] herausgegeben. Hier wurden bereits die wichtigsten Eigenschaften von CGM in ein für Web-Applikationen brauchbares Format gebracht. Aktuell ist die Spezifikation 2.1[4].

Um WebCGM in Browsern anzuzeigen, existieren zahlreiche Plugins und Viewer. Die meisten davon unterliegen kommerziellen Lizenzen und sind so dem privaten Anwender nicht zugänglich. Auch hier spiegelt sich das alte Bild von CGM als reinem Industrie-Format wider. Klassische CGM-Grafiken werden wie bereits erwähnt aus CAD-Systemen exportiert, wodurch diese jedoch oftmals mit proprietären Erweiterungen versehen sind oder sogar ein eigenes Profil enthalten, welches nicht direkt mit dem WebCGM-Profil konform ist.

Dies erschwert die Portierung von CGM zu WebCGM, um z. B. Zeichnungen direkt aus der Produktion in ein webtaugliches Format zu bringen und sie zu präsentieren. Daher wird empfohlen, WebCGM als Basisprofil zu verwenden und auf diesem aufbauend die eigenen proprietären Profile zu definieren, um die eigenen Entwicklungen weiterhin behalten zu können [Wor10, Abschnitt 1.7].

Um CGM-Grafiken in Browsern anzuzeigen, werden ausnahmslos zusätzliche Plugins benötigt, die der Benutzer installieren muss[5].

Dadurch, dass die meisten dieser Programme von Firmen entwickelt werden und daher nur in kompilierter Form vorliegen, ist es schwer, eigene Implementierungen zu entwerfen. Die einzige Möglichkeit ist, ein eigenes Programm zu entwerfen, welches die Darstellung von WebCGM-Grafiken in Browsern ermöglicht.

Ein selbst programmiertes Plugin hat jedoch erneut den Nachteil, dass stets eine Installation des Benutzers erforderlich ist, bevor er sich die Grafiken in seinem Browser betrachten kann.

3.2 SVGWeb

Die Darstellung von SVG-Grafiken wird zum Zeitpunkt der Erstellung dieses Dokuments in den meisten modernen Browsern unterstützt. Diese native Darstellung ist größtenteils unvollständig [Sch] (im Vergleich zur Spezifikation des Standards) und liefert unterschiedlichste Ergebnisse in den verschiedenen Engines.

[3] http://www.w3.org/TR/1999/REC-WebCGM-19990121/
[4] http://www.w3.org/TR/2010/REC-webcgm21-20100301/
[5] http://www.cgmopen.org/webcgm/viewers.html

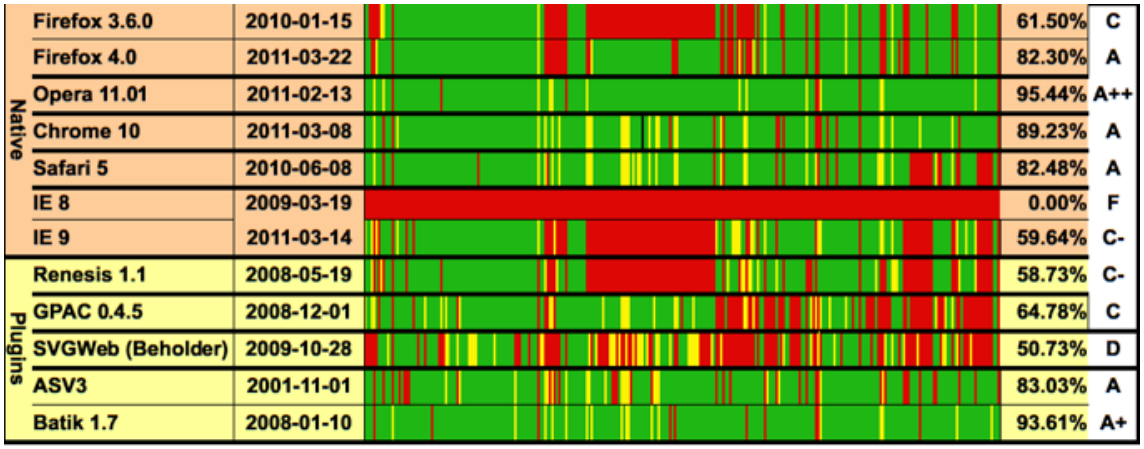


Abbildung 2: Nativer SVG-Support verschiedener Viewer

Um SVG in Browsern anzuzeigen, die es nicht nativ unterstützen, gibt es ebenfalls zahlreiche Plugins und Browser-Erweiterungen, die der Benutzer installieren muss[6].

Ein Gegenpol zu dieser Bewegung ist das *GoogleCode-Project* SVGWeb[7]. Es basiert auf einer Kombination aus JavaScript und ActionScript. Zwar ist letzteres auch ein installierbares Plugin für Browser, jedoch ist der *Adobe Flash Player* in fast jedem internetfähigen Rechner installiert [Ado], wodurch für nahezu alle Endbenutzer effektiv keine Installation zusätzlicher Software nötig ist.

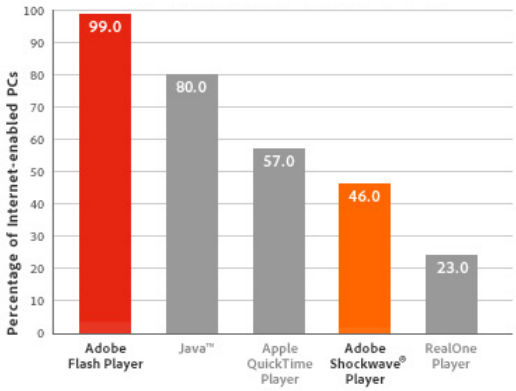


Abbildung 3: Marktdurchdringung des Adobe Flash Players

SVGWeb besteht aus zwei Kernkomponenten: der JavaScript-Bibliothek und dem Flash-Renderer. Über die JavaScript-Bibliothek kann herausgefunden werden, ob der Browser des Benutzers eine native SVG-Unterstützung bietet. Wenn das der Fall ist, wird standardmäßig auch diese native Darstellung verwendet. Sollte dies nicht der Fall sein, wird über eine asynchrone Anfrage der Flash-Renderer mit der SVG-Datei angefragt.

[6] Z. B. den Adobe SVG Viewer - http://www.adobe.com/svg/viewer/install/
[7] http://code.google.com/p/svgweb/

Dieser parst die Grafik und übersetzt deren Inhalt in eine Flash-Darstellung, welche dann im Browser anstelle der SVG-Grafik eingesetzt wird.

Diese Software ermöglicht dadurch auch Browsern ohne native SVG-Unterstützung das Anzeigen von SVG-Grafiken. Das Projekt selbst ist ein OpenSource-Projekt, wodurch die individuelle Anpassung der Software an Kundeninteressen einfach möglich ist. SVGWeb ist durch eine *Apache License 2.0*[8] geschützt, was zum Verändern des Codes berechtigt.

Zwar ist auch hier die Unterstüzung des SVG-Standards nicht zu 100% gewährleistet, jedoch sind die grundlegenden Funktionen und vor allem jene, die für technische Grafiken von besonderer Bedeutung sind, ausreichend vorhanden [Sch].

3.3 JavaScript-Bibliotheken

Um die dynamische Generierung und Veränderung von Inhalten und Vektorgrafiken zu gewährleisten, wird oft auf JavaScript oder ECMA-Script gesetzt. Diese Programmiersprache ist in nahezu allen Browsern implementiert [Ham] und erlaubt, Inhalte von Webseiten zu verändern, auf Benutzereingaben und Events zu reagieren, ohne die angezeigte Seite neu zu laden.

Auch wenn ECMAScript standardisiert ist, hat jeder Browserhersteller eigene Implementierungen geschaffen, die mehr oder weniger bestimmte Sprachkonstrukte unterstützen und Inkompatibilitäten zum Standard aufweisen. Dies erschwert die Entwicklung von JavaScript-Anwendungen enorm, da immer der Anspruch besteht, dass die Anwendung auf jedem Browser gleich oder zumindest ähnlich agieren und reagieren soll.

Um Entwicklern dies zu erleichtern, gibt es zahlreiche Funktionsbibliotheken, welche die Funktionalitäten von JavaScript kapseln und so eine einheitliche Befehlsschnittstelle schaffen, mit der Programme unabhängig von der darunter liegenden Browser-Engine entwickelt werden können.

Alle diese Bibliotheken zeichnen sich durch eine hohe Abstraktion von Befehlen aus, die dem Programmierer Zugriff auf Funktionen bieten, ohne dass er Kenntnis von dessen interner Verarbeitung benötigt.

Listing 1: Beispiel ohne JavaScript-Bibliothek

```
1 var elements = document.getElementsByTagName("p");
2 for (j = 0; j <= elements.length; j++) {
3   elements[j].innerHTML = "Element #" + j;
4 }
```

Listing 2: Beispiel mit JavaScript-Bibliothek (jQuery)

```
1 $("p").each(function(index) {
2   $(this).html("Element #" + index);
3 });
```

Bereits in diesem sehr einfachen Beispiel erkennt man, dass die Verwendung der Bibliothek in Listing 2 wesentlich kürzer zu schreiben und auch leichter zu verstehen ist.

[8] http://www.apache.org/licenses/LICENSE-2.0

Im Hintergrund wird der Code durch die Bibliothek an die JavaScript-Implementierung des aufrufenden Browsers angepasst, sodass die Funktion unabhängig vom Browser gleich ausgeführt wird. Durch das Einsetzen einer solchen Bibliothek (auch Framework genannt), wird der Prozess des Programmierens auf das tatsächliche Lösen von Problemen verlagert. Der Programmierer muss nur noch die Funktionalität programmieren und nicht mehr selbst testen und sicherstellen, dass sein Code auf jedem Browser gleich funktioniert.

4 Aufbau einer Beispielapplikation

4.1 WebCGM

Der Einsatz einer WebCGM-Technik war während des Entwicklungszeitraums keine Möglichkeit aufgrund mangelnder offener Plugins, welche eine ansprechbare API zur Verfügung stellen. Die Beispiel-Applikation wird daher nur für den Einsatz mit SVG-Grafiken in Verbindung mit SVGWeb konzipiert werden, ist aber theoretisch mit Modifikationen auch in der Lage, WebCGM-Grafiken zu manipulieren.

So ist es beispielsweise mit dem Larson CGM Viewer[9] nur möglich, Elemente zu referenzieren und diese ein/auszublenden. Ein Verlinken von Objekten und die direkte Interaktion mit der Grafikdatei oder die Manipulation derselbigen ist nicht möglich [Lar].

4.2 SVG

Um die Manipulation der SVG-Grafiken durch JavaScript zu ermöglichen, müssen diese einige Konventionen erfüllen. Besonders die Hervorhebung von komplexen Bauteilen kann nur erfolgen, wenn die SVG-Grafiken korrekt gruppiert worden sind. Dies ist insbesondere dann ein Problem, wenn die Grafiken durch einen automatisierten Prozess erstellt werden, bei welchem möglicherweise die Gruppen verloren gehen.

Eine SVG-Grafik mit der folgenden Struktur ist zur Verwendung nicht geeignet.

Listing 3: Ungruppierte SVG-Grafik

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <svg xmlns:svg="http://www.w3.org/2000/svg"
3    xmlns:xlink="http://www.w3.org/1999/xlink"
4    xmlns="http://www.w3.org/2000/svg" xml:space="preserve" version="1.0"
5    width="200" height="200" viewBox="0 0 200 200">
6    <rect width="100" height="100"
7      x="50" y="50" id="rect_1" style="fill:#000000;" />
8    <text x="55" y="50" id="text_1">Hervorheben Rect1</text>
9    <rect width="100" height="100"
10     x="50" y="50" id="rect_2" style="fill:#000000;" />
11   <text x="55" y="50" id="text_2">Hervorheben Rect2</text>
12 </svg>
```

Die Problematik besteht darin festzustellen, welche Element der SVG-Grafik zu welchem Bauteil gehören. Das einfache Beispiel oben kann nur sehr begrenzt diesen Sachverhalt verdeutlichen. In den produktiv eingesetzten Grafiken sind bis zu 20.000 einzelne `<path>`- oder `<line>`-Elemente vorhanden, von denen jedes eine eindeutige ID aufweist. Auch wenn man so die einzelnen Elemente einfach identifizieren kann, ist es nicht möglich aus den Informationen des Elements eindeutig nachzuweisen, zu welchem Bauteil diese Linie nun gehört.

[9] http://www.cgmlarson.com/Firefox_CGM_Viewer_Plugin.html

SVG bietet aus diesem Grund die Möglichkeit, Gruppen zu definieren, welche Elemente zusammenfassen, die zusammengehören. Da auch diese Gruppen mit IDs versehen werden, ist so eine eindeutige Zuordnung der Linien zu Bauteilen möglich.

Listing 4: Gruppierte SVG-Grafik

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns:svg="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns="http://www.w3.org/2000/svg" xml:space="preserve" version="1.0"
  width="200" height="200" viewBox="0 0 200 200">
  <g id="group_1">
    <rect width="100" height="100"
      x="50" y="50" id="rect_1" style="fill:#000000;" />
    <text x="55" y="50" id="text_1">Hervorheben Rect1</text>
  </g>
  <g id="group_2">
    <rect width="100" height="100"
      x="50" y="50" id="rect_2" style="fill:#000000;" />
    <text x="55" y="50" id="text_2">Hervorheben Rect2</text>
  </g>
</svg>
```

Durch die Gruppierung ist nun jedes Element Teil eines Bauteils, welches durch die Gruppe identifiziert wird.

Eine weitere Anforderung an die zu verwendende SVG-Grafik ist die Validität. Fehler in der XML-Struktur der Grafiken können dazu führen, dass die SVGWeb-Bibliothek die Bilder nicht rendern kann, was entweder zu unschönen Fehlermeldungen oder zu einer fehlerhaften Darstellung führt. Durch den sehr ausführlich definierten Standard von SVG besteht hier wenig Spielraum.

