Universität Paderborn

Fachbereich Wirtschaftswissenschaften

Fach Wirtschaftsinformatik

Seminararbeit

„Serverseitige Generierung von dynamischen Scalable Vector Graphics (SVG) zur Darstellung von Businessgrafiken“

vorgelegt von

Alexander Lindhorst

Matrikelnummer 3485702

vorgelegt bei

Prof. Dr. Leena Suhl

Dipl. Wirt-Ing. Michael Scholz

Paderborn, den 20. Juni 2003

Inhaltsverzeichnis

1. Einleitung....3
1.1. Motivation und Zielsetzung....3
1.2. Aufbau der Arbeit....5

2. Grundlagen....6
2.1. Grundlagen von Scalable Vector Graphics....6
2.1.1. Beschreibung....6
2.1.2. Aufbau einer SVG-Datei....6
2.1.3. Besondere Eigenschaften von SVG-Grafiken....9
2.2. Das Apache Batik Projekt....12
2.3. Grundlagen von serverseitiger Inhaltsgenerierung....13
2.3.1. Webbasierter Zugriff mit CGI und Servlets....13
2.3.2. Cocoon-Framework....14

3. Serverseitige Generierung von dynamischen Scalable Vector Graphics....16
3.1. Generierung von dynamischen Scalable Vector Graphics....16
3.1.1. Framework zur Erstellung von Grafiken....17
3.1.2. Überführung in dynamische Scalable Vector Graphics....23
3.2. Serverseitige Generierung....25
3.2.1. Ein Cocoon-Transformer zur Erstellung von SVG-Businessgrafiken....25
3.2.2. Dynamische Businessgrafiken mit Servlets....27

4. Fazit 29

Literatur 30

Anhang A: Listings....32

Anhang B: Klassendiagramm....36

Abbildungsverzeichnis

Abbildung 1: Darstellung der SVG-Grafik....9
Abbildung 2: Wetterdaten als SVG-Grafik (Quelle: eigene Darstellung)....11
Abbildung 3: Cocoon-Pipeline-Aufbau in Anlehnung an [Langham/Ziegeler 2002, Seite 71]....15
Abbildung 4: Tortendiagramm mit Farbskala aus ColorFactory....23

Listingverzeichnis

Listing 1: XML-Prolog.... 7
Listing 2: Document Type Definition Verweis....7
Listing 3: svg-Element....7
Listing 4: rect-Element....8
Listing 5: circle-Element....8
Listing 6: path-Element....8
Listing 7: Beispieldatei in SVG.... 8
Listing 8: Elementaufbau am Beispiel des text-Elements....10
Listing 9: Wetterdaten als XML-Dokument....11
Listing 10: Java2D-Befehle....24
Listing 11: Resultierender SVG-Code....24
Listing 12: Eingabedefinition für den Transformer....26

Tabellenverzeichnis

Tabelle 1: Ablauf bei Events für verschiedene Tags....26

1. Einleitung

1.1. Motivation und Zielsetzung

Das Ende der neunziger Jahre des 20. Jahrhunderts war geprägt von einer rasanten Verbreitung des Internets (siehe [Domain Survey]) und insbesondere von hohen Zuwachsraten des Dienstes „World Wide Web“ (WWW). Die schnelle, unkomplizierte Verfügbarkeit von weltweit in der Form einer Webseite verteilt vorliegenden Informationen stellte sich als *die* wichtige Eigenschaft heraus, die das Medium innerhalb weniger Jahre so interessant werden ließ, dass mittlerweile 38% der Europäer ein Internetanschluss zur Verfügung steht (vgl. [Datenreport 2002]).

„Ein Bild sagt mehr als tausend Worte.“ Diese alte Volksweisheit hat auch im digitalen Zeitalter noch Bestand und so entwickelte sich mit zunehmender Verbreitung des WWW auch ein steigernder Bedarf für eine grafische Aufbereitung von dynamischen Informationen. Dieser Bedarf wurde verstärkt, als gegen Ende der neunziger Jahre viele Unternehmen sich ein eigenes Intranet aufbauten und ihre betrieblichen Informationssysteme mit einem Webfrontend für eben diese internen Netze ausstatteten. Als logische Konsequenz entstanden Laufzeitbibliotheken, die es möglich machten, Informationen auf dem Server dynamisch aufzubereiten und dem Konsumenten als Grafik – meist in einem Rasterformat – in eine Webseite eingebunden zur Verfügung zu stellen.

Heute gehen die Erwartungen an digital vorliegende Informationen einen entscheidenden Schritt weiter: Um dem letztendlichen Empfänger „Mensch“ eine breite Palette an Zusatzinformationen zur Verfügung stellen zu können, müssen Maschinen automatisiert weitergehende Auswertungen durchführen können. Dazu ist es notwenig, dass solche Informationen mit geringem Aufwand in bestehende Anwendungen integrierbar und über Systemgrenzen hinweg portierbar sind. Als Lösung dieser Problematik wird heute im Allgemeinen die Abkehr weg von proprietären Formaten hin zu offenen Standards gesehen. Besonders geeignet scheint hierbei die Verwendung der Extensible Markup Language (XML), die vom World Wide Web Consortium (W3C) spezifisch zur Lösung dieser Probleme entwickelt wurde. Mit ihr können textuelle Dokumente beispielsweise automatisiert auf Validität geprüft und in – wenigstens theoretisch – beliebige Zielformate transformiert werden. Außerdem können in XML-Dokumenten enthaltene Informationen verlässlich extrahiert und zum Beispiel in einer Suchmaschine vorgehalten werden.

Um diese Vorteile auch bei grafischen Dokumenten nutzen zu können, formulierte das W3C in XML eine Auszeichnungssprache zur Darstellung skalierbarer Vektorgrafiken, die mit dem Kürzel SVG (Scalable Vector Graphics) benannt wurde (siehe [Andersson et al. 2003]). Genau genommen handelt es sich bei solchen Dokumenten nicht um Grafiken, sondern vielmehr um Textdokumente, die aus Befehlen bestehen, mit denen auf dem verarbeitenden System die Darstellung einer entsprechenden Grafik vorgenommen werden kann.

Heute liegen Grafiken meist in einem Rasterformat wie beispielsweise im Graphics Interchange Format (GIF), Joint Photograph Expert Group Format (JPEG) oder Portable Network Graphics Format (PNG) vor. Solche Formate enthalten die darzustellenden Informationen in Form von Farbwerten für jedes enthaltene Pixel und sind damit aufgrund der großen Informationsmenge für

das menschliche Gehirn nur zu verarbeiten, wenn sie als Bild dargestellt werden. Außerdem sind solche Grafiken nach ihrer Erstellung statisch, in der Regel ist eine Skalierung die einzige Änderung, die im Nachhinein noch an ihnen vorgenommen werden kann. Diese ist in der Regel zusätzlich mit Qualitätsverlusten bei der Darstellung behaftet.

Die Verwendung von SVG-Grafiken bietet den genannten Formaten gegenüber verschiedene Vorteile. Zum einen haben SVG-Grafiken alle Eigenschaften von XML-Dokumenten wie Lesbarkeit, Validierbarkeit, Transformierbarkeit und die Möglichkeit zur automatisierten Informationsextraktion und -verarbeitung, da es sich bei SVG um eine XML-Anwendung handelt. Zum Anderen sind solche Grafiken Vektorgrafiken, das heißt, dass bei Größenänderungen oder Ausschnittsvergrößerungen die Vektoroperationen, die zur bisherigen Darstellung geführt haben, entsprechend transformiert werden und die gewünschte Darstellung somit verlustfrei erreicht wird. Ein weiterer Vorteil besteht darin, dass bei der Formulierung von SVG die verstärkte Nachfrage nach interaktiven Grafikformaten berücksichtigt wurde und entsprechende Elemente in die Spezifikation aufgenommen wurden. Zusätzlich stellt SVG wie alle XML-Dokumente eine Schnittstelle zum programmatischen Zugriff auf die enthaltenen Elemente zur Verfügung, so dass Interaktivität über die in der Spezifikation definierten Möglichkeiten hinaus auch mit Hilfe von individuellen Skripten erreicht werden kann (vgl. [Andersson et al. 2003, Seite 607]).

Als letztes ist als Vorteil anzuführen, dass für die Speicherung der textuellen Anweisungen für das Rendering der Grafiken in der Regel deutlich weniger Speicherplatz benötigt wird als – im Fall von Rasterformaten – für die Speicherung der Farbwerte einiger tausend Pixel, aus denen auch kleine Grafiken bestehen. Diese im Vergleich geringere Dateigröße führt zu kürzeren Übertragungszeiten, was bei Webanwendungen ein kritischer Faktor bezüglich der Akzeptanz beim Nutzer ist. Zusätzlich lassen sich textbasierte Informationen nach [Andersson et al. 2003, Seite 663] in der Regel sehr gut komprimieren, was zu einer weiteren Verkleinerung des Speicherbedarfs führt. Da per Spezifikation alle SVG-Implementierungen Kompression und Dekompression unterstützen müssen (vgl. [Jackson 2002]), kann dieser Mechanismus zur Ladezeitverkürzung in Webanwendungen bedenkenlos genutzt werden.

Unabhängig von objektiven Vorteilen kann ein solches Grafikformat nur dann Erfolg haben, wenn möglichst viele Hersteller Unterstützung für dieses Format in ihren Anwendungen implementieren. Da an der Spezifikation von SVG eine Reihe namhafter Branchenführer wie zum Beispiel AOL, Adobe Systems, Ericsson, IBM, Microsoft, Nokia und Sun Microsystems (vgl. [Andersson et al. 2003, Seite 4] für eine vollständige Aufzählung) beteiligt waren, ist diese Unterstützung für SVG in absehbarer Zeit zu erwarten. Entsprechende Äußerungen finden sich unter [SVG Testimonials]. Die umfangreiche Liste heute bereits vorliegender Werkzeuge unter [SVG Implementations], die das SVG-Format auf unterschiedlichen Ebenen unterstützen, zeigt, dass das Format bereits eine breite Basis hat. Darüber hinaus wird die zunehmende Unterstützung der Verarbeitung von XML-Textdokumenten in Browsern voraussichtlich ebenso dazu beitragen, dass XML-basierte Grafikformate wie SVG, die sich in solche Textdokumente unkompliziert einbetten lassen, verstärkt im World Wide Web Verwendung finden werden.

Mit SVG liegt ein Standard vor, der viele Unzulänglichkeiten der heute häufig verwendeten

Rastergrafiken beheben kann. In der vorliegenden Arbeit wird gezeigt, dass das Format den Aufgabenstellungen, die sich bei der Nutzung insbesondere im geschäftlichen Umfeld ergeben, nicht nur gerecht wird, sondern darüber hinaus auch Ansätze enthält, die in diesem Umfeld deutliche Vorteile gegenüber den bisher üblichen Rasterformaten aufweisen.
Ziel der Arbeit ist die Implementierung von serverseitig nutzbaren Komponenten zur Erstellung von ausgewählten Geschäftsgrafiktypen im SVG-Format. Anhand dieser Komponenten wird gezeigt werden, inwiefern die Verwendung des SVG-Formats Vorteile bietet bei der Interaktion mit dem Benutzer oder bei der Grafikerstellung durch Transformationsregeln aus Datenquellen, die Daten im XML-Format zur Verfügung stellen.