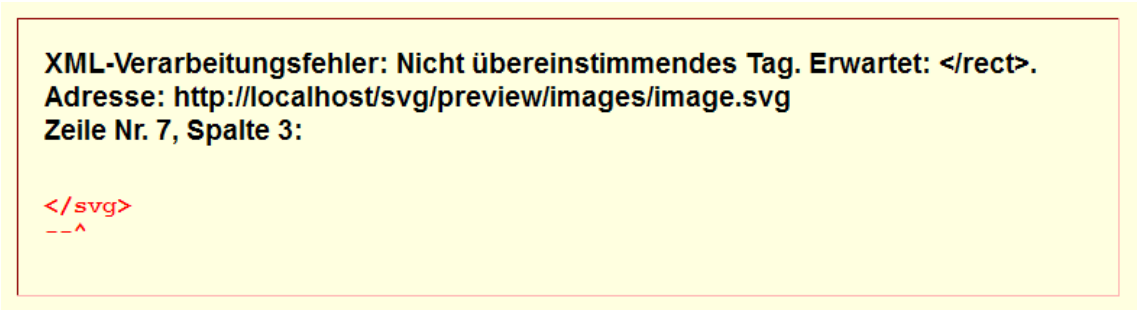


Abbildung 4: XML-Verarbeitungsfehler bei nicht valider SVG-Grafik

Für die Beispielapplikation wurden originale SVG-Grafiken eines Kunden verwendet, für welchen ein auf diesem Beispiel basierender ETK angefertig werden soll. Diese Explosionsgrafiken zeichnen sich durch eine hohe Komplexität aus, die durch die darin beschriebenen Maschinen bedingt wird.

Alle Bauteile der Maschine werden einzeln aufgezeigt und der Zusammenhalt aller einzelnen Teile schematisch dargestellt. (siehe Abbildung 5)

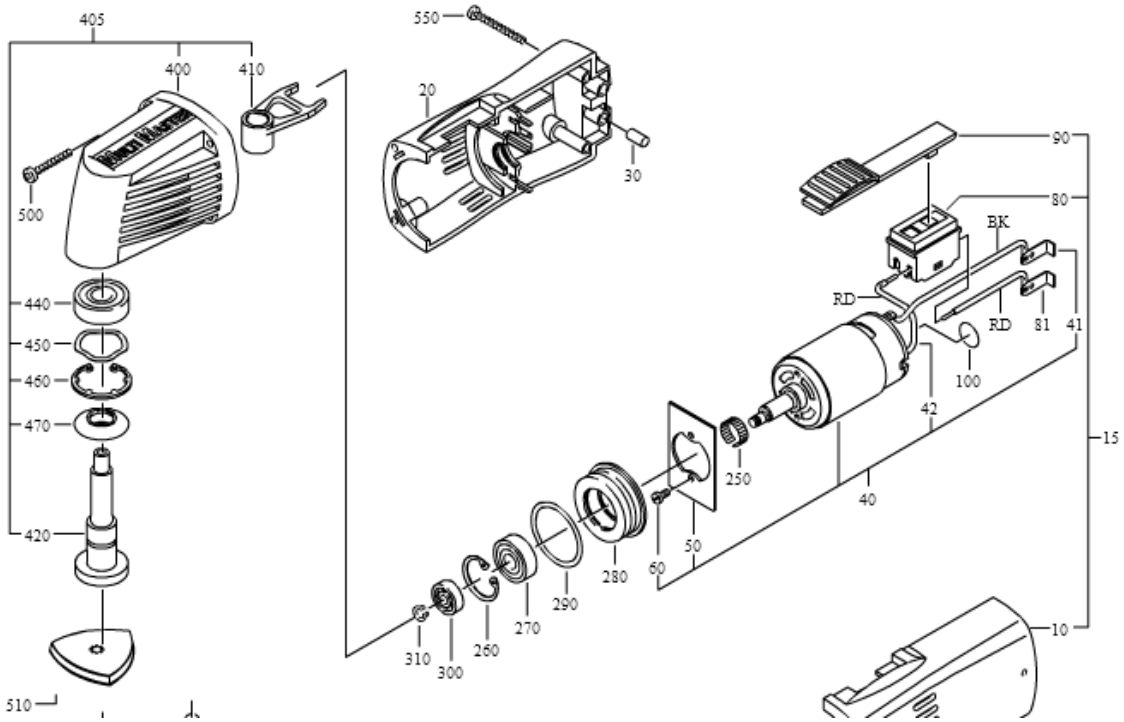


Abbildung 5: Beispiel einer technischen SVG-Grafik für den ETK-Einsatz

4.3 jQuery

Um die Möglicheiten von SVGWeb in Verbindung mit jQuery auszutesten, wird eine Beispiel-Applikation entwickelt, welche im Groben alle wichtigen Aufgaben eines ETKs simuliert. Dazu gehören neben den in Kapitel 2.2 definierten Anforderungen noch folgende:

- Hervorheben von ganzen Bauteilen in der Grafik und gleichzeitiges Hervorheben des zugehörigen Eintrags in einer Liste der auf der Grafik sichtbaren Bauteile und umgekehrt
- Einfaches Navigieren innerhalb der Grafik für die Skalierung (Zoom) und das Bewegen des sichtbaren Ausschnitts (Viewport)
- Springen zu anderen Grafiken durch Links innerhalb der Grafik (z. B.: Durch Klick auf ein Bauteil wird die Grafik dieses Bauteils geladen, in welcher es detalliert angezeigt wird)
- Einbinden von statischen Rastergrafiken

Um die Anforderungen zur Demonstation der Möglichkeiten von SVGWeb in Verbindung mit jQuery gerecht zu werden, werden während der Entwicklung die Entwicklungsvarianten der jeweiligen Bibliotheken benutzt. Diese unterscheiden sich von den Produktivvarianten insofern, als dass in der Entwicklungsvariante keine abkürzenden Methoden benutzt wurden, um die Größe der Quelltextdateien klein zu halten. Diese Methode wird gerade bei JavaScript-Bibliotheken häufig benutzt, da sie das Transfervolumen beim Laden der Datei deutlich reduziert.

Listing 5: Entwicklungsvariante

```
var jQuery = function( selector, context ) {
  return new jQuery.fn.init( selector, context, rootjQuery );
},
_jQuery = window.jQuery,
_$ = window.$,
rootjQuery,
quickExpr = /^(?:[^<]*(<[\w\W]+>)[^>]*$|#([\w\-]+)$)/,
[...]
```

Listing 6: Produktivvariante

```
function(a,b){function cy(a){return f.isWindow(a)?a:a.nodeType===9?a.defaultView
    ||a.parentWindow:!1}function cv(a){if(!cj[a]){var b=f("<"+a+">").appendTo("
    body"),d=b.css("display");b.remove();if(d==="none"[...]
```

4.4 SVGWeb

Das Gerüst der Beispielapplikation ist eine HTML-Datei, in welcher alle Skripte geladen und ausgeführt werden. Ebenfalls wird in dieser HTML-Datei die SVG-Grafik geladen und angezeigt. Dies wird durch die SVGWeb-Bibliothek automatisch durchgeführt.

Listing 7: Einbinden der SVG-Grafik

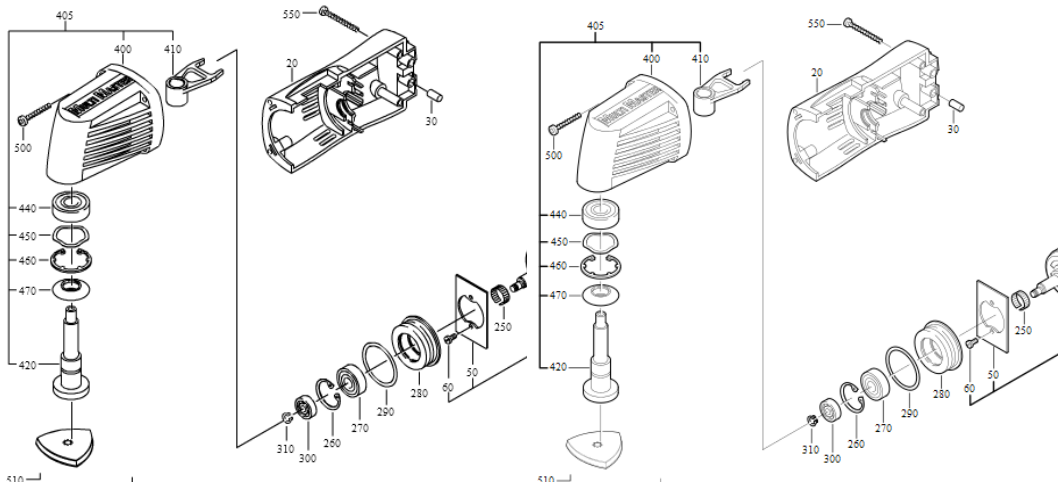
```
<object data="images/test4.svg" width="800" height="600"
  type="image/svg+xml" id="image">
</object>
```

Zur Laufzeit wird das `<object>`-Tag durch ein `<embed>`-Tag ersetzt, welches eine Flash-Repräsentation der SVG-Grafik enthält.

Listing 8: Ersetzter embed-Tag

```
<embed height="600" width="800" class="embedssvg"
  style="visibility: visible;"
  pluginspage="http://www.macromedia.com/go/getflashplayer"
  flashvars="uniqueId=image&amp;sourceType=string&amp;clipMode=neither&amp;debug
      =true&amp;svgId=image"
  type="application/x-shockwave-flash" allowscriptaccess="always"
  swliveconnect="true" name="image" id="image" wmode="transparent"
  quality="high" src="src/svg.swf"/>
```

Die Flash-Repräsentation wird jedoch nur in den Browsern verwendet, die keine native Unterstützung von SVG bieten. Dies erkennt SVGWeb und ist so in der Lage, je nach Browser entweder die SVG-Grafik normal anzeigen zu lassen oder sie durch die Flash-Repräsentation zu ersetzen.



(a) Natives Rendering in Firefox 3.0 (b) Natives Rendering in Firefox 5.0

Abbildung 6: Unterschiede in nativem Rendering von SVG-Grafiken

Auch wenn diese Möglichkeit sehr praktisch ist, wurde für diese Beispielapplikation darauf verzichtet, das SVG nativ anzeigen zu lassen. Auch in Browsern, die eine native Anzeige von SVG erlauben, wird die Flash-Repräsentation geladen. Diese Entscheidung fiel aufgrund der Anforderung, dass das ETK-Environment auf allen Browsern möglichst identisch aussehen soll. Da die nativen Implementierungen sich in der Anzeige der SVG-Grafik unterscheiden (Strichstärken, Größenverhältnisse oder Textfamilien), ist es aufgrund der Konsistenz sinnvoller in jedem Browser die Flash-Repräsentation zu erzwingen. (s. Abbildung 6)

Auch wenn die Unterschiede in diesem Beispiel eher marginal sind, ist es für den Wiedererkennungswert eines ETKs sinnvoller, auf eine exakt gleiche Darstellung in jedem Browser zu bestehen.

Dies wird durch das Setzen des Parameters `svg.render.forceflash` auf `true` im Kopf der HTML-Seite ermöglicht, welche die SVGWeb-Bibliothek dazu veranlasst, immer die Flash-Repräsentation zu laden. (s. Abbildung 7)

Für den Zugriff auf die SVG-Grafik bietet SVGWeb ein virtuelles Interface, welches eine Kommunikation aus JavaScript heraus mit Flash ermöglicht. So werden JavaScript-Befehle an die Flash-Datei gesendet, die dann dort ausgeführt werden. Das gesamte Verhalten ist dem Document Object Model Level 2[10] angepasst, wodurch es als Entwickler keinen Unterschied macht, ob man mit der echten SVG-Grafik arbeitet oder mit der Schnittstelle zu dessen Flash-Repräsentation.

[10] http://www.w3.org/DOM/DOMTR#dom2

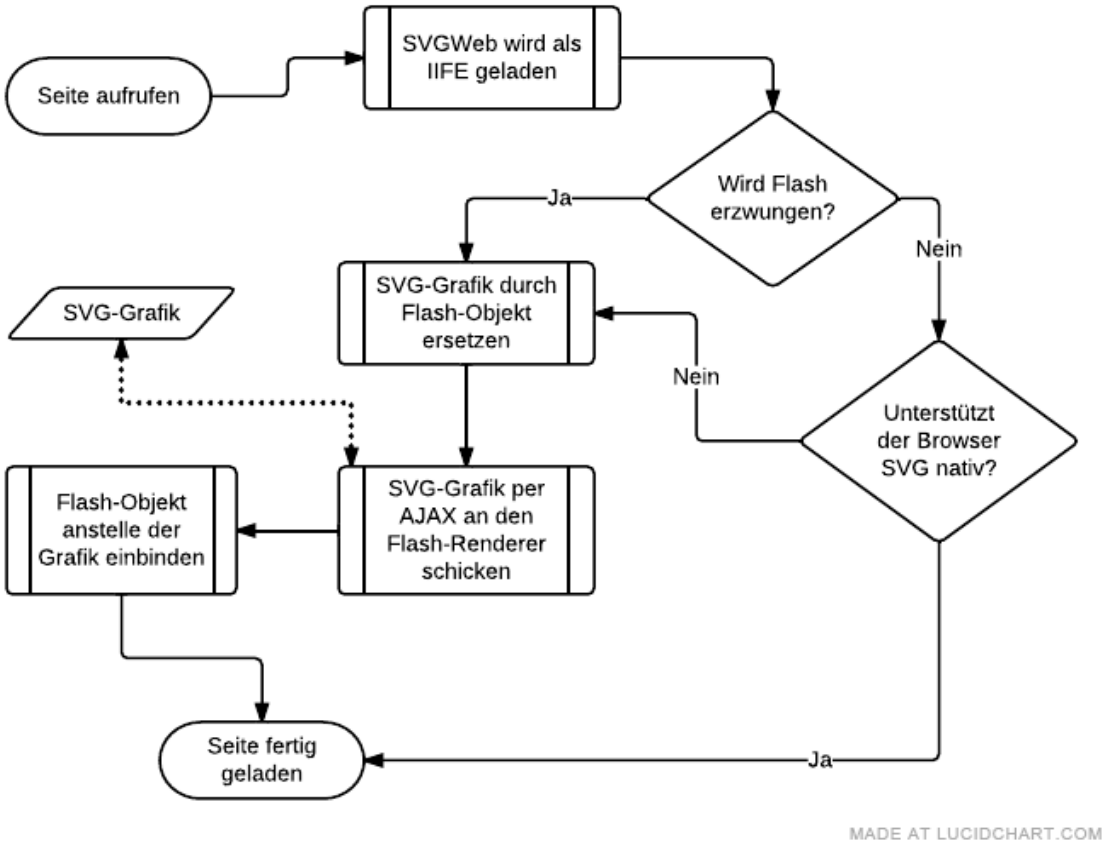


Abbildung 7: Erzwingen des Flashrenderers

Dies ermöglicht es dem Entwickler, alle DOM-Elemente der SVG-Grafik zu erfassen und zu manipulieren. Als Beispiel wird in der folgenden SVG-Grafik über die DOM-Manipulation mittels JavaScript ein Rechteck nach dem Klicken eines Textes in roter Farbe hervorgehoben. Dies ist eine häufig durchgeführte Aufgabe in einem ETK, um Bauteile aus der Gesamtgrafik abzusetzen.