1.2. Aufbau der Arbeit

Zum besseren Verständnis beschäftigt sich Kapitel 2 detailliert mit den theoretischen Grundlagen der Aufgabenstellung. Dazu werden zunächst die Grundlagen von Scalable Vector Graphics näher erläutert erläutert. In diesem Bereich wird auf den typischen Aufbau eines SVG-Dokuments und einige typische Anweisungen zur Darstellung von Grafikprimitiven eingegangen. Im weiteren Verlauf stellt dieser Teil des Kapitels kurz ein Projekt vor, das sich unter dem Dach der Apache Foundation mit der Generierung von SVG-Grafiken beschäftigt.
Als weiterer Bestandteil der Aufgabenstellung beschäftigt sich das Kapitel theoretisch mit einigen heute üblichen Technologien im Bereich der serverseitigen Generierung von Inhalten. In diesem Kontext werden auch die Frameworks und Architekturen vorgestellt, für die die zu implementierenden serverseitigen Komponenten entwickelt werden.
Kapitel 3 identifiziert zunächst der Aufgabenstellung zu Grunde liegende Einzelprobleme und zeigt für diese Lösungen anhand der in Kapitel 2 erörterten Grundlagen auf. Im weiteren Verlauf werden diese Lösungen zur Generierung von Diagrammen in SVG erarbeitet und das Zusammenspiel der spezifischen Komponenten aufgezeigt.
Der zweite Teil des dritten Kapitels setzt sich mit der Einbettung der Lösungen zur SVG-Generierung aus dem ersten Kapitelabschnitt in serverseitige Komponenten auseinander und erläutert mit Bezug auf die Grundlagen aus Kapitel 2 das Zusammenspiel der einzelnen Komponenten. In diesem Zusammenhang wird auch auf eventuell weitere für die Zielplattformen notwendige Aspekte wie zu erstellende Document Type Definitions eingegangen. Das Kapitel schließt Beispiel zur serverseitigen Erstellung von SVG-Code, der clientseitig Interaktion mit dem Benutzer zulässt.
Als Fazit der Arbeit bewertet das vierte Kapitel die implementierte Lösung und betrachtet, ob und inwiefern sich die Verwendung von SVG für die Lösung als geeignet erwiesen hat. Ferner geht das Kapitel auf eventuelle weitere lohnende Schritte ein, die sich aus der gefundenen Lösung ergeben, und schließt mit einem Ausblick auf zu erwartende weitere Entwicklungen.

2. Grundlagen

2.1. Grundlagen von Scalable Vector Graphics

2.1.1. Beschreibung

Auf [W3C SVG Home] wird SVG wie folgt beschrieben: „SVG ist eine Sprache zur Beschreibung zweidimensionaler Graphiken in XML. Sie enthält drei Arten von Grafikobjekten: Vektorgrafikfiguren (z.B. Pfade aus geraden Linien und Kurven), Bilder und Text. Grafikobjekte können gruppiert, mit Stilinformationen versehen und aus schon dargestellten Objekten zusammengesetzt werden. Text kann [...] vorliegen, wie die Applikation ihn braucht, was die Durchsuchbarkeit und die Zugänglichkeit erhöht. Die Möglichkeiten umfassen eingebettete Transformationen, Zuschneiden von Pfaden, Alphamasken, Filter Objekte, Schablonen und Erweiterbarkeit.
SVG-Grafiken können dynamisch und interaktiv sein. Das Document Object Model (DOM) für SVG, das das gesamte XML DOM enthält, ermöglicht direkte und effiziente Vektorgrafikanimation durch Skripting.[1]“
SVG ist also eine in XML definierte Markupsprache zur Darstellung von Informationen, mit denen geeignete Programme grafische (unter Umständen animierte) Elemente ableiten und darstellen können. Wie viele andere XML-Formate auch, wurde SVG vom W3C unter Mitarbeit vieler Branchenführer entwickelt (vgl. [Andersson et al. 2003, Seite 4]). Derzeit liegt die Version 1.1 der Spezifikation als Empfehlung des W3C vor. Darüber hinaus gibt es Vorschläge zur Umsetzung von Subsets der SVG-Spezifikation für die Unterstützung auf Geräten mit weniger Rechenleistung wie Personal Digital Assistants und Mobiltelefone.
SVG wird mittlerweile von vielen Produkten auf verschiedenen Ebenen unterstützt (vgl. [SVG Implementations]), zur Einbindung von SVG in Webseiten ist derzeit jedoch noch die Verwendung eines Plugins notwendig. Im Moment handelt es sich dabei in den meisten Fällen um das SVG-Viewer-Plugin von Adobe Systems (http://www.adobe.com), seit einiger Zeit gibt es für Windows-Plattformen auch ein Plugin von Corel (http://www.corel.com). Allerdings gibt es auch Projekte zur nativen Darstellung von SVG in Browsern, zum Beispiel. das Mozilla SVG Projekt (siehe [Mozilla SVG]). Im Zuge der zunehmenden Unterstützung der direkten Verarbeitung von XML-Dokumenten in Webbrowsern ist aber auch bei anderen Browsern ein natives Rendering von SVG-Grafiken zu erwarten.

2.1.2. Aufbau einer SVG-Datei

Da es sich bei SVG um eine XML-Anwendung handelt, gelten die Anforderungen, die allgemein an XML-Dokumente gestellt werden auch für SVG-Dateien. Dies bedeutet insbesondere, dass SVG-Dateien wohlgeformt sein müssen. Wohlgeformtheit bedeutet, dass jedes verwendete Element aus einem öffnenden und einem schließenden Tag bestehen und durch diese begrenzt sein muss. Ferner

[1] Eigene Übersetzung

müssen SVG-Dateien wie alle XML-Dateien als ersten Dateieintrag einen XML-Prolog haben, der die XML-Version sowie den verwendeten Zeichensatz enthält. Für SVG-Dateien üblich ist die Verwendung des Unicode-Zeichensatzes UTF-8.

Listing 1: XML-Prolog

```
<?xml version="1.0" encoding="UTF-8"?>
```

Als zweiten Eintrag enthalten SVG-Dateien einen Verweis auf ihre Document Type Definition (DTD). Durch diesen Verweis können die Elemente der SVG-Datei gegen die angegebene DTD auf Korrektheit geprüft werden; eine XML-Datei mit Verweis auf eine DTD heißt daher validierbar.

Listing 2: Document Type Definition Verweis

```
<!DOCTYPE svg PUBLIC '-//W3C//DTD SVG 1.1//EN'
'http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd'>
```

Als dritter Eintrag in der SVG-Datei folgt die Deklaration des SVG-Elements, des obersten Knotens in der Dokumentstruktur. Dabei handelt es sich um das eröffnende Tag des Elements. Das schließende Tag ist der letzte Eintrag in einer SVG-Datei. Zwischen diesen beiden Tags werden die Anweisungen für das Rendering der Grafikprimitiven platziert.
Das SVG-Element verfügt über einige Attribute. Dabei handelt es sich um Stilattribute (z.B. Füllfarbe des Renderingkontexts) sowie um die Dimensionsinformationen Breite (`width`) und Höhe (`height`) des Renderingkontexts. Die Stilattribute müssen nicht unbedingt als Attribute des SVG-Elements formuliert werden, sie können alternativ auch mit Cascading-Stylesheet-Angaben im Attribut `style` platziert werden.
Als letzte Information müssen bei Verwendung von Attributen und Elementen aus mehreren XML-Anwendungen noch die entsprechenden Namespaces (auch der SVG-Namespace) im SVG-Element deklariert werden. Ein mögliches SVG-Element wird in Listing 3 dargestellt.

Listing 3: `svg`-Element

```
<svg width="190" height="80" version="1.0" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" font-family="&apos;sansserif&apos;"
stroke="black" stroke-width="1" stroke-dashoffset="0" font-weight="normal"
stroke-opacity="1">
...
</svg>
```

Zwischen dem öffnenden und dem schließenden Tag des Elements werden die Direktiven für die Grafikprimitiven eingebettet. Eine vollständige Beschreibung der verfügbaren Anweisungen würde den Rahmen dieser Arbeit sprengen; stattdessen werden hier nur exemplarisch die Befehle zur Darstellung von Rechtecken und Kreisen sowie eines Dreiecks als Beispiel für eine zusammengesetzte Figur, einen so genannten Pfades, erläutert.
Rechtecke werden dargestellt, indem von anzugebenden Koordinaten ein Rechteck mit anzugebender Höhe und Breite gezeichnet wird. Das Element für Rechtecke hat den Namen `rect`, die entsprechenden Attribute lauten `x`, `y`, `height` und `width` respektive. Außerdem kann das Rechteck

abgerundete Ecken erhalten, indem mit den Attributen `rx` und `ry` die X- und Y-Werte der Radien der zu verwendenden Ellipsenausschnitte angegeben werden.

Listing 4: `rect`-Element

```
<rect x="10" y="10" width="50" height="50" rx="2" ry="2"
style="stroke:black;stroke-width:1mm;fill:blue;"/>
```

Ein Kreis wird mit dem `circle`-Befehl erstellt. Dieser benötigt als Parameter die Koordinaten des Kreismittelpunkts auf der X- und Y-Achse (`cx` bzw. `cy`) sowie den zu verwendenden Radius `r`.

Listing 5: `circle`-Element

```
<circle cx="95" cy="35" r="25" style="stroke:black;stroke-
width:1mm;fill:red;"/>
```

Um ein Dreieck darzustellen, muss ein Pfad entlang der Außenlinie des Dreiecks gezeichnet werden. Dies geschieht mit dem `path`-Element. Die Daten des Pfades werden im Attribut `d` platziert. Diese Daten bestehen wiederum aus einer Aneinanderreihung von einzelnen Zeichenoperationen. Im Falle eines Dreiecks muss der „virtuelle Zeichenstift" zunächst zu den Startkoordinaten bewegt werden (`M` bzw. `m`) und von da aus müssen hintereinander 3 Linien zu den jeweils anzugebenden Endkoordinaten gezeichnet werden (`L` bzw. `l`). Dabei deutet ein in Kleinbuchstaben übergebener Befehl, dass die angegebenen Koordinaten relativ zum Ausgangspunkt zu interpretieren sind. Großbuchstaben bedeuten, dass es sich um absolute Koordinaten handelt. Zum Schluss muss der Pfad noch mit dem Befehl `z` geschlossen werden.

Listing 6: `path`-Element

```
<path d="M130 60 l50 0 l-25 -50 l-25 50 z" style="stroke:black;stroke-
width:1mm;fill:green;"/>
```

Insgesamt ergibt sich die SVG-Datei wie in Listing 7 dargestellt und erzeugt die in Abbildung 1 gezeigte Darstellung.

Listing 7: Beispieldatei in SVG

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC '-//W3C//DTD SVG 1.1//EN'
'http://www.w3.org/TR/2003/SVG11/REC-SVG-20030114/DTD/svg11.dtd'>
<svg width="190" height="80" version="1.0" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" >
<rect x="10" y="10" width="50" height="50" rx="2" ry="2"
style="stroke:black;stroke-width:1mm;fill:blue;"/>
<circle cx="95" cy="35" r="25" style="stroke:black;stroke-
width:1mm;fill:red;"/>
<path d="M130 60 l50 0 l-25 -50 l-25 50 z" style="stroke:black;stroke-
width:1mm;fill:green;"/>
</svg>
```

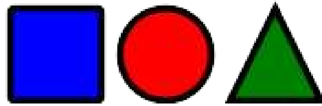


Abbildung 1: Darstellung der SVG-Grafik

Mit dem g-Element können Gruppierungen vorgenommen werden. Änderungen von Eigenschaften eines g-Knotens wirken sich auf die Kindknoten des Knotens so aus, als wäre bei ihnen allen die Eigenschaft geändert worden. Auf diese Weise ist es möglich, einzelne Elemente zu einem größeren Ganzen zusammenzufassen und dadurch eine Semantik in den SVG-Code zu bringen. Bei [Jackson 2002] heißt es: "Ein Bild beschrieben als vier Räder, eine Karosserie, einige Türen und Sitze stellt wahrscheinlich irgendein Fahrzeug dar.[2]" Dies wird noch deutlicher, wenn diese Elemente in einer Gruppierung mit der ID "Auto" zusammengefasst werden.
Diese Semantik stellt gleichzeitig eine wichtige Voraussetzung für den dynamischen Zugriff durch Skripte auf Teile der Grafik dar. Diese Skripte müssen dadurch nur auf ein Element zugreifen und können dabei gleichzeitig alle gruppierten Elemente verändern. Wenn die oben erwähnte Gruppe "Auto" mit ihren Einzelteilen bewegt wird, entsteht wohl am ehesten der Eindruck eines fahrenden Autos, nicht der Eindruck von vier sich bewegenden Reifen und einer sich bewegenden Karosserie.

2.1.3. Besondere Eigenschaften von SVG-Grafiken

Wie in der Einleitung angedeutet, handelt es sich bei SVG um eine XML-Anwendung. Dadurch erbt SVG von XML einige besondere Eigenschaften, die für die Verwendung des Formats im World Wide Web von bedeutendem Vorteil sind. Diese Eigenschaften werden im Folgenden näher erläutert.

Textbasiert

Anders als Rastergrafikformate wie die weit verbreiteten Formate Graphics Interchange Format (GIF), Joint Photograf Expert Group (JPEG) Format und Portable Network Graphics (PNG) enthalten SVG-Dateien keine Farbwertinformationen für einzelne Pixel. Stattdessen bestehen sie wie bereits in Kapitel 2.1.2 gezeigt aus textuellen Anweisungen zur Darstellung von Grafikprimitiven wie Ellipsen, Kreisen, Rechtecken, Linien etc. sowie zur Darstellung von Text und zur Durchführung von Animationen. Da es sich bei diesen Anweisungen um XML handelt, sind diese Anweisungen auch für Menschen relativ einfach zu lesen und zu verstehen.
Diese Lesbarkeit bietet Vorteile bei der Erstellung von SVG-Grafiken. Zum einen sind SVG-Anweisungen mit einer Semantik behaftet, die dafür sorgt, dass der Bearbeiter schon beim Lesen des Quellcodes einen gewissen Eindruck vom Ergebnis bekommt. Beispielsweise wird es dadurch möglich, dass Mitarbeiter der Marketingabteilung eines Unternehmens für eine Marketingaktion im Internet ein bestehendes Logo selbstständig mit einem neuen Text versehen. Dazu muss lediglich der Wert des entsprechenden `text`-Knotens in einem Editor geändert werden. Bei Rastergrafiken muss für eine solche Änderung in der Regel durch einen Grafiker eine neue Grafik erstellt werden, weil der

[2] Eigene Übersetzung

Aufbau einer Rastergrafik für den Marketingmitarbeiter nicht verständlich ist und er nicht das Know-How für die notwendigen Zeichenoperationen zur Erstellung der Grafik hat.
Zum anderen bietet die Lesbarkeit durch Menschen laut [Jackson 2002] den Vorteil, dass Entwickler von den Werken anderer Entwickler – ähnlich wie bei HTML in den Anfangszeiten des World Wide Web – lernen können. Dadurch kann die Verfügbarkeit von professionellen Anwendungen auf Basis von SVG beschleunigt werden.

Wohlgeformt

In Kapitel 2.1.2 wurde bereits erläutert, dass eine SVG-Datei als wohlgeformt bezeichnet wird, wenn jedes ihrer Elemente durch ein öffnendes und ein schließendes Tag begrenzt sein muss; als Tag bezeichnet man in XML eine Markierung, die aus dem Elementnamen (unter Umständen ergänzt durch Attribute und ihre Werte) besteht und mit dem Zeichen „<“ beginnt und mit dem Zeichen „>“ endet. Der textuelle Wert eines Elements steht zwischen dem öffnenden und dem schließenden Tag. Im schließenden Tag wird dem Elementnamen ein Schrägstrich vorangestellt[3]. Das Aussehen eines Elements ist am Beispiel des `text`-Elements in Listing 8 nachvollziehbar.

Listing 8: Elementaufbau am Beispiel des `text`-Elements

```
<text x="20" y="20">Ein SVG-Text</text>
```

Da Wohlgeformtheit die Reichweite eines Elements klar begrenzt, können alle XML-Dokumente, und damit auch SVG-Grafiken, verlässlich maschinell eingelesen und geparst werden; anders als zum Beispiel bei HTML gibt es keine Grauzonen bezüglich der Elementreichweite, die Interpretationen notwendig machen. Verschiedene Interpretationen der vorliegenden Daten im Kontext verschiedener Applikationen führen zu unzuverlässiger Weiterverarbeitung oder machen diese vollständig unmöglich. Bei XML-Dokumenten kann durch jede beteiligte Anwendung exakt festgestellt werden, welche Information zu welchem Element gehört.
Dadurch ist es in SVG anders als in Rastergrafiken möglich, enthaltene Textinformationen zu extrahieren und diese anderen Anwendungen zur Verfügung zu stellen. Zum Beispiel könnten in SVG erstellte technische Zeichnungen auf die Verwendung spezifischer Baugruppen durchsucht und direkt aus den Zeichnungen heraus Links auf die Teiledatenbank eines Zulieferers erstellt werden. Unter [Corel Supplier Sample] ist ein solches Szenario beispielhaft realisiert.

Validierbar

Ebenfalls in Kapitel 2.1.2 wurde bereits auf die Validierbarkeit von SVG-Dokumenten durch den Verweis auf eine Document Type Definition (DTD) hingewiesen. Dadurch können SVG-Dokumente verlässlich auf Korrektheit überprüft werden. Wenn eine SVG-Grafik von einem Programm korrekt gerendert wurde, muss ein anderes Programm, das dieselbe SVG-DTD unterstützt, die Grafik exakt genauso darstellen.
Fehlerhafte SVG-Grafiken können als Ganzes abgelehnt werden, so dass Daten nicht potenziell fehlerhaft oder in einem falschen Kontext dargestellt werden, und das rendernde Programm kann

[3] Alternativ können Knoten ohne eigenen Wert aus nur einem Tag bestehen, das den Schrägstrich als vorletztes Zeichen enthält.

zusätzlich einen genauen Hinweis darauf geben, warum die Darstellung gescheitert ist.

Transformierbar

Weil SVG-Dokumente textbasiert, wohlgeformt und validierbar sind, können sie automatisch weiterverarbeitet werden. Insbesondere ist es möglich, SVG-Grafiken auf der Basis von Transformationsregeln in andere XML-Formate, also auch in eine andere SVG-Grafik zu überführen. Andererseits können SVG-Grafiken auch als Ergebnis einer Transformation aus beliebigen XML-Dokumenten abgeleitet sein.

So können die Daten einer XML-Quelle empfängergerecht aufbereitet werden. Während zum Beispiel die in XML zur Verfügung gestellten Daten eines Wetterdienstes für weiterverarbeitende Maschinen bereits eventuell im richtigen Format vorliegen, können dieselben Daten für menschliche Empfänger mit einer Transformation grafisch aufbereitet werden. So können die Daten aus Listing 9 in die Darstellung aus Abbildung 2 überführt werden[4].

Listing 9: Wetterdaten als XML-Dokument