Listing 9: SVG-Beispiel-Grafik

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns:svg="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns="http://www.w3.org/2000/svg" xml:space="preserve" version="1.0"
  width="200" height="200" viewBox="0 0 200 200">
  <g id="group_1">
    <rect width="100" height="100"
      x="50" y="50" id="rect_1" style="fill:#000000;" />
    <text x="55" y="50" id="text_1">Hervorheben</text>
  </g>
</svg>
```

Listing 10: Manipulation der SVG-Beispiel-Grafik

```
1 document.getElementById("text_1").addEventListener("click", function() {
2   var el = document.getElementById("rect1");
3   el.style.fill = "#ff0000";
4 }, "true");
```

Diese relativ einfache Manipulation der SVG-Grafik im Hinblick auf interaktive und dynamische Funktionalität ist das Kernstück der Beispielapplikation.

Auch für die Manipulation der HTML-Seite ist mit der JavaScript-Bibliothek *jQuery* ein mächtiges Werkzeug vorhanden. Wie bereits beispielhaft demonstriert, ermöglicht diese Bibliothek die schnelle und konsistente Entwicklung von dynamischen Inhalten für Webseiten. Gleichzeitig ist es durch den Einsatz von jQuery möglich, die Interoperabilität der Software auf allen gängigen Browserumgebungen ohne zeitraubende Tests und tiefgreifende Entwicklungsarbeit zu gewährleisten[11].

4.5 Umsetzung

4.5.1 Aufbau einer Funktionsbibliothek

Alle Funktionalitäten, die im Rahmen der Beispielapplikation entwickelt wurden, sind in einer so genannten *Immediately Invoked Function Expression* gekapselt. Dieses sehr beliebte Entwicklungsmuster ermöglicht es, Funktionen und Objekte in einem geschlossenen Rahmen zu beschreiben, der von äußeren Einwirkungen unberührt bleibt. So kann man eine JavaScript-Pseudoklasse aufbauen, in der alle Funktionalitäten, die die Bibliothek zum Arbeiten braucht, gekapselt sind.

Listing 11: Einfache JavaScript Immediately Invoked Function Expression

```
1 (function() {
2   var test = "World";
3   console.log("Hello " + test); // Eine Lognachricht mit dem Inhalt "Hello World
      " wird geschrieben
4 })();
5
6 console.log(test); // Dies wird einen Fehler verursachen, da "test" außerhalb
    der Funktion nicht bekannt ist
```

Die so aufgebaute Bibliothek zur Steuerung und Manipulation der SVG-Grafik und der HTML-Umgebung wird mittels einer JavaScript Pseudo-Klasse[12] in das `window`-Objekt eingebunden und ist so im globalen Wertebereich der Applikation verfügbar.

[11] http://docs.jquery.com/Browser_compatibility

[12] Eine solche Pseudo-Klasse ist eigentlich eine Funktion, die Felder enthält, welche wiederum Methoden, Objekte oder Werte enthalten können. Dadurch wird das Verhalten einer Klasse simuliert.

Listing 12: Einbinden der Funktionsbibliothek

```
(function() {
// Deklaration von Variablen...
var config;

// Aufbauen der Pseudo-Klasse
function MyClass() {
  // Methodendeklaration
  var myMethod = function() {
    // Methodenkörper
  };

  // Klassenfelder
  this.field = "value";
}

// Instanziieren der Pseudo-Klasse und Bekanntmachung im globalen Wertebereich
window.MyClass = new MyClass();
})();
```

Durch diese Methode der Deklaration ist die Möglichkeit gegeben, ähnlich wie bei Klassen, private und öffentliche Methoden und Felder zu definieren. So ist z. B. das Feld `field` öffentlich über den Bezeichner `window.MyClass.field` zugänglich. Die Methode `myMethod` ist jedoch nur innerhalb von `MyClass` möglich. Ein Aufruf von `window.MyClass.myMethod()` führt zu einem Fehler.

4.5.2 Hervorheben von Elementen

Um Elemente in der SVG-Grafik hervorzuheben, müssen diese wie oben gezeigt gruppiert werden. So lässt sich das Hervorheben durch die Identifikation der Gruppe über deren `id`-Attribut realisieren. Wenn die Gruppe alle Pfade enthält, die zu einem bestimmten Element (z. B. einer Schraube) gehören, lässt sich so das gesamte Bauteil identifizieren. Um über alle Kindelemente der Gruppe zu iterieren, wird eine rekursive Funktion aufgebaut, die alle Kindelemente hervorhebt.

So ist gewährleistet, dass alle zueinander gehörenden Elemente zuverlässig hervorgehoben werden. Das in der Funktion verwendete Objekt `image.svgObject` ist durch die Konfiguration der Bibliothek mit dem Wurzel-Element der SVG-Grafik belegt worden. Auch `image.highlightNode` und `config.nodes.highlight.color` werden bei der Initialisierung der Funktionsbibliothek gesetzt.

Listing 13: Hervorheben eines Bauteils

```
function highlight(id, element) {
  // Gets the element to highlight
  if (typeof element == "undefined")
    element = image.svgObject.getElementById("p" + id);

  // Create the array field to save the highlighted nodes if it doesn't exist
  if (typeof image.highlightNodes[id] == "undefined")
    image.highlightNodes[id.toLocaleString()] = [];

  // Highlight all nodes and save them to the array of already highlighted nodes
  if (element.nodeName == "line" || element.nodeName == "path") {
    element.style.stroke = config.nodes.highlight.color;
    image.highlightNodes[id.toLocaleString()][image.highlightNodes[id.
        toLocaleString()].length] = element;
  }

  // Processes child nodes, if a group has to be highlighted
  else if (element.nodeName == "g") {
    for (var i = 0; i < element.childNodes.length; i++) {

      // Recursively call highlight using the child nodes, so
      // every level of nested children is highlighted
      this.highlight(id, element.childNodes[i]);
    }
  }
}
```

Durch das Aufrufen von `highlight("group_1");` werden so alle `path`- und `line`-Elemente innerhalb dieser Gruppe mit der konfigurierten Farbe `config.nodes.highlight.color` hervorgehoben.

Wenn Teile hervorgehoben werden können, muss es auch möglich sein, sie wieder in ihren Normalzustand zu versetzen. Zu diesem Zweck werden bereits beim Hervorheben alle Elemente, die durch die `highlight()`-Funktion verändert wurden, in einem Array gespeichert. Beim Aufrufen von `unhighlight()` werden all diese Elemente einzeln wieder in ihren Ursprungszustand versetzt und aus dem Array gelöscht.

Dabei ist das Array so aufgebaut, dass jedes Element als eigenes Feld existiert, in welchem wiederum ein Array gespeichert wird, welches die zugehörigen Elemente enthält. Dadurch ist es auch möglich, gezielt einzelne Bauteile zurückzustellen, ohne dass die Hervorhebung der anderen dadurch beeinflusst wird.

Listing 14: Aufbau des Arrays der hervorgehobenen Elemente

```
var image.highlightNodes = {
  "teil_1": [element1, element2],
  "teil_2": [element3, element4]
};
```

Die Funktion zum Zurücksetzen kann verschiedenste Parameter entgegennehmen. Wenn keiner gegeben ist, werden alle Elemente des gesamten Arrays gelöscht, was durch einen rekursiven Aufruf ermöglicht wird, welchem die Funktion die jeweiligen Unterarrays als Parameter übergibt. Erhält die Funktion ein solches Array, wird über die Inhalte des Arrays iteriert und alle darin enthaltenen Elemente werden gelöscht.

Bei einem *String* oder *Integer* werden nur die Elemente des Arrays gelöscht, die an dieser angegebenen Stelle stehen.

Listing 15: Hervorhebungen entfernen

```
function unhighlight(id) {
  // If no id is specified, all nodes have to be unhighlighted
  if (typeof id == "undefined") {
    for (var name in image.highlightNodes) {

      // Call unhighlight recursively and use the array of elements as parameter
      this.unhighlight(image.highlightNodes[name]);
    }

    // Clear the highlighted nodes
    image.highlightNodes = new Object();

  // If the id is a string or integer, delete all nodes within that property
  // of the highlightNodes object
  } else if (typeof id == "number" || typeof id == "string") {
    var idString = id.toLocaleString();

    // Call unhighlight recursively and use the array of elements as parameter
    this.unhighlight(image.highlightNodes[idString]);

    // Clear the array of the highlighted nodes
    delete image.highlightNodes[idString];

  // If id is an array, unhighlight all its elements
  } else if (typeof id.length != "undefined") {

    for (var i = 0; i < id.length; i++) {
      var del = id[i];

      if (del.nodeName == "line" || del.nodeName == "path")
        del.style.stroke = "#000000";
    }
  }
}
```

Die Funktionen werden als Methoden der in 4.5.1 beschriebenen Pseudo-Klasse gespeichert und sind somit im gesamten Dokument zugänglich. Ebenso können sie einfach auf andere Elemente über *EventListener* gebunden werden, sodass ein Klick auf eine Nummer in der Grafik dazu führt, dass das Bauteil hervorgehoben wird.

Die Funktionen zum Hervorheben und dem Löschen der Hervorhebung werden in einer *Wrapper*-Funktion gebündelt. Diese schaltet die Hervorhebung immer in den momentan nicht aktiven Zustand, d. h. wenn ein Bauteil momentan nicht hervorgehoben wird, sorgt die Funktion für die Ausführung von `highlight()`, andernfalls von `unhighlight()` für das jeweilige Bauteil. Dies ist die tatsächliche *Einstiegsfunktion* von außen. Sie wird aufgerufen, wenn der Benutzer auf einen Text oder ein Listenelement klickt.

Listing 16: Wrapper-Funktion toggleHighlight()

```
this.toggleHighlight = function(num) {
  if (typeof num != "integer")
      num = parseInt(num);

  // Toggle highlight class on list elements
  $(config.list.object).find("a#l" + num).toggleClass(config.list.highlight);

  // If the part is already in the hightlighted nodes, unhiglight it
  if (typeof image.highlightNodes[num.toLocaleString()] != "undefined")
    this.unhighlight(num);