```
<?xml version="1.0" encoding="ISO8859_15"?>
<!DOCTYPE weather [
      <!ELEMENT wheather (maxtemp|mintemp|sky)* >
      <!ELEMENT maxtemp (#PCDATA)>
      <!ELEMENT mintemp (#PCDATA)>
      <!ELEMENT sky EMPTY>
      <!ATTLIST sky appearance (sunny|cloudy|rainy) #REQUIRED>
      <!ENTITY deg "&#186;C">
] >
<weather>
      <sky appearance="sunny" />
      <mintemp>15&deg;</mintemp>
      <maxtemp>38&deg;</maxtemp>
</weather>
```

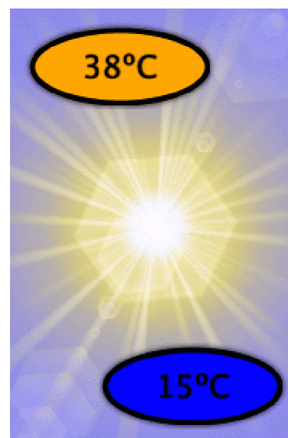


Abbildung 2: Wetterdaten als SVG-Grafik (Quelle: eigene Darstellung)

Elementezugriff via Document Object Model

Laut [Le Horst et al. 2000] handelt es sich beim Document Object Model (DOM) um „eine Anwendungsschnittstelle (API) für gültige HTML- und valide XML-Dokumente. Sie definiert die

[4] Alle für die Transformation notwendigen Dateien liegen der Arbeit in elektronischer Form sowie im Anhang bei.

logische Struktur und die Art und Weise, auf die auf ein Dokument zugegriffen und es manipuliert wird. [...] Sie basiert auf einer Objektstruktur, die stark der Struktur des Objekts gleicht, das sie modelliert.[5]"

Über die Schnittstelle DOM kann also programmatisch auf die einzelnen Elemente und die Werte und Attribute der einzelnen Elemente eines XML-Dokuments zugegriffen und diese verändert werden. Dadurch bieten XML-Dokumente externen oder auch eingebetteten Skripten Möglichkeiten zur Realisierung von Interaktionen mit dem Nutzer.

[W3C SVG Home] weist darauf hin, dass das SVG-DOM das komplette XML-DOM enthält und damit die direkte und effiziente Animation durch Skripting erlaubt. Ferner verfügt SVG über eine umfangreiche Sammlung an Event Handlern, die einem SVG-Objekt zugewiesen werden können, und erlaubt Skripting von verschiedenen Bereichen einer Webseite aus.

Eine vollständige Liste der SVG-DOM-Elemente sowie ihrer Eigenschaften und zuweisbarer Event Handler ist bei [Andersson et al. 2003, Seite 607ff.] zu finden.

Vektorbasiert

Bei SVG-Grafiken handelt es sich um Vektorgrafiken. Das heißt, dass die erzeugte Darstellung das Ergebnis von auf dem Zielsystem durchgeführten Berechnungen ist. Insbesondere impliziert dies, dass bei diesen Berechnungen spezifische Eigenschaften des darstellenden Systems berücksichtigt werden können. So können beispielsweise in einer technischen Zeichnung alle Bemaßungen in metrischen Einheiten angegeben sein. Auf dem Zielrechner können diese Zeichnungen dann unter Berücksichtigung der vorliegenden Bildschirmauflösung in realer Größe dargestellt werden. Bei einer Rastergrafik kommt es unter Umständen zu Abweichungen, wenn auf dem Erstellungs- und dem Darstellungssystem unterschiedliche Auflösungen verwendet werden. Außerdem kann bei einer Rastergrafik eine solche Zeichnung in der Regel nicht verlustfrei skaliert werden; bei einer Vektorgrafik wird dieser Verlust durch Anpassung der Berechnungen im Renderingprozess vermieden.

2.2. Das Apache Batik Projekt

Unter [Batik Overview] wird das Batik-Toolkit beschrieben als „ein auf Java™ Technologie basierendes Toolkit für Applikationen oder Applets, die Bilder im Scalable Vector Graphics (SVG) Format für verschiedene Zwecke wie das Betrachten, die Generierung oder die Manipulation benutzen wollen.[6]" Auf derselben Seite wird eine Liste von Produkten aufgeführt, die das Batik-Toolkit in Teilen oder ganz verwenden. Für die Reife des Projekts spricht, dass Branchenschwergewichte wie z.B. Oracle auf dieses Projekt zurückgreifen, um SVG in den eigenen Produkten verwenden zu können.

Inhalt des Projekts ist die Erstellung und die Pflege des gleichnamigen Toolkits. Das Toolkit besteht aus drei Teilen, von denen der erste sich dem Rendern von SVG-Grafiken widmet; d.h. das Toolkit enthält einige Java-Klassen, denen man eine SVG-Grafik übergibt und die dann diese Grafik rendern

5 Eigene Übersetzung

6 Eigene Übersetzung

können. Die in SVG formulierten Zeichenanweisungen werden dabei in Java2D-Zeichenbefehle überführt, mit denen die Grafik schließlich dargestellt wird.
Der zweite Teil des Toolkits funktioniert genau andersherum: Er übersetzt Java2D-Zeichenbefehle in SVG-Befehle, so dass man sich das Ergebnis von in Java ausgeführten Zeichenoperationen als SVG-Datei exportieren lassen kann. Zu diesem Zweck stellt das Toolkit eine Klasse zur Verfügung, die `java.awt.Graphics2D` erweitert. Auf diese Art kann man einer bestehenden Java-Applikation mit grafischer Oberfläche (GUI) eine Instanz dieser Klasse statt des systemeigenen graphischen Kontexts zur Darstellung auf dem Bildschirm übergeben und erhält diese Darstellung als SVG. Dieser Mechanismus erlaubt es, eine Reihe bestehender Toolkits z.B. zur Erstellung von Charts zu nutzen, um sich diese Charts in Zukunft als SVG-Grafik statt im Kontext einer GUI-Applikation ausgeben zu lassen.
Als dritte Komponente enthält das Toolkit Encoder, mit denen SVG-Grafiken unkompliziert in das JPEG oder das PNG Format überführt werden können. Dies ist insofern wichtig, als viele Anwender noch keinen SVG-Viewer für ihren Browser installiert haben und ihnen solche Grafiken in einem alternativen Format angeboten werden müssen. Allerdings kann die Interaktivität, die bei einer SVG-Grafik zur Verfügung steht, bei dieser Überführung nicht bewahrt werden.

2.3. Grundlagen von serverseitiger Inhaltsgenerierung

Zum besseren Verständnis der Anforderungen an die Lösungsfindung, die sich aus der angestrebten Einbettung der Komponenten zur Generierung von SVG-Grafiken in serverseitige Komponenten ergeben, werden im Folgenden einige aktuelle Technologien zur serverseitigen Verwendung erläutert, die in der Arbeit benutzt werden.

2.3.1. Webbasierter Zugriff mit CGI und Servlets

Mit steigender Verbreitung des WWW wuchs auch die Nachfrage nach dynamischen Webseiten. Daher wurde eine offene Schnittstelle, das Common Gateway Interface (CGI), entworfen, mit der Requestparameter an Programme zur Bearbeitung weitergegeben und das Ergebnis an den Webclient zurückgegeben wird, wenn die Programme das Interface implementieren. [Ayers et al. 2000, Seite 16] weisen darauf hin, dass diese Programme in nahezu jeder Programmiersprache erstellt sein können, was aber gleichzeitig den größten Nachteil dieser Lösung bedeutet, weil für jede Anfrage ein neuer Prozess gestartet werden muss und diese Lösung insbesondere bei Programmen in interpretierten Sprachen nur schlecht skaliert.
Sun Microsystems hat mit der Java Servlet API erfolgreich versucht, eine Lösung zur Erstellung dynamischer Webseiten in der Sprache Java zu finden, die die Probleme von CGI umgeht. [Ayers et al. 2000, Seite 17f.] führt verschiedene Aspekte auf, die Java Servlets zu einer besseren Lösung machen. Einer der größten Vorteile ist demnach die Plattformunabhängigkeit von Java, die Limitierungen beim Serverbetriebssystem umgeht. Weiterhin werden der Zugriff auf Enterprise Daten und Services mit den entsprechenden APIs ermöglicht. Die Servlettechnologie verfügt ferner über eine deutlich bessere Skalierbarkeit, weil Servletcontainer, in denen Servlets ablaufen, ein

Lebenszyklusmanagement betreiben, das einige wenige Objektinstanzen wiederverwendet und dadurch immer neue Prozesse vermeidet.
Die Qualität der Java Servlet Technologie und ihre Eignung zu Erstellung dynamischer Webanwendungen gegenüber bestehenden Lösungen lässt sich auch daran erkennen, dass direkte Konkurrenten von Sun Microsystems insbesondere bei Serverlösungen wie zum Beispiel International Business Machines, Inc. (IBM) diese Technologie für ihre eigenen serverseitigen Lösungen innerhalb kürzester Zeit übernommen hat.

2.3.2. Cocoon-Framework

Ein traditionelles Problem bei der Entwicklung von Webanwendungen ist die Vermischung von Kompetenzbereichen, die die Pflege, Wartung und Weiterentwicklung einer solchen Anwendung erschwert. Dies beruht darauf, daß in HTML Daten und Layout derselben stark gemischt sind (vgl. [Suhl et al. 2001, Seite 44]. Im Laufe des Entwicklungsprozesses kommen die Programmierer dadurch in die Situation, neben der Programmlogik auch die grafische Aufbereitung der Daten bereitstellen zu müssen, was meistens zu nicht optimalen optischen Ergebnissen sowie zu erheblichem Aufwand im Fall von Änderungen der zu Grunde liegenden Daten und deren Struktur führt.
Das Apache Cocoon-Projekt (vgl. [Cocoon Home]) ist entstanden, weil einer der Entwickler des Apache Projekts versucht hat, dieses Problem für sich zu umgehen. Das Ergebnis besteht im gleichnamigen, auf Java basierenden Framework, das es durch strikte Trennung von Logik und Präsentation ermöglicht, serverseitig dynamische Webseiten bereitzustellen, deren einzelne Teile vom jeweiligen "Profi" erstellt werden können; also z.B. die Programmlogik durch Programmierer, die Präsentation durch Grafiker. Diese Trennung wird im Cocoon Projekt als "Separation of Concerns" (SoC) bezeichnet(vgl. [Cocoon Overview]). Für diese Trennung wird auf XML als Zwischenformat zurückgegriffen, um sich die bereits in Kapitel 2.1.3 angeführten Eigenschaften dieses Formats im Prozess zu Nutze zu machen.
Bei dem Framework handelt es sich um eine Serverumgebung, die entweder als Java Servlet eingebettet in einer Servlet-Engine oder als unabhängiger Server von der Kommandozeile aus laufen kann. Informationen durchlaufen in diesem Framework eine Kette von Stationen, den [Cocoon Overview] als Pipeline bezeichnet. Eine solche Pipeline beginnt mit einem "Generator", der Daten aus beliebigen Quellen im XML-Format zur weiteren Bearbeitung erstellt. In der Folge werden diese Daten von 0 oder mehr "Transformern" in ein anderes XML-Format überführt; die häufigste Anwendung besteht hierbei in der Durchführung von XSL-Transformationen. Am Ende der Pipelinie steht ein "Serializer", der die XML-Daten in das gewünschte Präsentationsformat überführt. Zum Framework gehört ein Toolkit von Komponenten, das unter Anderem Serializer für die Formate XML, HTML, PDF, JPEG und PNG (aus SVG-Grafiken) enthält.
Abbildung 3 zeigt den grundsätzlichen Aufbau einer Cocoon-Pipeline. Eine genaue Erläuterung der einzelnen Komponenten einer solchen Pipelines sowie eine Beschreibung der zum Toolkit gehörenden Implementierungen der Komponenten findet sich bei [Langham/Ziegeler 2002, Seite

70ff.].

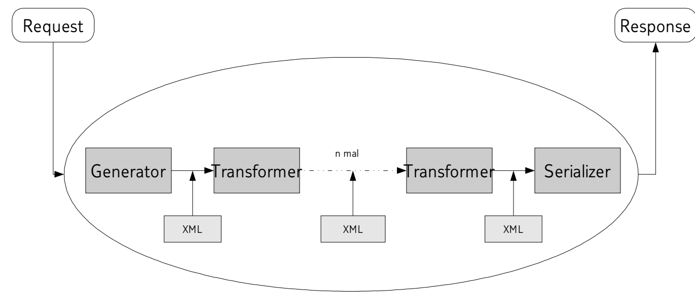


Abbildung 3: Cocoon-Pipeline-Aufbau in Anlehnung an [Langham/Ziegeler 2002, Seite 71]

3. Serverseitige Generierung von dynamischen Scalable Vector Graphics

Das vorliegende Kapitel widmet sich der Suche nach Möglichkeiten zur Erstellung von dynamischen Scalable Vector Graphics auf der Serverseite unter Verwendung der im vorhergehenden Kapitel vorgestellten Werkzeuge und Technologien. Diese Aufgabenstellung setzt sich aus zwei Hauptaspekten zusammen, nämlich aus Verfahren zur Erstellung solcher Grafiken sowie aus der Suche nach Möglichkeiten zur Anwendung dieser Verfahren auf dem Server. Damit die dabei entstehenden Teile nach ihrer Erstellung zusammen geführt werden können, ist zunächst nur eine Grundsatzentscheidung bezüglich der zu verwendenden Programmiersprache zu treffen, davon abgesehen können die beiden Aufgabenkomplexe zunächst fast vollständig getrennt voneinander betrachtet werden.

Wie das vorige Kapitel gezeigt hat, gibt es mächtige Werkzeuge und Programmierschnittstellen, die sich für die Aufgabenstellung eignen. Mit dem Apache Batik Toolkit steht ein leistungsfähiges Toolkit zur Erstellung von SVG-Grafiken zur Verfügung. Die Java-Servlet-Technologie und das darauf basierende Cocoon-Framework verfügen über ausgereifte Methoden zur Generierung von Inhalten auf einem Webserver. Da diese Technologien in der Programmiersprache Java implementiert sind, werden auch die Lösungsansätze der vorliegenden Arbeit in Java implementiert.

3.1. Generierung von dynamischen Scalable Vector Graphics

Wie bereits in Kapitel 2 erläutert, verfügt das Apache Batik Projekt unter Anderem über Klassen, die normale Java-Zeichenbefehle in SVG-Anweisungen überführen können. Dadurch ist es möglich, in Java realisierte GUI-Elemente statt auf dem Bildschirm in eine SVG-Datei ausgeben zu lassen. Bei solchen Elementen kann es sich beispielsweise um sogenannte Java Beans handeln, kleine Komponenten, aus denen eine Oberfläche zusammengesetzt wird. Ein Beispiel für eine solche Java Bean ist eine Komponente zur Darstellung von Businessgrafiken. Im Falle einer speziellen Anwendung kann die Kombination einer solchen Komponente mit den Möglichkeiten des Batik-Toolkits ausreichen, um SVG-Grafiken zu erzeugen.

Im vorliegenden Fall geht es allerdings um die Schaffung von generischen Möglichkeiten zur Erstellung von SVG-Grafiken; daher versucht diese Arbeit, ein Framework zu erstellen, das hinreichend generisch ist, um in möglichst vielen Fällen die Erzeugung von Geschäftsgrafiken zu erlauben.

Da mit Hilfe des Batik-Toolkits in Java erstellte Zeichnungen in SVG überführt werden können, kann der Aufgabenkomplex des vorliegenden Abschnitts wiederum unterteilt werden in die Erstellung eines Frameworks zur Darstellung von Grafiken mit den Mitteln der Programmiersprache Java sowie in die Anbindung des Batik-Toolkits an dieses Framework zwecks Überführung in SVG. Als Nebenprodukt dieser Vorgehensweise erhält man ein Framework, das seine Ergebnisse nicht nur in SVG-Dateien schreiben, sondern auch zur Darstellung von Grafiken in GUI-Programmen benutzt werden kann.

Im folgenden Abschnitt wird daher zunächst das Framework detailliert vorgestellt, in einem weiteren Abschnitt wird die Überführung der Ergebnisse in die SVG-Notation erläutert.

3.1.1. Framework zur Erstellung von Grafiken

Das im Zuge dieser Arbeit erarbeitete Framework unterstützt drei Grundtypen von Businessgrafiken:

- Tortendiagramm, nur dreidimensional
- Liniendiagramm, zweidimensional und dreidimensional
- Säulendiagramme, zweidimensional und dreidimensional

Die Komponenten zur Unterstützung dieser Grafiktypen stehen jedoch jeweils am Ende einer Vererbungshierarchie, bei der gemeinsame Aufgaben wenn möglich in gemeinsamen Oberklassen erledigt werden. Dies erhöht die Möglichkeit zur Wiederverwendung von Teilen des Frameworks für neue Aufgabenstellungen und hält den Aufwand zur Erstellung neuer Komponenten für andere Grafiktypen niedrig.

Das Framework nutzt den Java-typischen Container/Component-Ansatz. Dabei sind alle Elemente einer Oberfläche Komponenten zur Lösung einer ganz spezifischen Aufgabe und als solche Unterklassen von `java.awt.Component`. Solche Komponenten können zur Darstellung einem Container hinzugefügt werden, der sich dann um die Platzierung sowie um die Raumverteilung an die Komponenten kümmert. Als Container kommen alle Klassen in Frage, zu deren Vorfahren die Klasse `java.awt.Container` gehört. Da java.awt.Container ebenfalls eine Unterklasse von `java.awt.Component` ist, handelt es sich bei Container aber auch um Komponenten, die wiederum anderen Containern hinzugefügt werden können. Auf diese Art lassen sich Container mit Komponenten in andere Container rekursiv einbinden. Somit entsteht eine Baumstruktur mit Komponenten als Blättern und Containern als inneren Knoten, an dessen Spitze sich ein Container befindet, der gleichzeitg die oberste Ebene der GUI darstellt. Dabei kann es sich zum Beispiel um ein Anwendungsfenster handeln. Um nun das ganze Gebilde auf dem Bildschirm darzustellen, verteilt der oberste Knoten seine Zeichenfläche an seine Kindknoten. Handelt es sich dabei um Container, verteilen diese die ihnen so zugewiesene Zeichenfläche rekursiv an ihre Kindknoten weiter, bis die GUI komplett dargestellt ist.

Ausschlaggebend für die Darstellung einer Java-Komponente ist daher die Strategie, nach welcher die Zeichenfläche an die Kindknoten weitergegeben wird. Diese Strategie wird duch einen sogenannten Layout Manager (`java.awt.LayoutManager`) festgelegt, über den jeder Container verfügt. Ein solcher Layout Manager ermittelt von den beteiligten Komponenten ihren minimalen und ihren bevorzugten Platzbedarf und verteilt dann je nach Strategie auch eventuell überflüssige Zeichenfläche an die Kindknoten.

Um diesem Konzept folgen zu können, ist es daher notwendig, für jeden einzelnen Grafiktypen zu ermitteln, aus welchen Teilen er besteht und welche Rolle diese bei der Darstellung der Grafik übernehmen. Im Folgenden wird dies für die einzelnen Grafiktypen erläutert und gezeigt, wie die dabei gefundenen Komponenten im Framework eingebettet sind. Angegebene Klassen- und Paketangaben beziehen sich dabei auf das Paket `de.alexanderlinhorst.seminar.businessgraphics`. Methodennamen werden ohne Klammern und Argumente angegeben, sofern dies nicht notwendig ist, um zwischen zwei gleichnamigen Methoden mit verschiedenen Parametern zu unterscheiden. Ein UML-

Klassendiagramm, an dem die Ausführungen nachvollzogen werden können, befindet sich im Anhang.

Businessgrafiken im Allgemeinen

Allgemein kann für Grafiken Folgendes gesagt werden: Eine Grafik ist eine Darstellung von Datenreihen. Diese Datenreihen bestehen wiederum aus Daten mit einem Wert und einem Namen, der den Bezug des Wertes zu einem Referenzsystem ermöglicht. Dieser abstrakte Ablauf ist im Paket `chart` durch die abstrakte Klasse `Chart`, die die Grafik selbst darstellt, sowie durch die Schnittstellen `DataSet` für die Datenreihen und `Datum` für die einzelnen Werte umgesetzt. `Chart` verfügt über entsprechende Methoden zum Zugriff auf die Datenreihen. Die Klasse erbt direkt von java.awt.Container, was wiederspiegelt, dass es sich dabei um eine darstellbare Komponente handeln soll. Sie ist abstrakt, weil sie Grafiken im Allgemeinen modelliert, nicht etwa einen spezifischen Typ. `DataSet` enthält Methoden zum Zugriff auf die zur Datenreihe gehörigen Daten. Die Schnittstelle `Datum` stellt den Zugriff auf Name und Wert des jeweils darzustellenden Datums zur Verfügung. Es ist je nach Grafiktyp nicht klar, welche Rolle Datenreihen und Daten bei der grafischen Darstellung zukommt, dementsprechend wurden diese Bestandteile einer Grafik zunächst als Schnittstelle modelliert. Für die unterstützten Grafiktypen stehen, wo dies nötig ist, implementierende Komponenten zur Verfügung.

Bei der Spezialisierung auf die jeweiligen Grafiktypen kann zunächst eine grobe Unterscheidung gemacht werden in Typen, die für die Darstellung ihrer Werte Achsen als Referenzsystem benötigen, und in Typen, die ohne Achsen auskommen, wie zum Beispiel Tortendiagramme. Diese Unterscheidung wird im Framework nachvollzogen durch die beiden Unterklassen `PieChart` und `AbstractScaleChart`.

Tortendiagramme als achsenlose Diagramme

Ein Tortendiagramm stellt einen Spezialfall der oben aufgeführten Rollenverteilung innerhalb von Grafiken dar, weil hierbei Datenreihen und Daten in einer 1:1-Beziehung stehen. Ein Tortendiagramm stellt also verschiedene Datenreihen dar, die genau ein Datum beinhalten.

Die Klasse `PieChart` stellt die oberste Darstellungsebene eines Tortendiagramms dar. Aufgrund der Vererbungshierarchie handelt es sich dabei um einen Container, also eine Komponente zur Aufnahme anderer Komponenten. `PieSlice` ist für die Darstellung der einzelnen Datenreihen zuständig, die Klasse implementiert die Schnittstelle `DataSet`. Die Daten eines Tortendiagramms bildet das Framework mit der Klasse `PieItem` ab, die dementsprechend das Interface `Datum` implementiert.

Da Datenreihen und Daten in einer 1:1-Beziehung stehen, werden für die Daten selbst keine eigenen Komponenten benötigt. Daher handelt es sich bei der Klasse `PieSclice` nicht um einen Container, sondern um eine Komponente, bei `PieItem` handelt es sich um eine reine Datenstruktur zur Speicherung von Name und Wert. Um die einzelnen Stücke eines Tortendiagramms so darzustellen, dass sie “aneinandergelegt” und damit als ein Ganzes erscheinen, ist es notwendig, dass die Komponenten alle auf derselben Zeichenfläche operieren; dies ist eine Ausnahme, da in graphischen

Oberflächen normalerweise jede Komponente ihren eigenen, von dem anderer Komponenten abweichenden Zeichenraum hat. Daher ist diese gemeinsame Nutzung eines Zeichenraums mit den normalen Layoutstrategien von Java auch nicht zu erzielen. Stattdessen wird ein spezialisierter Layout Manager benötigt. Dabei handelt es sich um `PieLayout` aus dem Unterpaket `layout`. Da alle Komponenten in denselben Zeichenraum hineinzeichnen, besteht seine Aufgabe nicht in der Aufteilung des Zeichenraums, sondern darin, für alle Tortenstücke zu ermitteln, welcher Winkel ihrem Wert entspricht und um welchen Winkel versetzt sie sich darstellen müssen, um die anderen Tortenstücke nicht zu überzeichnen. Die ermittelten Werte reicht dieser weiter an die Methoden `setSliceAngle` bzw. `setOffsetAngle` der Instanzen von `PieSlice`. Diese Instanzen rufen dann in ihrer `paint`-Methode, die sie von java.awt.Component erben und überschreiben, die Methode `fillArc` mit den entsprechenden Winkelwerten auf.
Da sich die gewünschte Darstellung des Tortendiagramms nur dann erzielen lässt, wenn die genannten Komponenten verwendet werden, überschreibt `PieChart` die entsprechenden geerbten Methoden und stellt mit entsprechenden Prüfungen sicher, dass nur `PieLayout` als Layout Manager und Instanzen von `PieSlice` als Komponenten verwendet werden.
Die Implementierung dieses Grafiktyps zeichnet eine Ellipse, die perspektivisch den Eindruck eines liegenden Kreises und somit einer dreidimensionalen Darstellung erzeugt. Da eine zweidimensionale Darstellung durch die Verwendung eines echten Kreises erreicht wird und ein Kreis lediglich einen Spezialfall einer Ellipse darstellt, wurde auf eine entsprechende Implementierung verzichtet.

Diagramme mit Achsen

Die Verwendung von Achsen in Diagrammen ist im Falle der anderen beiden unterstützten Typen, bei Liniendiagrammen und bei Säulendiagrammen, üblich. Selbst wenn bei solchen Diagrammen auf die direkte Darstellung der Achsen verzichtet wird, sind die einzelnen Datenelemente normalerweise in Reihen entlang einer "gedachten" Achse angeordnet; wenn die Datenelemente zueinander in Bezug gesetzt werden, kann aus diesen Bezügen auf die Skalierung zurückgeschlossen werden. Die Darstellung dieser Achsensystemen ist in der Regel für Linien- und Säulendiagramm gleich. Sie besteht für zweidimensionale Darstellungen aus zwei rechtwinklig aufeinander stehenden Linien, auf denen die jeweiligen Werte (auf der Y-Achse) beziehungsweise die Namen der Datenreihen (auf der Z-Achse bei dreidimensionaler Darstellung) abgezeichnet sind. Normalerweise verfügen die Achsen über einen Titel, der andeutet, was auf der Achse abgebildet wird. Im Falle einer dreidimensionalen Darstellung wird eine dritte Achse den anderen beiden Achsen in einem Winkel hinzugefügt, der den gewünschten perspektivischen Eindruck erzeugt.
Diagramme mit Achsen werden im Framework durch die abstrakte Klasse `AbstractScaleChart` modelliert. Die Klasse ist abstrakt gehalten, da sie – wie oben angeführt – mehrere Diagrammtypen verkörpern kann. Sie stellt einen Container dar, der die Darstellung der Datendetails anderen Komponenten überlässt, für diese aber das Koordinatensystem zur Verfügung stellt. Zu diesem Zweck verfügt die Klasse neben der von `Chart` geerbten Funktionalität zur Verwaltung von Datenreihen über Methoden, die die Darstellung der Achsen regeln. Sie enthält dazu eine Variable der Datenstruktur `Scale`, die wiederum zwei oder drei (je nach gewünschter Darstellung) `Axis-`

Datenstrukturen enthält. Mit dem Verbund aus `Axis` und `Scale` kann im Graphen die Darstellung der Achsen manipuliert werden. Es ist mit den entsprechenden Methoden aus `Scale` möglich, die Darstellung von Achsen ein- und auszuschalten. Ferner können die Autoskalierung der Werte-Achse (Y-Achse) abgestellt und begrenzende Werte angegeben werden. Zur Darstellung der Achsen im Zeichenraum von `AbstractScaleChart` benutzt die Klasse spezielle Komponenten der Klasse `AxisHeader` beziehungsweise von deren Unterklassen. Auf diese Komponenten kann von außen nicht zugegriffen werden, so dass unerwünschte Effekte vermieden werden. Dabei ist es Aufgabe des Layout Managers sicherzustellen, dass zunächst die Achsen gezeichnet werden und dann die entsprechende Restfläche ermittelt wird, auf der die Datenkomponenten gezeichnet werden können.
Im Fall einer dreidimensionalen Darstellung kommt es auch hier zur Überlappung von Zeichenräumen, wenn zuerst diagonal eine dritte Achse (Z-Achse) potenziell über einen grossen Teil der Diagrammfläche eingezeichnet wird, und dieser Raum danach mit Datenelementen erneut beschrieben werden soll. Wie bei einem Tortendiagramm ist dies nicht mit den normalen Layoutstrategien von Java zu erreichen, weswegen auch `AbstractScaleChart` mit `ScaledLayout` über einen eigenen Layout Manager verfügt. Diese Klasse stellt die notwendigen Berechnungen an und berücksichtigt beim Layout zunächst die Achsen, um dann den Raum für die Restfläche zu errechnen. Damit in dieser Restfläche mehrere Komponenten platziert werden können, stellt `AbstractScaleChart` mit `ScaledPane` einen eigenen Container für die Komponenten bereit. Auf diesen Container kann von außen nicht zugegriffen werden, die Komponenten werden ihm durch die Methoden zur Verwaltung von Datenreihen hinzugefügt. Welche Beschaffenheit diese Komponenten haben, richtet sich nach dem darzustellenden Diagrammtyp.

Liniendiagramme

Liniendiagramme werden durch die Klasse AbstractLineChart beziehungsweise je nach benötigter Darstellungsart durch ihre Unterklassen `LineChart2D` und `LineChart3D`. Gegenüber den Methoden, die diese Klassen von `AbstractScaleChart` und `Chart` geerbt haben, bieten sie keine neue Funktionalität; allerdings überschreiben sie diese Methoden, um die Menge der möglichen Parametertypen auf die von diesen Grafiken benötigten Typen zu beschränken. So akzeptiert beispielsweise die weitervererbte Methode `addDataSet` in der Klasse `Chart` beliebige Objekte, die die Schnittstelle `DataSet` implementieren. Die konkrete Implementierung in der Klasse `LineChart2D` prüft allerdings, ob es sich bei den Parametern um Objekte der Klasse `Line2D` (oder Kinder davon) handelt. Auf diese Art wird sichergestellt, dass einem Liniendiagramm nicht etwa eine Komponente zur Darstellung von Säulen übergeben wird und dass die Oberklassen dennoch für die Implementierung von anderen Grafiktypen weiterverwendet werden können.
Damit ist schon angeklungen, daß auch die beiden Liniendiagrammtypen über ihre eigenen Komponenten zur Darstellung der Daten verfügen. Im zweidimensionalen Fall ist die Klasse `Line2D` für diese Darstellung zuständig; sie stellt die eine Datenreihe eines 2D-Liniendiagramms dar. Einzelne Punkte dieser Linie werden durch die Klasse `Dot` im Framework vertreten. Da es keinen Sinn macht, jeden einzelnen Punkt durch eine eigene Komponente darstellen und diese potenziell große Menge von einem Layout Manager anordnen zu lassen, handelt es sich bei `Dot` lediglich um

eine Datenstruktur. Die Darstellung wird von Line2D übernommen, bei der es sich um eine Komponente, nicht um einen Container handelt. Im Kontext der Klasse LineChart2D wird genau eine Instanz dieser Komponente dem ScaledPane aus der Klasse AbstractScaleChart hinzugefügt. Da die Achsen von den Oberklassen dargestellt werden, ist das Diagramm nach Zeichnen der Linie fertiggestellt.

Im dreidimensionalen Fall wird die Darstellung der Datenreihen von Instanzen der Komponente Line3D übernommen. Diese Klasse funktioniert im Prinzip genauso wie Line2D, allerdings zeichnet sie ihre Linie nicht als Linie, sondern durch das Hinzufügen von geringer optischer Tiefe als dreidimensionale Fläche. Im Gegensatz zu Line2D ist Line3D für die Darstellung von mehreren Datenreihen konzipiert. Um den perspektivischen Eindruck zu wahren, muss der verwendete Layout Manager die einzelnen Datenreihen leicht versetzt hintereinander anordnen. Dabei sind die Zeichenbereiche der einzelnen Datenreihen wieder überlappend angeordnet, was auch in diesem Fall einen eigenen Layout Manager erforderlich macht. Die Klasse Set3DLayout kann dreidimensionale Datenreihen versetzt hintereinander anordnen und in umgekehrter Reihenfolge rendern, so dass die vorderen Datenreihen notfalls die hinteren, schon gezeichneten Datenreihen überdecken können. Sie wird daher für das Layout der ScaledPane-Instanz verwendet, die als Container die verschiedenen Line3D-Datenreihen enthält. Außerdem berücksichtigt dieser Layout Manager, dass die einzelnen Datenreihen eventuell auf unterschiedlich großen Punktemengen basieren. Wenn beispielsweise eine Linie über 3 Punkte und eine über 6 Punkte gezeichnet wird, stellt der Layout Manager sicher, dass die erste Linie nur halb so viel Breite zur Verfügung hat wie die zweite Linie. Auf diese Art wird eine Vergleichbarkeit über die Datenreihen hinweg aufrecht erhalten.

Auch in diesem Fall werden die Achsen von den Oberklassen dargestellt, so dass das Diagramm mit dem Zeichnen der Flächen und der Anordnung der einzelnen Reihen beendet ist.

Säulendiagramme

Im Framework werden Säulendiagramme durch die Klasse AbstractColumnChart beziehungsweise durch ihre Unterklassen ColumnChart2D für zweidimensionale Diagramme und ColumnChart3D für dreidimensionale Diagramme modelliert. In ihrer Funktionsweise sind sie LineChart2D respektive LineChart3D sehr ähnlich; ColumnChart2D bildet eine einzelne Datenreihe mit zweidimensionalen Säulen ab, ColumnChart3D mehrere Datenreihen hintereinander mit dreidimensionalen Säulen. Im letzteren Fall geht die Ähnlichkeit der an die Klassen gestellten Aufgaben so weit, dass beide denselben Layout Manager Set3DLayout benutzen.

Ein grundlegender Unterschied tritt aber bei der Darstellung der einzelnen Datenelemente auf, da es für Säulen im Gegensatz zu Punkten auf Linien sinnvoll ist, diese als einzelne Komponenten von einem Layout Manager platzieren zu lassen. Daher handelt es sich bei den Elementen zur Darstellung der Datenreihen Row2D und Row3D um Container, denen ihre einzelnen Datenelemente als Komponenten (Column2D und Column3D respektive) hinzugefügt werden.

Bei zweidimensionaler Darstellung bestehen die Säulen aus Rechtecken, deren Höhe im Verhältnis zur Skala berechnet werden und dargestellt werden muss. Die Datenreihe wird durch die Anordnung

der von Column2D gezeichneten Säulen nebeneinander im Kontext einer Row2D-Instanz dargestellt, wobei der zur Verfügung stehende Platz gleichmäßig zwischen diesen aufgeteilt wird. Eine Überlappung der Zeichenbereiche ergibt sich dabei nicht, so dass eigentlich kein eigener Layout Manager zur Darstellung nötig ist. Da aber alle anderen Grafiktypen über ihren eigenen Layout Manager verfügen und um eventuell später zusätzliche Funktionen implementieren zu können, gibt es mit Column2DLayout auch für Row2D-Instanzen einen eigenen Layout Manager.

Die dreidimensionale Darstellung von mehreren Datenreihen mit Säulen ist dagegen etwas schwieriger. So muss beim Zeichnen der dreidimensionalen Säulen einer Datenreihe eine Reihenfolge so eingehalten werden, dass die räumlich weiter hintenliegenden Teile einer Säule von den weiter vorne liegenden Teilen der nächsten Säule überzeichnet werden können, wo dies durch die Perspektive notwendig ist. Hier kommt es also wieder zur Überlappung von Zeichenbereichen einzelner Komponenten; dementsprechend steht mit Column3DLayout ein Layout Manager zur Verfügung, der Column3D-Instanzen entsprechend im Kontext einer Row3D-Instanz platzieren kann.

Für die Anordnung der einzelnen Datenreihen der Klasse Row3D hintereinander gelten dieselben Schwierigkeiten, wie dies bei dreidimensionalen Liniendiagrammen der Fall ist. Auch hier müssen die einzelnen Datenreihen jeweils leicht versetzt dargestellt werden, auch in diesem Fall muss eine bestimmte Reihenfolge eingehalten werden, um einen Tiefeneindruck zu erzeugen. Außerdem ist ebenfalls notwendig, dass den einzelnen Datenreihen proportional zur Anzahl ihrer Elemente Breite zur Verfügung steht. Wie schon erläutert, ist der Layout Manager Set3DLayout für diese Aufgaben konzipiert und wird daher für das Layout der ScaledPane-Instanz herangezogen.

ColumnChart2D und ColumnChart3D erben wie die entsprechenden Liniendiagrammtypen von AbstractScaleChart, so dass die Achsen von dieser Oberklasse zur Verfügung gestellt werden. Nach Platzierung der Säulen und ihrem Rendering ist die Darstellung solcher Diagramme daher vollständig.

Farbskala

Farben sind bei Identifizierung von Zusammenhängen und Abgrenzungen insbesondere in Diagrammen mit vielen Elementen ein wichtiges Hilfsmittel. Daher benötigt das Framework neben den Methoden zur Platzierung der beteiligten Komponenten außerdem einen Mechanismus, der eine Farbskala für die Datenreihen bereitstellt. Im Framework wird diese Teilaufgabe durch die Klasse ColorFactory erledigt, die aus einem Anfangswert heraus 64 verschiedene Farben mit leicht variierenden Rot-, Grün- und Blauanteilen errechnet. Da die Farben aus nur leicht variierenden Schattierungen von Rot, Grün und Blau bestehen, liegen diese Farben subjektiv nahe beieinander; wie in Abbildung 4 zu sehen ist. Dadurch werden allzu starke Kontraste zwischen den einzelnen Datenreihen vermieden.

Dabei ist garantiert, daß alle Farben einer solchen Farbskala paarweise verschieden sind. Ferner ist garantiert, dass zwei Farbskalen mit demselben Startwert die gleichen Farben in der gleichen Reihenfolge zur Verfügung stellen. Der Startwert kann zufällig generiert oder bei der Instanzierung dem Konstruktor übergeben werden. Dadurch wird erreicht, dass bei dem mehrfachen Rendern einer Grafik immer dieselbe Farbskala verwendet wird.

Eine Instanz der Klasse `ColorFactory` wird von der abstrakten Klasse `Chart` über entsprechende Methoden zur Verfügung gestellt, so dass alle Diagrammtypen diesen Mechanismus auf einfache Art nutzen können.

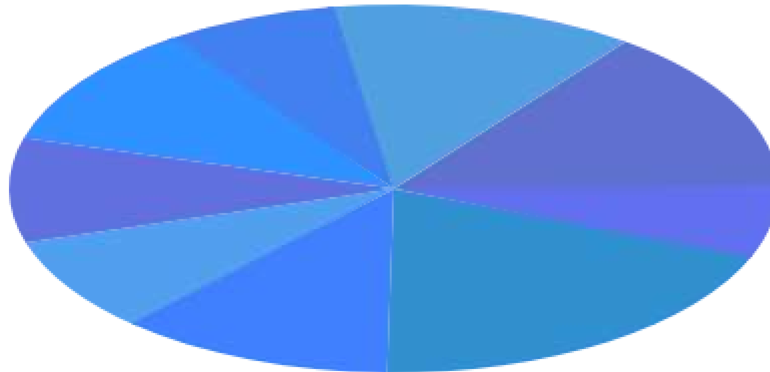


Abbildung 4: Tortendiagramm mit Farbskala aus `ColorFactory`

3.1.2. Überführung in dynamische Scalable Vector Graphics

Überführung von Java-Zeichenbefehlen in SVG-Befehle

In Kapitel 2.2 und in Kapitel 3.1 wurde bereits darauf eingegangen, dass ein Teil des Apache Batik Projekts Klassen zur Verfügung stellt, mit denen Java2D-Zeichenbefehle, also Befehle zum Zeichnen von Linien etc. in Java, in SVG-Anweisungen überführt werden können. Das konkrete Vorgehen besteht darin, den grafischen Kontext, den Java systemabhängig für das Zeichnen auf dem Bildschirm zur Verfügung stellt, zu ersetzen durch eine Instanz der Klasse `SVGGraphics2D` aus dem Batik Projekt. Wird jetzt die Methode `paint` einer Komponente, mit der die Darstellung dieser Komponente vorgenommen wird, mit diesem neuen Kontext als Parameter aufgerufen, werden die Zeichenoperation auf der `SVGGraphics2D`-Instanz ausgeführt, die die Überführung in SVG Anweisungen vornimmt.

In diesem Vorgehen besteht die einfachste Möglichkeit zur Erstellung von SVG-Dateien aus Java-Code. So ist es denkbar, ein Diagramm mit den im vorhergehenden Abschnitt vorgestellten Komponenten zu erstellen und der Diagrammkomponente den `SVGGraphics2D`-Kontext zur Darstellung der Grafik zu übergeben. Allerdings hat dieser automatisch generierte Code den Nachteil, dass die einzelnen Teile nicht in eine Semantik eingebettet sind. Die SVG-Grafik besteht nur aus einer Menge von einzelnen Anweisungen, eine Gruppierung zusammengehöriger Teile wie z.B. einzelner Säulen zu einer Datenreihe erfolgt nicht. Eine solche Gruppierung ist aber wie in Kapitel 2.1 ausgeführt insbesondere dann sinnvoll, wenn Zugriffe auf Teile der Diagramme mit dem Document Object Model erfolgen soll. Daher bedarf es zusätzlicher Schritte, um eine solche Gruppierung zu ermöglichen.

Gruppierung einzelner Teile für den dynamischen Zugriff

Listing 10 zeigt die Zeichenroutine einer Java-Klasse, die Batik benutzt, um den Inhalt eines auf dem

Bildschirm zu sehenden Applikationsfensters auch als SVG auf der Konsole auszugeben[7]. Die resultierenden SVG-Anweisungen sind dem in Listing 11 gegenüber gestellt. Dabei ist auffällig, dass der von Batik generierte Code Gruppen enthält.

Listing 10: Java2D-Befehle