  // Otherwise, highlight it
  else
    this.highlight(num);
};
```

Durch die Einführung dieser Funktion wird eine direkte Verbindung zwischen der SVG-Grafik und der Stückliste hergestellt, die alle in der Grafik vorhandenen Bauteile auflistet. Bei einem Klick auf eines der beiden Elemente werden stets *beide* hervorgehoben oder wieder ausgeglichen. Es spielt dabei keine Rolle, von welchem Element die Aktion des Klickens ausgeführt wird, da beide die gleiche Funktion aufrufen.

4.5.3 Navigation innerhalb der Grafik

Die Vorteile von vektorbasierten Grafiken liegen vor allem in der unbegrenzten Skalierbarkeit und der gleichbleibenden Qualität. Um diese Vorteile auch in der Beispiel-Applikation nutzen zu können, muss eine Navigation eingebaut werden, die es dem Benutzer ermöglicht, sich innerhalb der Grafiken zu „bewegen“. Dazu muss es möglich sein, die Grafik möglichst stufenlos in der Skalierung zu verändern (*Zoom*) sowie den Bildausschnitt zu verschieben (*Bewegung horizontal/vertikal*).

Durch diese grundlegenden Freiheiten kann der Benutzer die gesamte Grafik in beliebiger Detailtiefe betrachten und sich darin zurechtfinden.

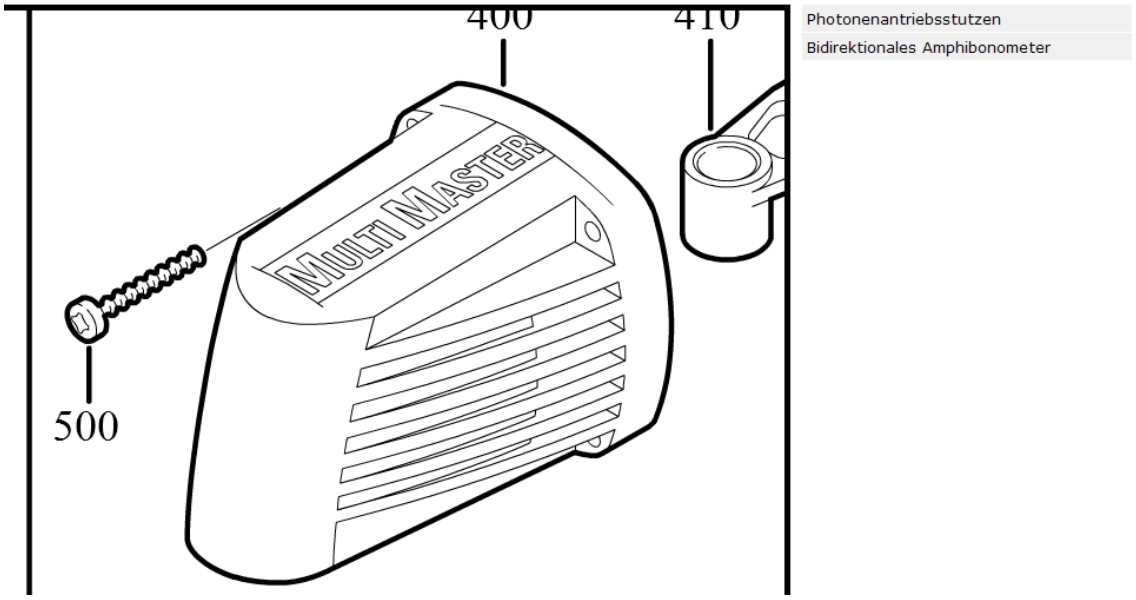


Abbildung 8: Nicht hervorgehobenes Bauteil

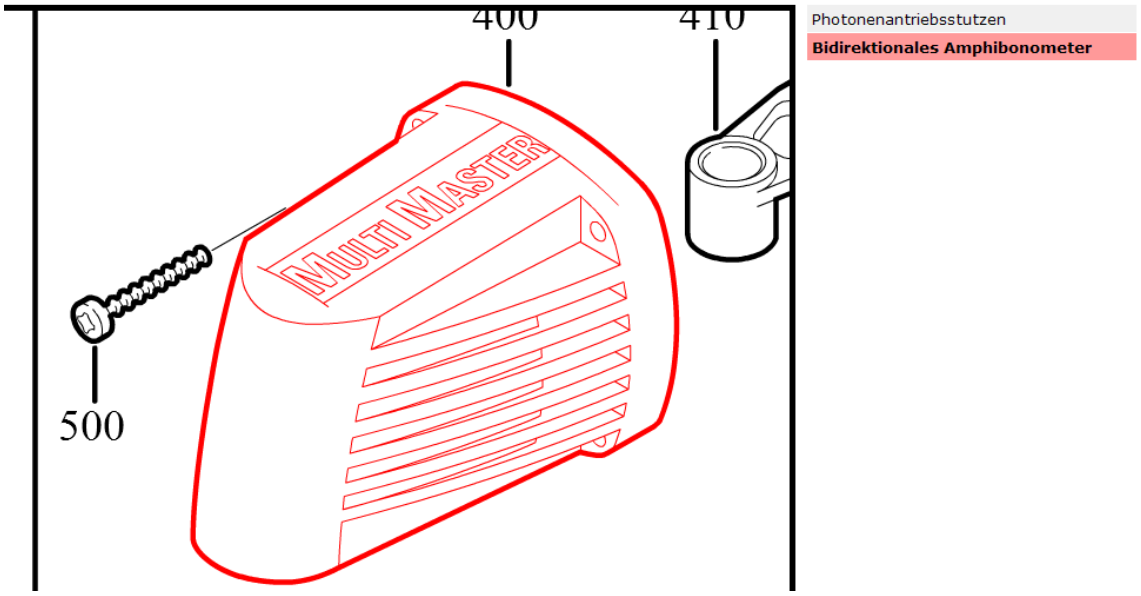


Abbildung 9: Hervorgehobenes Bauteil

Die SVG-Grafik verfügt in ihrem Wurzel-Element über das Attribut `viewBox`, welches den initialen Bildausschnitt definiert, der beim Laden der Grafik angezeigt wird. Dieser kann manipuliert werden, um die Bewegung in horizontaler und vertikaler Richtung zu ermöglichen.

Die Skalierung der Grafik wird über das Attribut `currentScale` gesteuert, welches zu Beginn auf *1* steht.

Im Zuge der Usability für Endanwender muss eine Möglichkeit gefunden werden, durch den Druck auf die Kontrollflächen nicht nur einen einzelnen Schritt (also einen Funktionsdurchlauf) auszulösen, sondern einen kontinuierlichen Aufruf. Hierfür wird eine in JavaScript integrierte Methodik benutzt: die `setTimeout()`-Funktion. Diese Funktion akzeptiert zwei Parameter; eine weitere Funktion und eine Zahl.

Listing 17: Beispielaufruf von setTimeout()

```
var msg = function() {
  alert("Hello world!");
};
setTimeout(msg, 3000); // Nach 3 Sekunden erscheint die Box mit der Nachricht "
    Hello world!"
```

Wie im Beispiel ersichtlich wird, setzt man mit `setTimeout()` eine zeitlich versetzte Aktion. Dies kann genutzt werden, um kontinuierliche Aktionen auszuführen. In der Beispielapplikation wird dies dazu verwendet, um eine Aktion (z. B. *Hineinzoomen*) so lange auszuführen, wie der Benutzer eine Kontrollfläche (Button) gedrückt hält. Sobald die Maustaste losgelassen wird, wird die Timeout-Funktion gelöscht und die Aktion damit abgebrochen.

Beim Klicken (genauer gesagt: wenn die Maustaste über der Fläche des Buttons *herunter* gedrückt wird) des Buttons *zoomIn* wird die Timeout-Funktion genutzt, um die Funktion, welche den Zoom durchführt (die Skalierung der SVG-Grafik vergrößert) innerhalb eines definierten Zeitraums wiederholt auszuführt. Das Event *MouseDown* wird kontinuierlich aufgerufen, solange die Taste gedrückt bleibt. Dadurch wird der Timeout immer wieder erneut gesetzt und verzögert ausgeführt.

Listing 18: zoomIn-EventListener

```
zoomIn: function() {
  image.svgObject.rootElement.currentScale += config.control.zoomFactor;
  control.repeatTimer = setTimeout(control.zoomIn, config.control.repeatTime);
};
[...]
$(".zoomIn").mousedown(function() {
  clearTimeout(control.repeatTimer);
  control.repeatTimer = setTimeout(control.zoomIn, config.control.repeatTime);
}).mouseup(function() {
  clearTimeout(control.repeatTimer);
});
```

Sobald das Event *MouseUp* ausgelöst wird, wird der gespeicherte Timeout wieder gelöscht und die Aktion des Hineinzoomens wird gestoppt. Alle kontinuierlich auszuführenden Aktionen, die für die Navigation in der Grafik zuständig sind, werden auf die gleiche Weise ausgelöst und kontrolliert. Kern jeder Aktion sind die beiden Ereignisse *MouseDown* und *MouseUp*, die wie gezeigt die entsprechenden Timeouts auslösen und wieder löschen.

4.5.4 Verlinken auf andere Grafiken

Für die Interaktion in einem ETK ist es vorteilhaft, wenn man Bauteile direkt mit anderen Baugruppen in Beziehung setzen kann. Z. B. können so in einer Übersichtsgrafik alle Maschinen dargestellt werden und der Benutzer kann dann durch einen Klick darauf zu den jeweiligen Explosionsgrafiken springen. Für solch eine Verlinkung muss es möglich sein, eine SVG-Grafik dynamisch zu laden ohne ihren Pfad im `<object>`-HTML-Tag direkt festzulegen.

Dies ist dank der guten Integration von DOM2 in SVGWeb ohne weiteres möglich. Genutzt wird hierfür ähnlich wie beim *Hervorheben von Bauteilen* ein *EventListener* für das Klicken eines Textes. Sobald dieses Ereignis ausgelöst wird, kann über JavaScript der URL der Seite geändert werden und mit einem beliebigen Übergabeparameter neu geladen werden.

Dieser Übergabeparameter kann *vor* dem Laden der SVG-Grafik abgefangen werden und dazu benutzt werden, die Adresse der geladenen Grafik zu ändern. Wichtig dabei ist, dass es vor der Fertigstellung des Ladens der alten Grafik geschieht, sodass der durch die Fertigstellung des Ladens ausgelöste *EventListener* für die korrekte Grafik ausgelöst wird (nämlich die neue).

Um die Übergabeparameter auszulesen, wird eine Funktion definiert, die diese als Array gespeichert zurückgibt. Dieses Array enthält alle Parameter und die dazugehörigen Werte, welche in dem URL übergeben wurden.

Listing 19: Auslesen der GET-Parameter eines URLs

```
function getUrlParams(url) {
  if (typeof url == "null" || typeof url == "undefined")
    url = window.location.href;
  else if (typeof url != "string")
    return null;

  var urlVars = {};

  url.replace(/[?&]+([^=&]+)=([^&]*)/gi, function(m, key, value){
    urlVars[key] = value;
  });

  return urlVars;
}
// getUrlParams("http://test.de/?hallo=welt"); gibt zurück: array["hallo"] = "
    welt";
```

Mittels dieser Funktion kann der URL auf den Parameter `image` hin überprüft werden. Dieser wird benutzt, um die Adresse der zu ladenden SVG-Grafik zu bearbeiten. Dies geschieht wie erwähnt vor dem Setzen eines *EventListeners* auf das Ereignis *SVG-Grafik geladen*.

Listing 20: Anpassen der Adresse der SVG-Grafik anhand des Parameters

```
var params = getUrlParams();
if (typeof params.image != "")
  $("#image").attr("data", params.image);
```

Dies ermöglicht das Austauschen einer Grafik zur Laufzeit. Da jedoch gleichzeitig auch noch die Liste der in der Grafik enthaltenen Bauteile angepasst werden muss, ist es sinnvoller, für jede Grafik eine eigene HTML-Seite generieren zu lassen. Das ist möglich, da der ETK stets durch einen automatisierten Generierungsprozess erstellt wird, bei dem alle Grafiken, Texte und Seiten aus einer Datenbank ausgelesen und zu einer HTML-Struktur zusammengesetzt werden.

Während dieser Zusammensetzung muss der Generator also die Liste aller Bauteile sowie die Adresse der SVG-Grafik in den Kopf der HTML-Datei schreiben, sodass die Bibliothek mit diesen Informationen geladen werden kann.

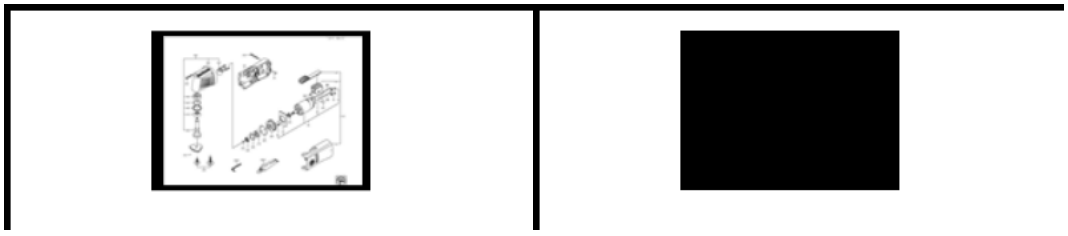
4.5.5 Einbinden von Rastergrafiken

Das Einbinden von Rastergrafiken ist im SVG-Standard des W3C definiert [Wor03, Abschnitt 5.7] und für den Einsatz in ETKs sinnvoll. SVGWeb unterstützt diese Funktionalität jedoch nicht. Der Flash-Renderer kann die Rastergrafiken nicht einbetten. Die native Anzeige des *Firefox* hingegen unterstützt die Funktion. Im folgenden Beispiel wurde eine Rasterversion der in 4.2 gezeigten SVG-Grafik in eine SVG-Grafik eingebettet.

Listing 21: Einbetten einer Rastergrafik

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns:svg="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns="http://www.w3.org/2000/svg" xml:space="preserve" version="1.0"
  width="200" height="200" viewBox="0 0 200 200">
  <rect x="2" y="5" width="55" height="40" />
  <image x="5" y="5" width="50" height="40"
    xlink:href="thumbnail.png">
    <title>Thumbnail</title>
  </image>
</svg>
```

Das Ergebnis im nativen und im Flash-Modus zeigt, dass die Anzeige von Rastergrafiken mit der Verwendung des Flash-Renderers nicht möglich ist.



(a) Natives Rendering (b) Flash-Rendering

Abbildung 10: Vergleich der Renderer mit eingebetteten Rastergrafiken

Auf eine Anfrage an den Hersteller von SVGWeb bezüglich dieser Funktionalität kam keine Antwort, wodurch die Verlässlichkeit dieser Funktion als unzureichend eingestuft werden muss.

Die einzige Möglichkeit ist eine Eigenentwicklung, die durch die Quelloffenheit von SVGWeb grundsätzlich möglich ist.

4.5.6 Usability

Die Usability[13] ist ein wesentlicher Punkt in der Entwicklung der Beispielapplikation. Damit der Benutzer sich in der Grafik stets zurecht findet, wurde eine Miniaturansicht eingebunden, die dem Benutzer jederzeit darstellen soll, wie die Skalierung des Bildes und seine momentane Position ist.

Um dies zu gewährleisten, muss ebenfalls der bereist erwähnte Generierungsprozess (siehe 4.5.4 Ende) angepasst werden. Von der erzeugten SVG-Grafik muss ein Abbild erzeugt werden, welches mit ungefähr 20-facher Verkleinerung gespeichert (als Rastergrafik, z. B. `.png`) wird. Diese Rastergrafik kann dann in der HTML-Struktur des ETK eingebunden werden und eine Miniaturdarstellung der Originalgrafik darstellen.

Die Miniaturgrafik wird über die Originalgrafik gelegt, um einen direkt sichtbaren Bezug zwischen den beiden Grafiken herzustellen.

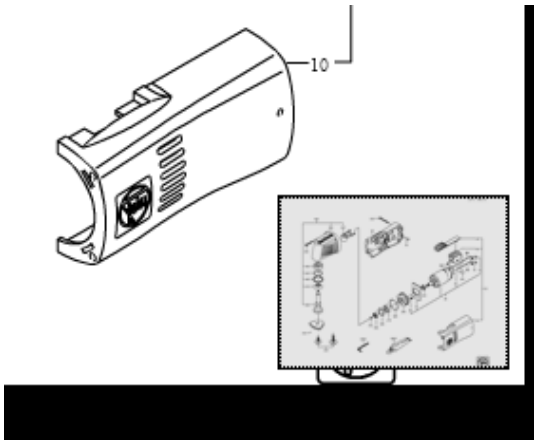


Abbildung 11: Miniaturansicht der Originalgrafik

Um zusätzlich die Navigation innerhalb der Grafik zu vereinfachen, werden alle Kontrollfunktionalitäten (Zoom, Bewegung) nicht nur auf die Originalgrafik ausgeführt, sondern auch auf die Miniatur. Dabei repräsentiert eine halbdurchsichtige Box, welche über der Miniaturgrafik liegt, den aktuellen Bildausschnitt auf der Originalgrafik. Dieser wird genau wie der tatsächliche Bildausschnitt mit jeder Aktion des Benutzers in Position oder Größe verändert.

Damit jede Bewegung und jedes Verändern der Skalierung gleichzeitig auf die Miniaturansicht anwendbar ist, muss das Verhältnis der Größen der beiden Grafiken berechnet werden. Nur so kann eine Bewegung um beispielsweise 100 Pixel im Originalbild in eine Bewegung um 5 Pixel auf der Miniaturansicht *übersetzt* werden.

[13] Dies bezeichnet die Einfachheit und Intuitivität der Bedienung der Benutzeroberfläche.

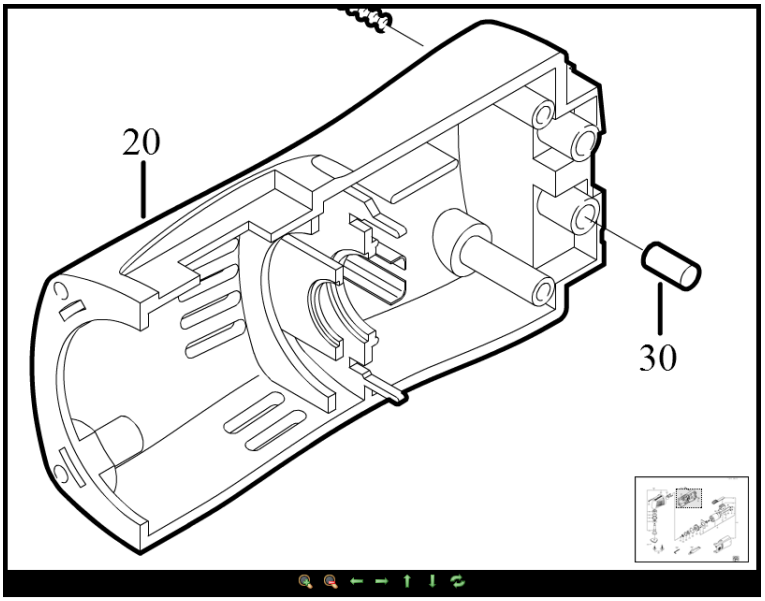


Abbildung 12: Bildausschnitt in der Originalgrafik und in der Miniaturansicht

Da die Höhen/Breiten-Verhältnisse der Originalgrafiken und damit auch der um einen Faktor kleineren Miniaturgrafiken oft unterschiedlich sind, kann keine einfache Verkleinerung der Werte um z. B. $\frac{1}{20}$ erfolgen.

Die Verhältnisse müssen für jedes Bild dynamisch berechnet werden, um immer eine exakte Darstellung des aktuellen Bildausschnittes zu gewährleisten. Dafür wird beim Laden der Funktionsbibliothek die Größe der Originalgrafik und die der Miniaturgrafik verglichen und deren Verhältnis zur Umrechnung der Bewegungen genutzt. Dieses Verhältnis wird mit der Bewegung multipliziert, um die entsprechende Bewegung in der Miniaturansicht darzustellen.

Listing 22: Umrechnen einer Bewegung auf die Miniaturgrafik

```
1 moveRight: function() {
2   currentX = image.svgObject.rootElement.currentTranslate.x;
3   image.svgObject.rootElement.currentTranslate.setX(currentX - config.control.
        moveFactor);
4   control.repeatTimer = setTimeout(control.moveRight, config.control.repeatTime)
        ;
5
6   var left = parseInt(control.mover.object.css("left"));
7   control.mover.left += config.control.moveFactor * control.mover.width / image.
        viewPort.width;
8   control.mover.object.css("left", parseInt(control.mover.left));
9 }
```

Für die korrekte Darstellung der Skalierung in der Miniaturgrafik muss die Skalierung der Originalgrafik berücksichtigt werden. Diese wird verwendet, um die Größe des Bildausschnittes in der Miniaturgrafik zu bestimmen. Dabei ist zu beachten, dass der Bildausschnitt der Miniaturgrafik prinzipiell nicht kleiner als 1 Pixel und nicht größer als die Gesamtgröße des Miniaturbildes werden kann.

Listing 23: Umrechnen der Skalierung auf die Miniaturgrafik

```
1  zoomIn: function() {
2    image.svgObject.rootElement.currentScale += config.control.zoomFactor;
3    control.repeatTimer = setTimeout(control.zoomIn, config.control.repeatTime);
4
5    var width = parseInt(control.mover.object.css("width"));
6    var height = parseInt(control.mover.object.css("height"));
7
8    control.mover.object.css({
9      "width": width == 1 ? 1 : control.mover.width / image.svgObject.rootElement.
           currentScale,
10     "height": height == 1 ? 1 : control.mover.height / image.svgObject.
           rootElement.currentScale
11   });
12 }
```

Zusätzlich zu der Kopplung der Bewegung und Skalierung der Originalgrafik kann mit dem Einführen der Miniaturgrafik noch ein weiteres Feature eingeführt werden: Das direkte Bewegen des Bildausschnittes durch Bewegung des Bildausschnittes der Miniaturgrafik. Dieser wird durch die Verwendung der jQuery-UI-Bibliothek[14] zu einem bewegbaren Objekt, welches durch Klicken-und-Ziehen der Maus innerhalb der Miniaturgrafik bewegt werden kann.

Die Bewegung dieses Bildausschnittes wird, ähnlich wie bei der Umrechnung der Bewegung der Originalgrafik auf die Bewegung in der Miniaturgrafik, auf eine Bewegung in der Originalgrafik umgerechnet und dort direkt angezeigt.

Um die Bewegung des Bildausschnittes dem Benutzer einsichtig zu machen, wird der Bildausschnitt mit einem besonderen Cursor versehen, der die Bewegbarkeit anzeigt.

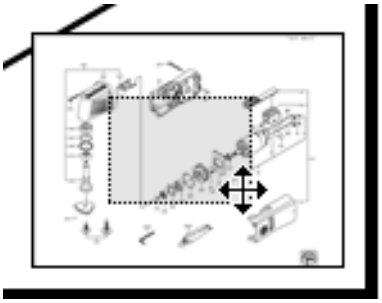


Abbildung 13: Bewegbarer Bildausschnitt in der Miniaturgrafik

[14] http://ui.jquery.com

Listing 24: Bewegung des Bildausschnittes der Miniaturgrafik

```
// Binds a dragging-function on the mover. When it is dragged along its parent,
// the movement is translated to the image.
control.mover.object.draggable({
  containment: "parent",
  drag: function() {
    diffX = parseInt($(this).css("left"));
    diffY =  parseInt($(this).css("top"));

    // Calculate the movement for the large image by
    // multiplying it with the size ratio between the viewport and the thumbnail
    setX = diffX * image.viewPort.width / control.mover.width;
    setY = diffY * image.viewPort.height / control.mover.height;

    // Invert both coordinates for correct display in large image
    image.svgObject.rootElement.currentTranslate.setXY(-1 * setX, -1 * setY);

    // Apply the moved coordinates to the virtual mover position
    control.mover.left = diffX;
    control.mover.top = diffY;
  }
});
```

4.6 Anpassung und Zukunftsfähigkeit

Die entwickelte Beispielapplikation muss für den produktiven Einsatz angepasst werden. Diese Anpassungen werden voraussichtlich nur die HTML-Struktur betreffen, da durch die Konzeption der Funktionsbibliothek als JavaScript-Pseudo-Klasse die Kernfunktionalitäten bereits umfassend und von außen abgekapselt implementiert sind. Dadurch ist das System, welches bei der Entwicklung der Beispielapplikation entstanden ist, einfach auf andere ETKs portierbar.

Dabei muss jedoch beachtet werden, dass die SVG-Grafiken, die für die Darstellung verwendet werden, den Richtlinien genügen, um alle Funktionalitäten des Systems ausnutzen zu können (s. 4.2). Dies betrifft vor allem die Gruppierung der Elemente zu Bauteilen, um das Hervorheben zu ermöglichen. Hier ist ebenfalls wichtig, dass alle Bauteile mit eindeutigen Nummern versehen werden.

Um die Funktionsbibliothek nutzen zu können, sind folgende Voraussetzungen einzuhalten:

SVG-Standard-Konformität Der Flash-Renderer verschluckt Fehlermeldungen, die beim Lesen der SVG-Grafik erzeugt werden. Daher ist es wichtig, dass die SVG-Grafiken von vorneherein auf ihre Konformität zum Standard geprüft und angepasst werden. Fehlerhafte SVG-Grafiken werden nicht angezeigt.

Gruppierung der Bauteile SVG-Elemente wie `line` oder `path`, welche ein Bauteil darstellen, müssen in einer Gruppe (SVG-Element `g`) zusammengefasst werden, welche alle Element enthält, die ein Bauteil darstellen. Die ID des Bauteiles (s. u.) muss auf diese Gruppe gesetzt werden.

Benennung der Bauteile In der SVG-Grafik müssen alle Bauteile sowie deren Nummern mit eindeutigen IDs versehen werden, damit ihre Zuordnung automatisch generiert werden kann. Dabei müssen die Text-Elemente, welche die Bauteilnummer enthalten, mit dem Präfix `t` und die Gruppen der Bauteile mit dem Präfix `p` versehen werden. Nach den Präfixen muss eine eindeutige Nummer folgen. Diese wird als Bauteilnummer zur Identifikation verwendet.

Erzeugung einer Miniaturgrafik Für die Anzeige muss eine Miniaturgrafik vorliegen, um die Navigation zu ermöglichen. Diese kann bereits im Generierungsprozess erstellt werden, indem z. B. die SVG-Grafik intern geladen, verkleinert und als Rastergrafik neben der Originalgrafik gespeichert wird. Wichtig ist, dass die Maße der Miniaturgrafik zwischen 50x50 und 150x150 Pixel liegen, sodass die Informationen der Originalgrafik nicht im Detail verloren gehen. Alternativ ist dieser Schritt auch manuell möglich.

Die individuelle Anpassung der Funktionsbibliothek wird über die Konfiguration derselbigen ermöglicht. In dieser kann angepasst werden, mit welcher Farbe die Bauteile im Bild hervorgehoben werden, welche CSS-Klasse die hervorgehobenen Bauteile in der Liste erhalten und welches `ul`-Element diese Liste enthalten soll. Weiterhin lassen sich verschiedene Einstellungen zur Navigation innerhalb des Bildes anpassen, z. B. die Geschwindigkeit des Scrollens und Zoomens.

Weiterhin müssen der Funktionsbibliothek alle Objekte übergeben werden, die zur Anzeige und Manipulation des Bildes benötigt werden. Dazu gehören:

- Das `<object>`-Element, welches die SVG-Grafik enthält.
- Das `<div>`-Element, welches die Miniaturgrafik enthalten soll.
- Das `<div>`-Element, welches den aktuellen Bildausschnitt enthalten soll.

Diese Objekte können entweder als native JavaScript-Objekte oder als jQuery-Objekte übergeben werden. Die Konfiguration sollte dem EventListener hinzugefügt werden, welcher beim Laden der SVG-Grafik ausgelöst wird, um sicherzugehen, dass die Funktionsbibliothek vollen Zugriff auf den erweiterten DOM-Baum der Grafik hat.

Eine Initialisierung der Funktionsbibliothek ist nicht nötig, da sie bereits durch das Einbinden der JavaScript-Datei geladen und öffentlich zugänglich gemacht wird.

Listing 25: Konfiguration der Funktionsbibliothek

```
$(document).ready(function() {
  window.addEventListener('SVGLoad', function() {
    tetkenv.init({
      list: $("#list"),
      image: $("#image"),
      imageSrc: imageSrc,
      thumbnail: $("#thumbnail"),
      mover: $("#mover")
    });
  });
});
```

Wenn Änderungen am Quellcode anstehen, können diese durch die Kapselung der Funktionsbibliothek problemlos durchgeführt werden, da an den Schnittstellen zur Anwendungsschicht nichts geändert werden muss. Dies macht Updates am Kern der Applikation sowie SVGWeb selbst einfacher und sichert so die Zukunftsfähigkeit der Bibliotheken.

Um die Kapselung der Daten noch einen Schritt weiter zu treiben, werden die Bauteile nicht direkt beim Laden des Bildes mit an die Funktionsbibliothek übergeben. Dies geschieht erst nach Laden des Bildes durch eine synchrone Anfrage an ein unabhängig bestehendes Skript. In diesem Skript wird mittels des Bildnamens ermittelt, welche Bauteile in dieser Grafik angezeigt werden.

Listing 26: Anfrage an Datenskript

```
$.ajax({
  url: "dataServer.php",
  data: { "data": "image_parts", "image_parts": loadConfig.imageSrc },
  success: function(data) {
    setParts(data);
  },
  error: function() {
    logError("Parts of image could not be loaded");
  },
  async : false
});
```

Zu diesen Grafiken werden alle relevanten Daten geladen und als JSON-Objekt (JavaScript Object Notation) zurückgegeben. Für die Entwicklung wurde das Skript jedoch soweit angepasst, dass stehts nur Blinddaten zurückgegeben werden.