```
public void paint(Graphics g) {
 g.setColor(Color.blue);
 g.fillRect(1,1, 100, 100);
 g.setColor(Color.red);
 g.fillOval(99,99, 100,100);
}
```

Listing 11: Resultierender SVG-Code

```
  <!--Generated by the Batik Graphics2D
SVG Generator-->
  <defs id="genericDefs" />
  <g>
    <g fill="blue" stroke="blue">
      <rect width="100" x="1"
height="100" y="1" stroke="none" />
      <circle fill="red" r="50"
cx="149" cy="149" stroke="none" />
    </g>
  </g>
</svg>
```

Die Dokumentation der API des Batik Projekts, zeigt, dass die Klasse `SVGGraphics2D`, die für das Überführen der Zeichenbefehle zuständig ist, intern mit einem DOM-Baum arbeitet, der von einer Instanz der Klasse `DOMTreeManager` verwaltet wird. Diese Klasse wiederum hat eine Methode `getTopLevelGroup`, die eine Referenz auf die oberste Gruppierungsebene des DOM-Baums zurückliefert. Gleichzeitig wird diese oberste Ebene mit allen darunterliegenden Ebenen aus dem DOM-Baum entfernt und eine neue, leere Gruppierung auf der obersten Ebene installiert. Außerdem verfügt die Klasse über eine Methode `setTopLevelGroup`, mit der eine programmatisch erstellte Gruppierung an die Spitze des DOM-Trees gesetzt wird.

Das Zusammenspiel dieser beiden Methoden ermöglicht das folgende Vorgehen: Ein Container ruft `getTopLevelGroup` auf und speichert die Referenz auf die erhaltene Gruppe in einer temporären Variablen. Der Container ruft die Zeichenoperationen seiner Kindknoten auf, die ihre Operationen im Kontext der automatisch neu erstellten Gruppe durchführen. Wenn der Aufruf zurückkehrt, ruft der Container wieder die `getTopLevelMethode` auf und hängt die zurückerhaltene Gruppe auf der zweiten Ebene der gespeicherten Gruppe aus dem ersten Aufruf in den DOM-Baum ein. Danach installiert er den Baum mit der ursprünglichen Gruppe an der Spitze durch Aufruf von `setTopLevelGroup` wieder im `DOMTreeManager`. Wenn seine Kindknoten beim Aufruf ihrer eigenen Zeichenroutinen rekursiv genauso vorgehen, steht am Ende des Aufrufs eine DOM-Baumstruktur aus Gruppierungen, die exakt dem normalen Ablauf der Java-Zeichenaufrufe zwischen Containern und Komponenten entspricht. Insbesondere stehen dabei die Zeichenoperationen von Komponenten innerhalb der Gruppierung ihrer Elterncontainer.

Dadurch steht ein Mechanismus zur Verfügung, mit dem die Zeichenvorgänge einzelner Java-Komponenten im SVG-Code gekappselt bzw. die Zeichenvorgänge von Containern mit denen der zugehörigen Komponenten aggregiert werden und die logischen Strukturen und Hierarchien des Frameworks abgebildet werden können. Indem jedes Element vor dem Einhängen „seiner" Gruppe in die DOM-Struktur dieser Gruppe eine eindeutige ID gibt, kann auf diese Elemente von externen

[7] Die Listings sind stark gekürzt und zeigen nur die hier wesentlichen Teile. Die kompletten Listings liegen der Arbeit im Anhang und in elektronischer Form bei.

Skripten verlässlich zur dynamischen Manipulation durch das DOM zugegriffen werden.
Die in Kapitel 3.1.1 vorgestellten Klassen des Frameworks sind jedoch Komponentenklassen zum Einsatz in einem AWT-Umfeld, ihre `paint`-Methoden berücksichtigen nur „normale“ Grafik-Kontexte ohne die Möglichkeit zur Manipulation von SVG-Code. Um diese Klassen dennoch zur Erzeugung von manipuliertem SVG-Code verwenden zu können, wird jede nicht-abstrakte Komponente und jeder nicht-abstrakte Container von einer Subklasse, deren Namen mit dem Präfix „SVG“ beginnt und die ansonsten wie ihre Oberklasse heißt, erweitert. Diese Klassen übernehmen die Funktionalitäten ihrer Oberklassen weitestgehend. Sie überschreiben allerdings die `paint`-Methode, um das geschilderte Vorgehen umzusetzen. Außerdem überschreiben sie diejenigen Methoden, in denen bisher „normale“ Framework-Klassen benutzt werden und stellen sicher, dass deren Pendants mit der Fähigkeit zum Manipulation des SVG-Codes zur Anwendung kommen.
Damit ist das Framework um die Möglichkeit der Erstellung von SVG-Grafiken mit einem Document Object Model, das semantisch die Elemente von Businessgrafiken wiederspiegelt, ergänzt.

3.2. Serverseitige Generierung

Dieser Teil des Kapitels setzt sich mit der Einbettung des im ersten Kapitelabschnitts vorgestellten Frameworks in serverseitige Komponenten auseinander. Dabei wird am Beispiel einer Komponente zur serverseitigen Generierung dynamischer Inhalte sowie am Beispiel einer Webanwendung gezeigt, wie das Framework zur Erzeugung von Businessgrafiken auf dem Server benutzt werden kann.

3.2.1. Ein Cocoon-Transformer zur Erstellung von SVG-Businessgrafiken

In Kapitel 2.3.2 wurde das Cocoon Framework kurz vorgestellt und es wurde erläutert, dass Cocoon eine Kette von Komponenten verwendet, um Daten über verschiedene Transformationsschritte in das gewünschte Präsentationsformat zu überführen. Dazu werden zwischen einem Generator, der die Ursprungsdaten zur Verfügung stellt, und einem Serializer, der die Aufbereitung für den Empfänger durchführt, 0 oder mehr sogenannte Transformer verwendet, die den ankommenden XML-Strom entgegennehmen und ihn dann in anderes XML-Format überführen, bis das Format den Erwartungen eines Serializers entspricht.
Um das Framework aus dem ersten Kapitelteil auch mit Cocoon-basierten Webanwendungen nutzen zu können, wird ein solcher Transformer benötigt, der XML-Daten entgegen nimmt, daraus mit Hilfe des Frameworks ein Diagramm erstellt und diesen dann in SVG überführt. Da es sich bei SVG um gültiges XML handelt (vgl. Kapitel 2.1), kann das Ergebnis dieser Transformation mit Hilfe des Cocoon-eigenen XML-Serializers an den Client geschickt werden.
Das Eingabeformat für den Transformer wird durch die Document Type Definition aus Listing 12 gegeben. Diese entspricht dem grundlegenden Paradigma des Frameworks, dass ein Diagramm aus Datenreihen besteht, die Daten enthalten.

Listing 12: Eingabedefinition für den Transformer