Der Vorteil dieser Methodik liegt auf der Hand: Das Skript, welches die Daten der Bauteile liefert, kann unabhängig von der Funktionsbibliothek angepasst werden. Ebenso spielt es keine Rolle, wo und auf welcher Sprache dieses Skript läuft. So kann dieses Skript in einer beliebigen Sprache programmiert werden - wichtig ist nur, dass das zurückgelieferte JSON-Objekt der nachfolgend definierten Schnittstelle genügt.

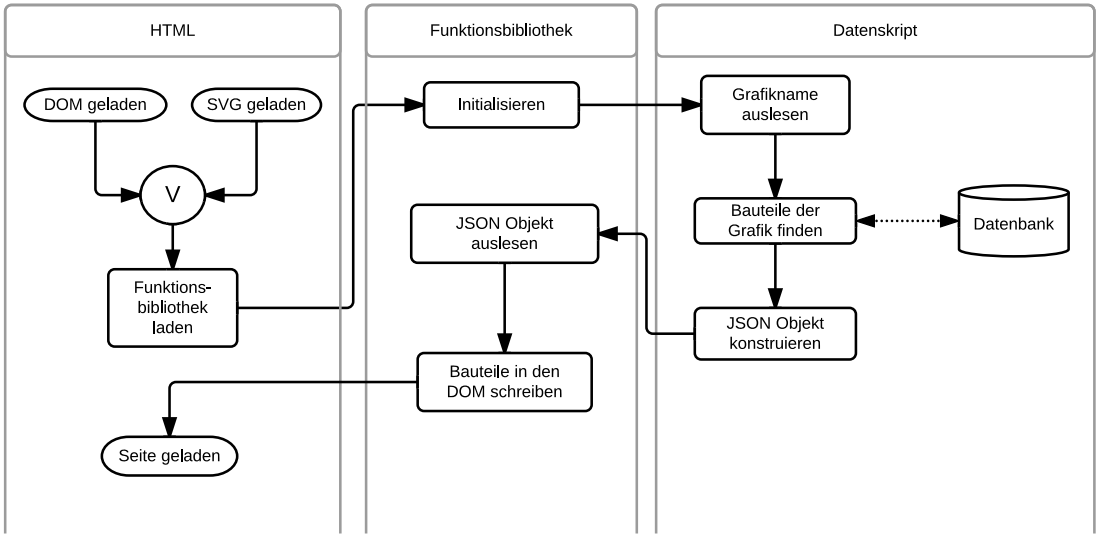


Abbildung 14: Laden der Bauteildaten durch ein unabhängiges Skript

Listing 27: Schnittstelle JSON-Bauteildaten

```
REQUEST-HEADER
==============
  Method: HTTP/1.1 GET  # Das Skript wird per HTTP-GET aufgerufen
  Parameter:  # Die Parameter, welchem dem Skript im HTTP-Header mitgegeben
      werden
    data=image_parts
    image_parts=GRAFIKNAME  # GRAFIKNAME wird durch den namen der Grafik ersetzt
       , welche geladen wird (z. B. "image_1.svg")

RESPONSE-HEADER
==============
  Content-Type: application/json  # nach RFC 4627

RESPONSE-BODY
==============
[ # Beginn des Arrays, welche alle Bauteile enthält
  { # Beginn eines Bauteil-Objekts
    "num": "123", # Die Nummer des Bauteils
    "name": "Bauteil",  # Der Name des Bauteils
    "description": "Beschreibung",  # Beschreibung des Bauteils
    "size": "20 cm",  # Größe des Bauteils
    "prize": "1.234", # Preis des Bauteils
    "detailed": "image-detail.svg", # Ein URI zu einer detailreicheren
       Darstellung dieses Bauteils
    "requirements": "Anderes Bauteil" # Ein anderes Bauteil, das benötigt wird,
       um dieses zu verwenden
  }, {...}  # Weitere Bauteilelemente durch Kommata getrennt
]
```

Diese Schnittstelle kann von jedem webfähigen Skript generiert werden. Was genau bei der Abfrage der Bauteile abläuft, ist unabhängig von der Funktionsbibliothek, die in die HTML-Seite eingebunden wird. Im Kundensystem wird dies durch eine SQL-Abfrage an eine Datenbank geschehen, aus welchem die Bauteile für jede Grafik hervorgehen.

5 Generator

5.1 Allgemeines

Der gesamte ETK wird durch einen so genannten *Generator-Prozess* erstellt. Dieser Prozess erstellt aus Stücklisten und weiteren Eingabeparameter den kompletten Katalog sowohl als Programm zur lokalen Ausführung als auch in einem HTML-Format für die Darstellung in Browsern. Dieser Generator ist enorm umfangreich und muss für den durch die entwickelte Funktionsbibliothek veränderten Aufbau der HTML-Seiten angepasst werden.

Nur mit dieser Anpassung kann der ETK in vollem Umfang und mit allen Funktionen der entwickelten Beispielapplikation genutzt werden. Dazu erfolgt zunächst eine Analyse des momentanen Prozesses, um die Stellen für die Anpassung zu finden.

5.2 Beschreibung

Bevor der Generator eingesetzt werden kann, müssen die Grafiken erzeugt werden. Dies geschieht durch eine externe Firma, welche aus den CGM-Grafiken EPS-Grafiken herstellt. Diese werden durch einen Mitarbeiter mit sog. *Hotspots* versehen. Diese definieren innerhalb der Grafik Gruppen, welche später mit einem Skript erkannt und manipuliert werden können.

Dem Generator selbst liegen viele verschiedene Eingeabedateien zu Grunde, aus denen der fertige Katalog gebildet wird. Diese Grunddaten werden von einem Kunden bereitgestellt, der den Katalog erzeugen möchte. Um den bisherigen Katalog zu *bauen*, sind folgende Dateien benötigt:

bauteile.csv Alle Bauteile, deren Beziehungen zueinander und deren Daten werden in dieser Datei zusammengefasst. Sie bildet die Basis der Kategorien im Katalog, sowie aller Maschinen, die bildlich dargestellt werden. Die Namen der Bauteile korrespondieren mit den zugehörigen Grafiken, die nachfolgend aufgezählt werden.

bauteil1.eps Die SVG-Grafik, welche später für die Darstellung eines bestimmten Bauteils im Browser verwendet wird.

bauteil1.jpg Eine Realdarstellung des Bauteils oder der Maschine, die in der zugehörigen **.eps* dargestellt ist.

preise.csv Eine tabellarische Auflistung der Preise für Bauteile oder Maschinen. Diese wird seperat von *bauteile.csv* gehalten, da sich die Preise ständig ändern könnten. Da diese Datei später als Datenbank für den Katalog dient, aus dem die Preise extrahiert werden, lassen sich so auch in einem laufenden Katalog ohne komplette Neugenerierung die Preise ändern.

Der Prozess des Kunden gliedert sich dabei in drei Aufgaben, die im Endeffekt zur Erstellung des Katalogs führen.

Schritt 1 Erzeugung der **.svgz*-Dateien. Dieser Schritt erzeugt aus den hinterlegten **.eps*-Grafiken komprimierte SVG-Grafiken. Diese sind mit einem ZIP-Verfahren[15] komprimiert. Über eine XSLT-Datei werden die im EPS hinterlegten Hotspot-Daten in die SVG-Datei importiert.

Schritt 2 Erzeugung des Gesamtkataloges. Dieser Schritt wird durch eine kompilierte Java-Applikation durchgeführt und greift alle oben aufgelisteten Dateien auf, verarbeitet diese, und erzeugt in einem definierten Ausgabeordner den fertigen ETK. Dabei lässt sich zwischen einer Online- und einer Offlineversion unterscheiden. Die Offline-Version wird so kompiliert, dass sie von einer CD gestartet werden kann. Für die Online-Version werden die entsprechenden HTML-, JavaScript- und CSS-Dateien erzeugt.

Schritt 3 Erzeugung der Preisdatenbank. Die `preise.csv`-Datei wird verwendet, um eine Datenbank zu erzeugen, in welcher die Preise aller Bauteile und Maschinen gespeichert sind. Diese wird vom ETK dynamisch eingelesen, was eine spätere Ersetzung auch im laufenden System ermöglicht. Die erzeugte Datei muss manuell in den Unterordner des ETKs verschoben und ersetzt werden, damit die Preisänderungen aktiv werden.

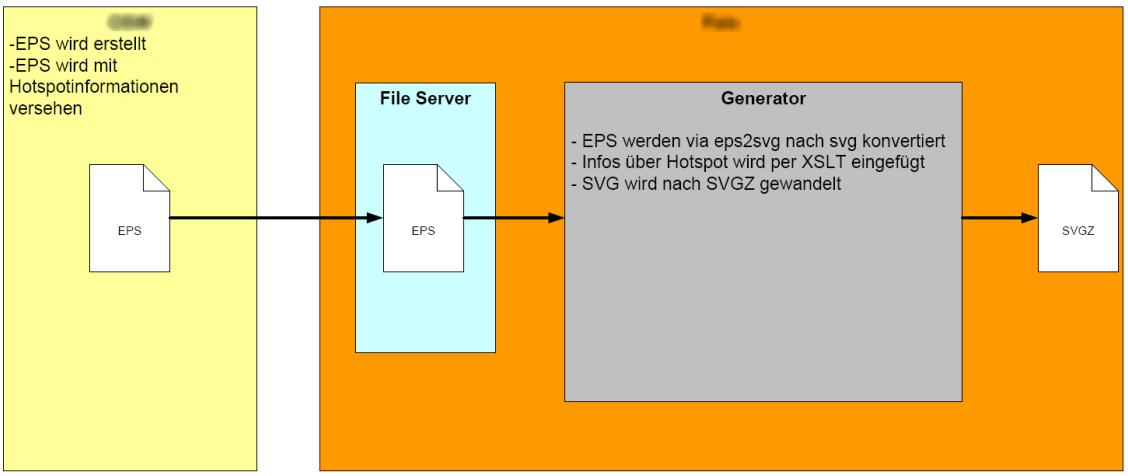


Abbildung 15: Prozess der Grafikerstellung für den Einsatz im ETK

Die Gesamtstruktur des Generators und den damit verbundenen Komponenten umfasst insgesamt vier verschiedene, voneinander unabhängige Tomcat-Server-Systeme. Auf diesen System werden die erstellten Kataloge publiziert. Sie sind dann für jeweils unterschiedliche Nutzergruppen zugänglich.

Tomcat 1 - Internet Dieser Server kann von jedem über das Internet erreicht werden. Er wird auf der Kunden-Homepage als Produktkatalog verlinkt.

[15] Da **.zip* nur eine generische Endung ist, ist nicht genau definiert, welcher Algorithmus verwendet wird. Die SVG-Interpreter können die meisten Algorithmen jedoch problemlos entschlüsseln.

Tomcat 2 - Extranet Der Extranet-Server ist identisch mit dem Internet-Server, mit dem Unterschied, dass er nur von bestimmten, freigeschalteten Benutzern erreicht werden kann. Hierfür wird eine zweite Datenbank benötigt, in welche die zugelassenen User samt Passwort gespeichert sind. Ebenfalls besitzt dieser Server eine eigene Preisdatenbank, da sich diese von den Preisen für die Benutzer, welche den *Internet*-Server verwenden, unterscheiden.

Tomcat 3 - Test Ein Testserver, auf dem neue Funktionen des Katalogs getestet werden können. Er greift auf die gleichen Datenbanken wie die beiden oberen Server zu, besitzt aber keine eigene Preisdatenbank.

Tomcat 4 - Preview Ein Vorschauserver, der lokal beim Kunden installiert wird. Wenn ein neuer Katalog erzeugt wird, muss er zunächst auf diesem Vorschauserver geladen werden, um eine Qualitätsprüfung durchzuführen. Wenn die Prüfung ergibt, dass der Katalog einsatzbereit ist, wird er per FTP auf die Systeme *Tomact 1 - 3* übertragen. Dieser Server besitzt eine getrennte Datenbank.

Das Zusammespiel der einzelnen Server, Datenbanken und des Generators wird in Abb. 16 dargestellt.

Eine Besonderheit ist das Testsystem. Dieses erlaubt den Wechsel zwischen den beiden Preis-Datenbanken des Internet- und Extranet-Systems. So können die einzelnen Datenbanken leichter getestet werden. Das Testsystem ist nur über einen direkten, geheimen Link erreichbar und anderweitig von Außen nicht sichtbar.

5.3 Anpassungen und Verbesserungen

Bei der Analyse der Prozesse, die bei der Generierung eines Katalogs durchlaufen werden, sind mehrere problematische Schnittstellen aufgetreten, die durch eine Anpassung aller beteiligten Prozesse behoben werden sollen.

So wurden bisher z. B. von einem Grafikdienstleister **.eps* an den Kunden geliefert. Diese wurden wie oben beschrieben durch den Generator in **.sgv* bzw. **.svgz*-Dateien umgewandelt. Bei der Analyse der Geschäftsprozesse kam heraus, dass der Grafikdienstleister intern bereits mit SVG-Grafiken arbeitet. Für die Lieferung an den Kunden wurden diese dann in EPS-Grafiken überführt.

Dieser überflüssige Schritt soll mit dem neuen Generator ebenfalls entfernt werden. So wird der Grafikdienstleister von vornerein SVG-Grafiken an den Kunden liefern. Der Prozessschritt des Umwandelns von EPS zu SVG(Z) entfällt damit für den neuen Generator.

So können die vom Grafikdienstleister[16] bereitgestellten Grafiken direkt in den fertigen ETK übernommen werden. Die spart Zeit und verringert die Fehleranfälligkeit des Gesamsystems erheblich.

[16] In Abb. 17 blau dargestellt

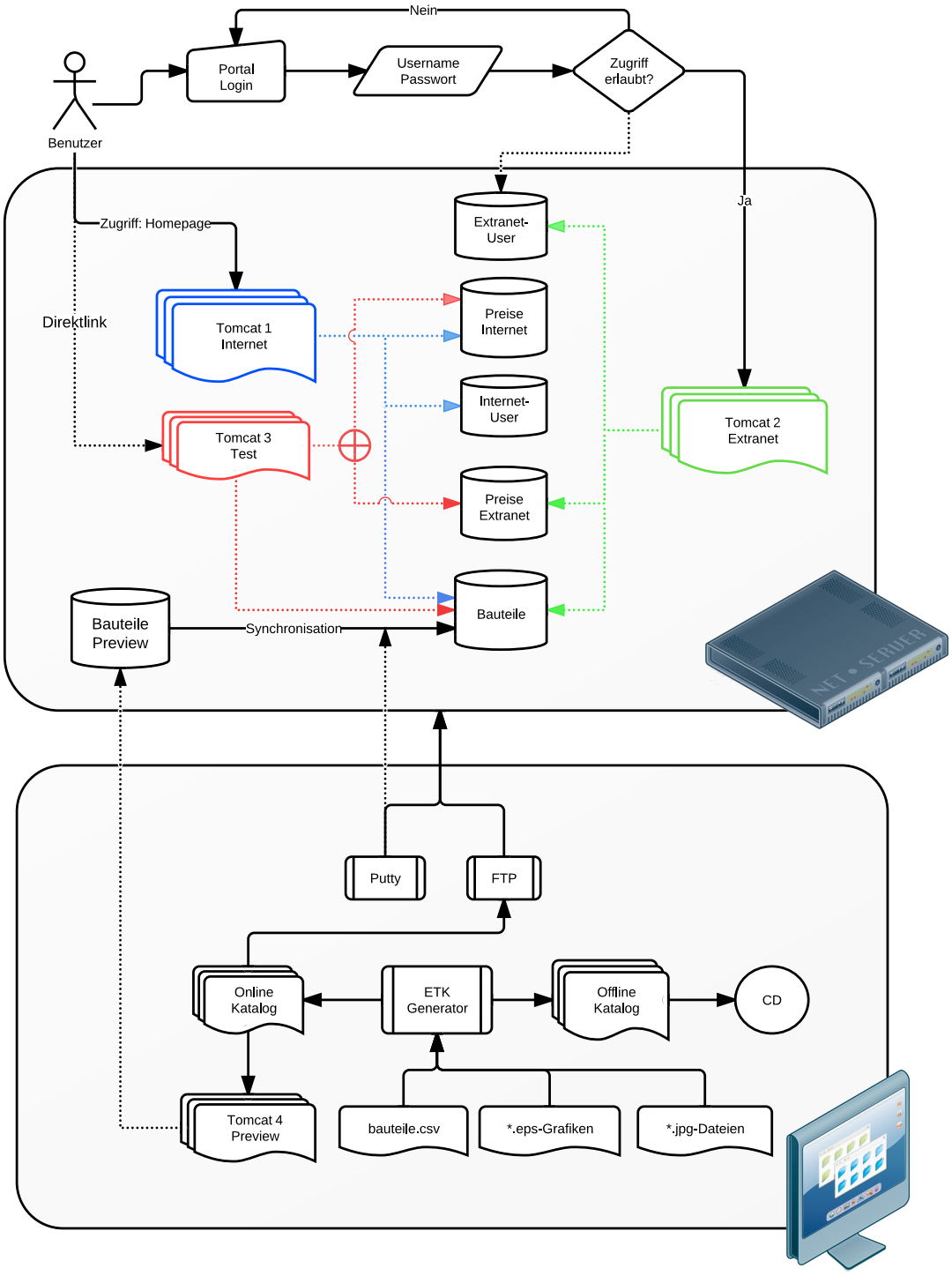


Abbildung 16: Komplette ETK-Struktur online und offline inklusive verwandter Prozesse

Ebenso soll für den neuen Generator die Bedienerführung verbessert werden. Dafür wird ein neuer Generator auf Basis aktuellster Java-Laufzeitumgebungen konzipiert, der modular aufgebaut ist.

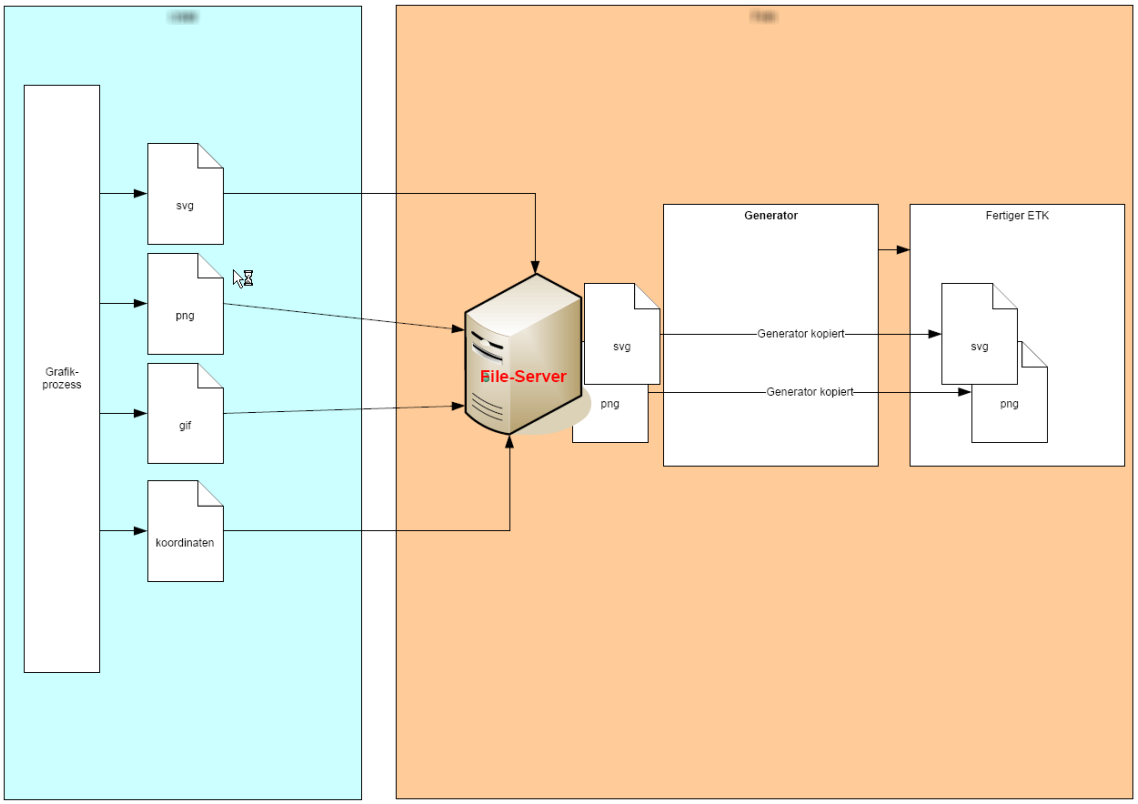


Abbildung 17: Neuer Grafikprozess

So lassen sich die einzelnen Schritte des bisherigen Prozesses einfach in ein neues Gesamtsystem einbinden und automatisieren. Der Benutzer muss wesentlich weniger komplexe Aufgaben ausführen, um einen neuen Katalog zu erzeugen und ihn online verfügbar zu machen.

Dabei verändert sich nur die Struktur des lokalen Generators[17]. Die Server-Struktur bleibt identisch. Lediglich der vom Generator erzeugt HTML-Code wird aufgrund der neuen Funktionsbibliothek wesentlich geändert, da die bisherige HTML-Struktur nicht mit den neuen Funktionen vereinbar ist.

Bei der Analyse der vom Grafikdienstleister gelieferten SVG-Grafiken wurde festgestellt, dass die ursprünglich zur Identifikation von Bauteilen herangezogenen Bauteilnummern in dieser Form im Quellcode des SVG nicht vorhanden sind. Stattdessen sind der SVG-Grafik mehrere `<hotspot>`-Elemente hinzugefügt worden, die die Position und die Nummer der Bauteile in der Grafik beinhalten. Diese müssen von der Funktionsbibliothek dynamisch ausgelesen werden, um die korrekte Zuordnung zwischen Bauteilen und deren Nummern herzustellen.

[17] vgl. Abb. 16 — Der untere Teil der Grafik zeigt den lokalen Generator.

Listing 28: Hotspot-Elemente im SVG

```
[...]
<tspan x="0 4.448" y="0" id="tspan194" style="font-size:8px;font-variant:normal;
    font-weight:normal;writing-mode:lr-tb;fill:#231f20;fill-opacity:1;fill-rule:
    nonzero;stroke:none;font-family:Helvetica World;-inkscape-font-specification
    :HelveticaNeue">10</tspan>
[...]
<hotspot id="tspan194" x="438" y="250" width="14" height="11" font-family="
    Helvetica World" font-size="8" status="active">10</hotspot>
[...]
```

Aus dem Beispiel wird ersichtlich, dass die `id` des `hotspot`-Elements der des `tspan` entspricht. Dies ist prinzipiell ein Verstoß gegen die Richtlinien des W3C, da `id` ein Element eindeutig identifizieren muss. Daher darf eine `id` in einem Dokument stets nur ein einziges Mal vorkommen. Da das `hotspot`-Element jedoch eine proprietäre Erweiterung für die SVG-Grafik darstellt, wird es von den Parsern nicht erkannt und gefährdet daher nicht die Integrität und Konformität der Grafik.

Um die `hotspot`- mit den `tspan`-Element zu verbinden, wird eine neue Variable eingeführt, die ein sog. *Mapping*[18] durchführbar macht. Um diese Variable zu füllen, werden zunächst alle in der Grafik vorhandenen `hotspot`-Elemente gesucht und einzeln auf ihr `id`-Attribut untersucht.

Listing 29: Suchen und Analysieren der Hotspot-Elemente

```
var hotspots = image.svgObject.getElementsByTagNameNS("http://www.w3.org/2000/
    svg", "hotspot");
$(hotspots).each(function() {
  tspanIdMapping[this.firstChild.nodeValue] = this.id;
});

// Ergebnis-Array
tspanIdMapping : [
  120 = "tspan320",
  12 = "tspan344",
  653 = "tspan32",
  ...
]
```

Mit dem so aufgebauten Array von Mapping-Informationen lassen sich die Bauteilnummern (der Index des Arrays) eindeutig den `tspan`-Elementen in der Grafik zuordnen. Dafür muss der Aufruf, welcher beim Hervorheben einer Bauteilnummer folgendermaßen angepasst werden.

[18] Das ist einfach gesagt eine Zuordnung von einem Element zu einem anderen, also `hotstpot -> MAPPING -> tspan`

Listing 30: Angepasste Hervorheben-Methode

```
this.highlight = function(id, element) {

// Gets the element to highlight
if (typeof element == "undefined") {
  element = image.svgObject.getElementById("p" + id)
    || image.svgObject.getElementById(tspanIdMapping[id]);
}

// Create the array field to save the highlighted nodes if it doesn't exist
if (typeof image.highlightNodes[id] == "undefined")
  image.highlightNodes[id.toLocaleString()] = [];

// Highlight all nodes and save them to the array of already highlighted nodes
if (element.nodeName == "line" || element.nodeName == "path") {
  element.style.stroke = config.nodes.highlight.color;
  image.highlightNodes[id.toLocaleString()][image.highlightNodes[id.
      toLocaleString()].length] = element;
} else if (element.nodeName == "tspan") {
  element.style.fill = config.nodes.highlight.color;
  image.highlightNodes[id.toLocaleString()][image.highlightNodes[id.
      toLocaleString()].length] = element;
}

// Processes child nodes, if a group has to be highlighted
else if (element.nodeName == "g") {
  for (var i = 0; i < element.childNodes.length; i++) {

    // Recursively call highlight using the child nodes, so
    // every level of nested children is highlighted
    this.highlight(id, element.childNodes[i]);
  }
}
};
```

Die Methode, welche das Hervorheben wieder rückgängig macht, muss nicht angepasst werden. Da sie ein von der oben gelisteten Methode gefülltes, generisches Array durchläuft und somit nur indirekt mit der SVG-Grafik arbeitet, kann diese mit durch die obige Anpassung unverändert bleiben.

Diese Anpassung ist eine proprietäre Veränderung, die der Funktionsbibliothek ausschließlich für diesen Kunden eine zusätzliche Funktion verleiht. Für die Erstellung einer generischen Lösung ist diese Anpassung nicht zu empfehlen, da sie zu spezialisiert auf Kundenvorgaben ist.

6 Diskussion

6.1 Analyse der Lösungen

6.1.1 WebCGM

Eignung für technische Grafiken WebCGM ist für technische Grafiken sehr gut geeignet. Da der Standard in enger Zusammenarbeit mit leitenden Unternehmen der Industrie entwickelt wurde, ist hier eine sehr tiefgehende Unterstützung aller wichtigen Eigenschaften einer technischen Zeichnung gegeben. Sowohl CGM als auch WebCGM sind als Binärformat durch kleine Dateigrößen ausgezeichnet, wodurch auch komplexe Darstellungen schnell geladen werden können [LH03, Abschnitt „Encoding“].

Implementierung in Erstellungsprozesse Die Erstellung einer CGM-Grafik aus CAD-Daten ist vergleichsweise einfach, da oft bereits in die Programme eingebaut. Diese CGM-Formate lassen sich jedoch nur dann einfach in WebCGM umwandeln, wenn sie dem strikten Standard WebCGMs entsprechen - proprietäre Lösungen von Firmen müssen ggf. angepasst werden um den Anforderungen von WebCGM gerecht zu werden. Dadurch können Prozesse verlängert werden und sich die Auslieferung eines tauglichen Katalogs deutlich verzögern.

Usability für Endanwender Die Viewer für WebCGM sind benutzerfreundlich aufgebaut, bieten in den meisten Fällen auch die grundlegenen Möglichkeiten zur Navigation innerhalb der Grafik (Bewegung und Zoom). Die Interaktion zwischen individuellen Skripts und einer Befehlsschnittstelle der Viewer ist schwer bis unmöglich. Wenn überhaupt, muss der Hersteller des Viewers explizit eine öffentliche Application Programming Interface (API) zur Verfügung stellen, über welche eigene Befehle an den Viewer geschickt werden können, die dieser dann ausführt.