```
<?xml version="1.0" encoding="ISO8859_15"?>
<!ELEMENT chart (set)+ >
<!ELEMENT set (datum)+>
<!ELEMENT datum (value)>
<!ELEMENT value (#PCDATA)>
<!ATTLIST chart type (pie|column|line) #REQUIRED>
<!ATTLIST set name ID #REQUIRED>
<!ATTLIST datum name CDATA #REQUIRED>
```

Die Komponenten des Cocoon-Frameworks arbeiten mit XML-Strömen; sie verwenden daher zum Einlesen der Daten einen auf der SAX-Technologie (Simple API for XML) basierenden Ansatz. Dabei wird nicht der gesamte Strom geparst, sondern für jedes einzelne XML-Element werden jeweils ein Start-Event und ein End-Event generiert. Bei den dazwischen übergebenen Daten muss eine Komponente entscheiden, ob sie diese sofort bearbeiten und eventuell auf eine Validierung verzichten will oder ob sie diese zur späteren Bearbeitung zwischenspeichert. Im vorliegenden Fall ist die Dokumentstruktur einfach genug, um diese mit geringem Aufwand bezüglich der Zwischenspeicherung von Information validieren zu können. Daher werden Informationen sofort verarbeitet und die Grafik parallel zum Einlesen der Elemente erstellt. Dabei werden die in Tabelle 1 dargestellten Schritte durchgeführt[8]:

Elementname	Ereignis	Aktion
chart	startElement	Lese Diagrammtypen und instanziere ein entsprechendes Diagramm
	characters	Nicht erlaubt, löse Fehler aus
	endElement	Instanziere SVGGraphics2D-Objekt und generiere SVG-Ausgabe
set	startElement	Instanziere neue Datenreihe je nach Diagrammtyp und setze den Namen gemäß Attributwert
	characters	Nicht erlaubt, löse Fehler aus
	endElement	Füge Datenreihe dem Diagramm hinzu
datum	startElement	Instanziere ein Datum-Objekt je nach Diagrammtyp und setze den Namen gemäß Attributwert
	characters	Nicht erlaubt, löse Fehler aus
	endElement	Füge Datum-Objekt der Datenreihe hinzu
value	startElement	Nichts
	characters	Setze den Wert des aktuellen Datum-Objekts
	endElement	Nichts

Tabelle 1: Ablauf bei Events für verschiedene Tags

Das Einlesen sowie die Generierung des SVG-Codes ist, wie Tabelle 1 entnommen werden kann, mit relativ geringem Aufwand verbunden; dies war zu erwarten, da die meiste Arbeit im erarbeiteten Framework sowie in den Klassen des Batik Frameworks erledigt wird. Ungleich aufwändiger ist die

[8] Die Komponente liegt der Arbei in elektronischer Form bei.

Ausgabe für das nächste Element der Cocoon-Pipeline. Da die Daten mit der SAX-Technologie weitergegeben werden sollen, muss für jedes Element der SVG-Ausgabe wieder ein Ereignis erzeugt werden, damit die nächste Komponente die Daten ebenso wie die beschriebene Transformer-Komponente Event-basiert einlesen kann.

Diese Form der Weiterreichung setzt voraus, dass die Daten in einzelnen Knoten vorliegen, deren Namen, Attribute, Textwerte etc. bekannt sind. Dies ist nicht der Fall, wenn sich der Transformer die Daten als String ausgeben lässt. Stattdessen wird nach Beendigung der Transformation von der `SVGGraphics2D`-Instanz der Root-Knoten des DOM-Baums erfragt. Ausgehend von diesem Knoten wird der gesamte Baum durchlaufen und für jeden Knoten werden die entsprechenden SAX-Events erzeugt.

An diesem Beispiel wird sichtbar, was die an der Transformation in SVG beteiligten Frameworks leisten, denn obwohl die Transformation die Hauptaufgabe der Komponente darstellt, besteht der größte Teil der Arbeit in der Generierung von Events für die Weitergabe.

3.2.2. Dynamische Businessgrafiken mit Servlets

Als weiteres Anwendungsbeispiel erläutert dieser Abschnitt eine Webanwendung, bei der die Grafiken dynamisch auf dem Server generiert werden und zusätzlich die dynamischen Eigenschaften des SVG-Formats zur Anwendung kommen.[9]

Bei der Webanwendung handelt es sich um eine auf Java Servlet/JSP Technologie basierende Anwendung (vgl. Kapitel 2.3). Durch Interaktion mit dem Server kann der Benutzer ein Diagramm erstellen. Dazu wählt er auf einer Einstiegsseite den Typ des zu erstellenden Diagramms aus. Auf der nächsten Seite kann er angeben, wie viele Werte er eingeben möchte. Bei Linien- oder Säulendiagrammen kann er dazu eine Matrix aus Zeilen und Spalten angeben, bei einem Tortendiagramm die Anzahl der Elemente insgesamt. Auf einer weiteren Seite wird ihm eine Eingabemaske ähnlich einem Blatt aus einer Tabellenkalkulation gemäß seinen Angaben angeboten; bei einem Tortendiagramm steht nur eine Zeile von Eingabefeldern zur Verfügung. Nach Füllen der Maske mit Werten und Senden der Werte an den Server erhält der Benutzer das nach seinen Angaben mit Hilfe des Frameworks erstellte Diagramm als in die Webseite eingebettete SVG-Datei. Die Daten werden serverseitig zwischengespeichert, durch Klicken eines „Ändern"-Knopfes gelangt der Benutzer wieder in die Eingabemaske, die dann mit diesen Werten vorbelegt ist.

Als konzeptionelle Anwendung zur Demonstration der Interaktionsfähigkeiten von SVG durch Scripting befindet sich innerhalb der Grafik ein Anzeigebereich, in dem der Wert eines Datenelements aus dem Diagramm eingeblendet wird, wenn der Benutzer die Maus darüber bewegt.

Zur Erzielung des Effekts wird auf der Serverseite von einem Servlet zunächst das Diagramm gemäß den Benutzervorgaben mit Hilfe des Frameworks gezeichnet. Datenelemente erhalten dabei die Kennung „element_x", wobei x eine laufende Nummer ist. Die Chart-Komponente wird dann in einen anderen Container eingebettet, die eine von `java.awt.Textfield` abgleitetete Komponente enthält. Danach wird der Container mit den eingebetteten Komponenten an eine Instanz von

---

[9] Die Webanwendung liegt der Arbeit in elektronischer Form bei.

SVGGraphics2D übergeben, wobei das Textfeld im SVG-Code einen leeren text-Knoten mit der ID „Ausgabe“ platziert. Im nächsten Schritt wird der entstandene DOM-Baum nach Elementen mit der ID „element_x“ (siehe oben) abgesucht und die Werte der gefundenen Knoten mit ID gespeichert. Außerdem wird der Event Handler „onmouseover“ für diese Knoten auf einen Funktionsnamen gesetzt. Als letzter Schritt wird dem DOM-Baum ein CDATA-Knoten – also ein Knoten, der nicht XML-Inhalte enthält – mit einem Script hinzugefügt, das die Funktion für die Event Handler sowie ein Array aller gefundenen IDs und der dazugehörigen Werte enthält. Zum Schluss wird nach dem bekannten Verfahren der SVG-Code von der SVGGraphics2D-Instanz geholt und an den Client weitergegeben.

Wenn der Benutzer nun mit der Maus über ein Datenelement im so erstellten Diagramm fährt, wird der Event Handler ausgelöst, die aufgerufene Funktion ermittelt darauhin die ID des Elements, das das Ereignis ausgelöst hat, ermittelt im Array den dazugehörigen Wert und weist diesen als Text dem Knoten „Ausgabe“ zu. Dadurch wird der Wert angezeigt.

Dieses Beispiel zeigt, dass SVG sich hervorragend eignet, serverseitig dynamische Inhalte zusammenzustellen, diese optisch aufbereitet als Bild an den Benutzer zu senden und dennoch die Kontrolle über die Anzeige der Daten zu behalten, indem die Daten dynamisch nur bei Erfüllung bestimmter Voraussetzungen angezeigt werden. Mit den heute üblichen Rastergrafiken sind solche Effekte nur möglich, wenn bei jeder Änderung der Umstände eine erneute Anfrage an den Server gesendet wird. Auch in diesem Sinne darf SVG als „bandbreitenfreundlich“ angesehen werden.

4. Fazit

Die vorliegende Arbeit hat gezeigt, dass SVG als Grafikformat für die Darstellung dynamisch aufbereiteter Information sehr gut geeignet ist. Da es sich bei SVG um einen XML-Dialekt und damit um einen offenen Standard handelt, sind in diesem Umfeld mächtige Werkzeuge entstanden. Namentlich hat das Apache Batik Projekt einen wichtigen Beitrag zur einfachen Handhabung des Formats erbracht, indem es ein Toolkit zur Verfügung stellt, mit dem sich die Erstellung von SVG-Grafiken auf das Zeichnen von Java-Elementen beschränkt. Außergewöhnlich daran ist aber auch, dass durch die Eigenschaften von SVG, insbesondere durch sein Document Object Model, Methoden zur Verfügung stehen, mit denen Änderungen an den Grafiken möglich werden, die gar nicht a priori vom Toolkit unterstützt werden.

Bezogen auf Business Grafiken ist es dadurch möglich, einer eigentlich fertig gezeichneten Grafik nachträglich gewünschte und eventuell notwendige Eigenschaften zu verleihen. Andererseits kann genau dieselbe Schnittstelle benutzt werden, um immer noch auf Zustandsänderungen zu reagieren, wenn die Grafik bereits an den Client ausgeliefert wurde und somit dem Einfluss des Servers eigentlich schon entzogen ist. Insbesondere diese Eigenschaft hebt das SVG-Format deutlich von Rastergrafikformaten ab.

Diese Offenheit und die daraus resultierenden Möglichkeiten werden in absehbarer Zeit dazu führen, dass SVG eine sehr viel dominantere Rolle unter den Grafikformaten einnehmen wird. Da es sich bei SVG auch um XML handelt, ist SVG prädestiniert für den Austausch von Multimediadaten als Ergebnis einer über das Internet erbrachten Dienstleistung, eines sogenannten Webservices. Diese Technologie verwendet im zunehmenden Maße XML für die Verteilung der dabei verwendeten Daten.

Lohnende weitere Schritte könnten daher außer in der Weiterentwicklung des Toolkits zur Unterstützung weiterer Grafiktypen eventuell darin bestehen, in einem ersten Schritt eine verteilte Anwendung zu erstellen, die aus beliebigen Programmen heraus über das Internet aufgerufen wird und Bilddaten als Ergebnis liefert. Während dies in einer ersten Implementierung noch mit proprietären Mitteln geschehen könnte, sollte der schnellen Entwicklung im Bereich XML dadurch Rechnung getragen werden, dass diese verteilte Anwendung in einem weiteren Schritt mit XML-Daten, und in einem dritten Schritt eventuell mit SVG als nativem Format auch für den Transport der Ergebnisse arbeitet.

Literatur

[Andersson et al. 2003] Ola Andersson,Phil Armstrong, Henric Axelsson, Robin Berjon, Benoît Bézaire, John Bowler,Craig Brown, Mike Bultrowicz, Tolga Capin, Milt Capsimalis, Mathias Larsson Carlander, Jakob Cederquist, Charilaos Christopoulos, Richard Cohn, Lee Cole, Don Cone, Alex Danilo, Thomas DeWeese, David Dodds, Andrew Donoho, David Duce, Jerry Evans, Jon Ferraiolo, Darryl Fuller, FUJISAWA Jun, Scott Furman, Brent Getlin, Peter Graffagnino, Rick Graham, Vincent Hardy, HAYAMA Takanari, Lofton Henderson, Jan Christian Herlitz, Alan Hester, Bob Hopgood, ISHIKAWA Masayasu, Dean Jackson, Christophe Jolif, Lee Klosterman, KOBAYASHI Arei, Thierry Kormann, Yuri Khramov, Kelvin Lawrence, Håkon Lie, Chris Lilley, Philip Mansfield, Kevin McCluskey, MINAKUCHI Mitsuru, Luc Minnebo, Tuan Nguyen, ONO Shuichiro, Antoine Quint, SAGARA Takeshi, Troy Sandal, Peter Santangeli, Haroon Sheikh, Brad Sipes, Peter Sorotokin, Gavriel State, Robert Stevahn, Timothy Thompson, UEDA Hirotaka, Rick Yardumian, Charles Ying, Shenxue Zhou: „Scalable Vector Graphics (SVG) 1.1 Specification - W3C Recommendation“, http://www.w3.org/TR/SVG11/REC-SVG11-20030114.pdf, 14. Januar 2003[Ayers et al. 2000] Danny Ayers, Hans Bergsten, Michael Bogovich, Jason Diamond, Matthew Ferris, Marc Fleury, Ari Halberstadt, Paul Houle, Piroz Mohseny, Andrew Patzer, Ron Phillips, Sing Li, Krishna Vedati, Marc Wilcox, Stefan Zeiger: „Professional Java Server Programming“; Wrox Press Ltd., 6th Edition, 2000

[Batik Overview] „Batik SVG Toolkit – Batik Overview“; http://xml.apache.org/batik/, letzter Zugriff am 17.05.2003

[Burke 2001] Eric M. Burke: „Java and XSLT“; O'Reilly and Associates, First Edition, 2001

[Cocoon Home] “cocoon.apache.org”, http://cocoon.apache.org, letzter Zugriff am 6. Juni 2003

[Cocoon Oveview] “Overview of Apache Cocoon”, http://cocoon.apache.org/2.1/overview.html, letzter Zugriff am 6. Juni 2003

[Cocoon Introduction] “Introducing Cocoon”, http://cocoon.apache.org/2.1/introduction.html, letzter Zugriff am 6. Juni 2003[Langham/Ziegeler 2002] Matthew Langham and Carsten Ziegeler: „Cocoon: Building XML Applications“; New Riders Publishing, First Edition, 2002

[Corel Supplier Sample] „Auto Parts“, http://www.corel.com/content/CSGS/demos/auto_demo.html, letzter Zugriff: 26. Mai 2003[Datenreport 2002] Statistisches Bundesamt: „Pressekonferenz Datenreport 2002, Statement von Präsident Johann Hahlen“, Statistisches Bundesamt, Pressestelle, 29. August 2002

[Domain Survey 2002] „Internet Domain Survey Host Count“, Quelle: Internet Software Consortium (http://www.isc.org/ds/hosts.html), letzter Zugriff am 25.05.2003

[Eisenberg 2002] J. David Eisenberg: „Server Side SVG“; http://www.xml.com/lpt/a/2002/02/27/batik/index.html, letzter Zugriff am 27.04.2003

[Jackson 2002] Dean Jackson: „SVG On The Rise“, 6. Juni 2002, http://www.oreillynet.com/lpt/a/2439, letzter Zugriff am 27.04.2003

[Kao 2002] Odej Kao: „Programmiermodelle für verteilte Anwendungen“, Folienskriptum zur Veranstaltung „Verteilte Systeme“ im Wintersemester 2002/2003 an der Universität Paderborn, Kapitel 5

[Le Hors et al. 2000] Arnaud Le Hors,Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, Steve Byrne: „Document Object Model (DOM) Level 2 Core Specification, Version 1.0 - W3C Recommendation“, http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113, 13. November 2000

[Mozilla SVG] „Mozilla SVG Project“, http://www.mozilla.org/projects/svg, Letzter Zugriff am 5. Juni 2003

[Quint 2001] Antoine Quint: „SVG: Where are we now?“; http://xml.com/lpt/a/2001/11/21/svgtools.html, letzter Zugriff am 27.04.2003

[Spona 2001] Helma Spona: „SVG – Webgrafiken mit XML“; Verlag Moderne Industrie Buch AG&Co KG, 1. Auflage, 2001

[St. Laurent et al. 2001] Simon St. Laurent, Joe Johnston, Edd Dumbill: „Programming Web Services with XML-RPC“; O'Reilly and Associates, First Edition, 2001

[Suhl et al. 2001] Prof. Dr. Leena Suhl, Dipl. Wirt.-Inform. Stephan Kassanke, Dipl. Wirt.-Ing. Michael Scholz: „Grundlagen von Web Based Systems”, Universität Paderborn, WS 2001/2002

[SVG History] „W3C Scalable Vector Graphics (SVG) - History“; http://www.w3.org/Graphics/SVG/History.htm8, letzter Zugriff am 10.05.2003

[SVG Implementations] „SVG Implementations“, http://www.w3.org/Graphics/SVG/SVG-Implementations.htm8; letzter Zugriff am 27.04.2003

[SVG Testimonials] „Testimonials for W3C's SVG 1.1 Candidate Recommendation“, http://www.w3.org/2002/04/svg11-testimonial, letzter Zugriff am 09.05.2003

[Tidwell 2001] Doug Tidwell: „XSLT“; O'Reilly and Associates, First Edition, 2001

[W3C SVG Home] „Scalable Vector Graphics (SVG)“; http://www.w3.org/Graphics/SVG/Overview.htm8, letzter Zugriff am 16.05.2003

Anhang A: Listings

Kapitel 2: Beispiel zum Aufbau von SVG

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC '-//W3C//DTD SVG 1.1//EN'
'http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd'>
<svg width="190" height="80" version="1.0" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" font-family="&apos;sansserif&apos;"
stroke="black" stroke-width="1" stroke-dashoffset="0" font-weight="normal"
stroke-opacity="1">
<rect x="10" y="10" width="50" height="50" rx="2" ry="2"
style="stroke:black;stroke-width:1mm;fill:blue;"/>
<circle cx="95" cy="35" r="25" style="stroke:black;stroke-
width:1mm;fill:red;"/>
<path d="M130 60 l50 0 l-25 -50 l-25 50 z" style="stroke:black;stroke-
width:1mm;fill:green;"/>
</svg>
```