Manipulation der Grafiken Die verfügbaren Viewer für WebCGM stellen nur eine eingeschränkte API zur Manipulation von Grafiken zur Verfügung. Diese sind jedoch für die für ETKs wichtigen Funktionen wie Ein- und Ausblenden sowie Veränderungen der Farbe meistens ausreichend. Hier ergibt sich jedoch das Problem der Abhängigkeit von den Entwicklern dieser Viewer und die damit verbundene Unmöglichkeit der Entwicklung eigener Erweiterungen und Anpassungen.

Integration in Browser Die Einbindung der Viewer muss über eine Installation des Benutzers erfolgen, danach werden automatisch alle WebCGM-Inhalte einer Webseite erkannt und durch eingebettete Tags des Viewers ersetzt. Durch die sehr spezielle Einbindung der Viewer muss teilweise für jeden Browser ein eigener Viewer oder eine besondere Version des Viewers installiert werden.[19]

[19] http://www.cgmopen.org/webcgm/viewers.html

6.1.2 SVGWeb

Eignung für technische Grafiken SVG ist für die Darstellung technischer Grafiken geeignet, wenn auch nicht primär dafür konzipiert. Aufgrund der XML-Basis und der daraus resultierenden Dateigrößen ist es jedoch weniger gut für den direkten Austausch solcher Grafiken gedacht, sondern hauptsächlich für dynamische Anzeigen. Grundsätzlich ist SVG dafür geeignet, jedoch nur in zweiter Reihe nach CGM [LH04, Abschnitt „Conclusion“].

Implementierung in Erstellungsprozesse Die Erstellung von SVG-Grafiken aus CAD-Daten oder bereits vorhandenen CGM-Grafiken ist oft nicht direkt von der eingesetzten Software unterstützt. Wenn jedoch der Prozess zur Erstellung für SVG-Grafiken bereits besteht, sind die Anpassungen für den Einsatz gering. Oftmals muss jedoch der Umweg über (E)PS-Dateien gemacht werden, durch welchen gerade die Gruppierung von Daten gefährdet werden kann.[20]

Usability für Endanwender Die API von SVGWeb ermöglicht die Entwicklung sehr umfangreicher und mächtiger Programme, die dem Anwender weitreichende Möglichkeiten bieten, sich in der SVG-Grafik zurechtzufinden. Da die komplette API quelloffen ist, lassen sich beliebig komplexe Funktionen entwickeln und dem Anwender so ein sehr stimmiges Gesamtkonzept der Bedienung bieten. Auch die Individualisierung ist durch die offene API sehr einfach und in vollem Umfang möglich.

Manipulation der Grafiken Die Manipulation der Grafiken ist mit SVGWeb problemlos möglich. Es wird ein kompletter DOM-Baum über die Struktur der SVG-Grafik aufgebaut, der mit JavaScript bearbeitet werden kann. Die Änderungen sind sofort in der Grafik sichtbar, auch bei Verwendung des Flash-Renderers, ohne dass das Bild neu geladen werden muss.

Integration in Browser Durch die zunehmend verbreitete, native Unterstützung von SVG ist die Integration von SVG-Grafiken im Browser bereits möglich. Bei Verwendung von SVGWeb muss der Benutzers nur den Adobe Flash Player installieren, woraufhin die Funktionsbibliothek in vollem Umfang funktionsfähig ist. Ältere Browserversionen werden jedoch nur bedingt unterstützt, genauso wie ältere Versionen des Flash Players. Im Großen und Ganzen kann jedoch von einer ausreichenden Befriedigung der Anforderungen ohne zusätzliche Aufwände auf Seiten des Benutzers gesprochen werden.

[20] Diese Einschätzung basiert auf den Daten, welche durch das beauftragende Unternehmen geliefert wurden und ist somit mit einem Fragezeichen zu versehen!

6.2 Schlussfolgerung

6.2.1 Proprietäre ETKs

SVGWeb ist für die Umsetzung des interaktiven ETKs die richtige Wahl. Auch wenn nicht alle Anfoderungen komplett befriedigt werden können (z. B. das Einbinden von Rastergrafiken in die SVG-Grafik), so ist die Umstellung der vorhandenen Prozesse zur Erstellung des ETKs wesentlich kostengünstiger, als den kompletten ETK inklusive der Anzeigen auf eine alternative Technologie wie WebCGM umzustellen.[21]

Zusätzlich ist nach wie vor die Benutzbarkeit des ETKs in einer Browserumgebung eine der wichtigsten Anforderungen. Dieser kann WebCGM aufgrund des benötigten Viewers nicht gerecht werden. Es kann nicht davon ausgegangen werden, dass alle Benutzer bereit oder berechtigt sind, einen solchen Viewer auf ihrem Rechner zu installieren, weshalb nicht gewährleistet werden kann, dass alle Benutzer und interessierten Kunden die Möglichkeit haben, den ETK anzusehen.

Für proprietäre Anpassungen oder Weiterentwicklungen, stellt die Lizenz, unter der SVGWeb veröffentlicht wird, genügend Möglichkeiten, sodass auch angepasste Versionen der Software ohne Probleme auch öffentlich zugänglich eingesetzt werden können.

6.2.2 SVGWeb zur Anzeige in Browserumgebungen

Die Möglichkeiten, die SVGWeb bietet, sind beeindruckend. Das Projekt ist zwar erst im Beta-Stadium und dennoch ist es in der Lage, einen Großteil des SVG-Standards abzudecken und über den Flash-Renderer in allen Browsern zugänglich zu machen. Die Quelloffenheit ist ein Faktor, der eindeutig für den Einsatz dieses Programmes zur Erstellung von browserbasierten Anwendungen im Umgang mit vektorbasierten Grafiken spricht.

Gerade die Browserweiche, die es ermöglicht, je nach Browser des Clients entweder auf die native oder auf die Flash-Darstellung umzustellen, machen SVGWeb zu einem mächtigen Werkzeug für skalierbare Grafiken im Web-Umfeld. Die Entwicklung des Webs geht ohnehin in diese Richtung, wofür man mit dem Einsatz von SVGWeb gewappnet ist.

6.2.3 SVG zur Anzeige vektorbasierter Grafiken

Wie bereits in 6.1.2 beschrieben, ist SVG für die Anzeige generischer vektorbasierter Grafiken geeignet. Dabei spielt vor allem die leichte Lesbarkeit für Mensch und Maschine eine große Rolle. Während der Entwicklungen zu dieser Projektarbeit ist gerade die XML-Struktur von SVG ein entscheidender Faktor für die Einfachheit mit dem Umgang mit SVG gewesen.

So lässt sich eine SVG-Grafik in jedem beliebigen XML-Editor einfach öffnen[22], lesen und bearbeiten. Dies ist mit CGM-Grafiken nicht ohne weiteres möglich.

[21] Diese Einschätzung ist nur für den auftragenden Kunden korrekt. Für eine allgemein gültige Aussage müssen die bestehenden Systeme einzeln analyisert werden.

[22] Vorausgesetzt, man hat genügend Speicher für die teilweise mehrere MB großen Dateien verfügbar

Zwar gibt es auch hier gerade in der professionellen technischen Dokumentation genügend Viewer, jedoch sind diese meistens sehr teuer und damit für den normalen Kunden nicht erschwinglich.

6.3 Ausblick

Die Erweiterbarkeit von SVGWeb und der Funktionsbibliothek ist sehr gut, da beide Projekte quelloffen sind und dadurch auch im Falle eines Abbruchs von Seiten der bisherigen Entwickler eine eigenständige Weiterentwicklung möglich ist.

SVG selbst ist ein lebendiger Standard, der besonders durch die langsam fortschreitende aber kontinuierliche Einführung von nativer Unterstützung in Browsersystemen immer mehr an Bedeutung gewinnt. Die Möglichkeiten der Interaktion und Animation innerhalb von SVG-Grafiken sprechen ebenfalls für eine weitergehende Durchsetzung dieses Formats als Standard für jegliche vektorbasierten Grafiken für Web-Umgebungen.

Zwar ist mit WebCGM auch ein durchaus mächtiger Standard geschaffen worden, jedoch ist gerade die fehlende native Unterstützung eines der größten Probleme bei der effektiven Durchsetzung dieses Formats. WebCGM bietet für technische Grafiken definitiv die bessere Plattform, was vor allem mit dessen Wurzeln bei CGM zusammenhängt, welches als von der Industrie gestütztes Format im Offline-Bereich auf jeden Fall die am meisten eingesetzte Technologie ist und bleiben wird [LH, Abschnitt „Outlook - WebCGM“].

Die weitere Entwicklung von SVG ist unsicher, jedoch ist davon auszugehen, dass vektorbasierte Formate im Web immer wichtiger werden. Momentan zeigt die zunehmende native Unterstützung eine klare Tendenz hin zu SVG, da es als XML-basiertes Format leicht eingebunden und mit den bereits vorhandenen XML-Editoren und Parsern leicht gelesen, geprüft und bearbeitet werden kann.

Die Funktionsbibliothek und SVGWeb selbst werden weiterentwickelt, so dass auch die darauf basierenden ETKs weiterhin problemlos unterstützt und auf dem aktuellen Stand gehalten werden können.

Die weiteren Schritte des Projektes werden eine beispielhafte Implementierung des ETK wie in 5.3 beschrieben, die Entwicklung eines daraus hervorgehenden neuen Generators und dessen Weitergabe inklusive Schulung für den Kunden sein. Diese Schritte werden jedoch in der laufenden Betriebsphase nicht mehr stattfinden, sodass die Projektarbeit mit diesem Stand schließt.

Literaturverzeichnis

[Ado] Adobe System Incorporated. PC Penetration. http://www.adobe.com/de/products/flashplatformruntimes/statistics.html.

[Ham] David Hammond. Web Browser ECMAScript Support. http://www.webdevout.net/browser-support-ecmascript.

[ISO99] ISO/IEEC. Metafile for the storage and transfer of picture description information. PDF, Dezember 1999.

[Lar] Larson CGM Software. Summary of capabilities in the WebCGM 2.0 DOM. PDF.

[LH] Dieter Weidenbrück Lofton Henderson. Applicability of CGM versus SVG for technical graphics. http://www.cgmopen.org/technical/webcgm_svg.htm.

[LH03] Dieter Weidenbrück Lofton Henderson. Applicability of CGM versus SVG for technical graphics, Mai 2003. http://www.cgmopen.org/technical/cgm-svg-20030508-2.htm.

[LH04] Dieter Weidenbrück Lofton Henderson. Applicability of CGM versus SVG for technical graphics, April 2004. http://www.cgmopen.org/technical/cgm-svg-20040419.html.

[Sch] Jeff Schiller. SVG Support Table. http://www.codedread.com/svg-support.php.

[Wor03] World Wide Web Consortium. Scalable Vector Graphics 1.1 Specification. Recommendation, Januar 2003. http://www.w3.org/TR/2003/REC-SVG11-20030114/.

[Wor10] World Wide Web Consortium. WebCGM 2.1. Recommendation, März 2010. http://www.w3.org/TR/2010/REC-webcgm21-20100301/.

Abbildungsverzeichnis

1 Vergleich der Zoomstufen 2
2 Nativer SVG-Support verschiedener Viewer 7
3 Marktdurchdringung des Adobe Flash Players 7
4 XML-Verarbeitungsfehler bei nicht valider SVG-Grafik 11
5 Beispiel einer technischen SVG-Grafik für den ETK-Einsatz 12
6 Unterschiede in nativem Rendering von SVG-Grafiken 14
7 Erzwingen des Flashrenderers 15
8 Nicht hervorgehobenes Bauteil 21
9 Hervorgehobenes Bauteil 21
10 Vergleich der Renderer mit eingebetteten Rastergrafiken 24
11 Miniaturansicht der Originalgrafik 25
12 Bildausschnitt in der Originalgrafik und in der Miniaturansicht 26
13 Bewegbarer Bildausschnitt in der Miniaturgrafik 27
14 Laden der Bauteildaten durch ein unabhängiges Skript 31
15 Prozess der Grafikerstellung für den Einsatz im ETK 34
16 Komplette ETK-Struktur online und offline inklusive verwandter Prozesse 36
17 Neuer Grafikprozess 37

Listings

1 Beispiel ohne JavaScript-Bibliothek . . . 8
2 Beispiel mit JavaScript-Bibliothek (jQuery) . . . 8
3 Ungruppierte SVG-Grafik . . . 10
4 Gruppierte SVG-Grafik . . . 11
5 Entwicklungsvariante . . . 13
6 Produktivvariante . . . 13
7 Einbinden der SVG-Grafik . . . 13
8 Ersetzter embed-Tag . . . 13
9 SVG-Beispiel-Grafik . . . 15
10 Manipulation der SVG-Beispiel-Grafik . . . 16
11 Einfache JavaScript Immediately Invoked Function Expression . . . 16
12 Einbinden der Funktionsbibliothek . . . 17
13 Hervorheben eines Bauteils . . . 18
14 Aufbau des Arrays der hervorgehobenen Elemente . . . 18
15 Hervorhebungen entfernen . . . 19
16 Wrapper-Funktion toggleHighlight() . . . 20
17 Beispielaufruf von setTimeout() . . . 22
18 zoomIn-EventListener . . . 22
19 Auslesen der GET-Parameter eines URLs . . . 23
20 Anpassen der Adresse der SVG-Grafik anhand des Parameters . . . 23
21 Einbetten einer Rastergrafik . . . 24
22 Umrechnen einer Bewegung auf die Miniaturgrafik . . . 26
23 Umrechnen der Skalierung auf die Miniaturgrafik . . . 27
24 Bewegung des Bildausschnittes der Miniaturgrafik . . . 28
25 Konfiguration der Funktionsbibliothek . . . 30
26 Anfrage an Datenskript . . . 30
27 Schnittstelle JSON-Bauteildaten . . . 30
28 Hotspot-Elemente im SVG . . . 38
29 Suchen und Analysieren der Hotspot-Elemente . . . 38
30 Angepasste Hervorheben-Methode . . . 39