Kapitel 2: Transformationsbeispiel

weather.xml

```
<?xml version="1.0" encoding="ISO8859_15"?>
<!DOCTYPE weather [
        <!ELEMENT wheather (maxtemp|mintemp|sky)* >
        <!ELEMENT maxtemp (#PCDATA)>
        <!ELEMENT mintemp (#PCDATA)>
        <!ELEMENT sky EMPTY>
        <!ATTLIST sky appearance (sunny|cloudy|rainy) #REQUIRED>
        <!ENTITY deg "&#186;C">
] >
<weather>
        <sky appearance="sunny" />
        <mintemp>15&deg;</mintemp>
        <maxtemp>38&deg;</maxtemp>
</weather>
```

weather.xsl

```
<?xml version="1.0" encoding="ISO8859_15"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:svg="http://www.w3.org/2000/svg">
<xsl:output encoding="ISO8859_1"/>

<xsl:template match="weather">
 <svg xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" width="200" height="300">
        <style type="text/css">
        <![CDATA[
                .degree {
                        text-anchor: middle;
                        baseline-shift: -6pt;
                        font-family: Verdana sans-serif;
                        font-size:16pt;
                        filter: url(#textshadow);
                }
                ellipse {
                        stroke-width: 1mm;
                        stroke: black;
                        filter: url(#shadow);
                }
        ]]>
        </style>
        <defs>
                <filter id="shadow">
                        <feGaussianBlur result="blur" stdDeviation="2"/>
```

```
                        <feOffset in="SourceGraphic" result="original"/>
                        <feMerge>
                                <feMergeNode in="blur"/>
                                <feMergeNode in="original"/>
                        </feMerge>
                </filter>
                <filter id="textshadow">
                        <feGaussianBlur result="blur" stdDeviation="1"/>
                        <feOffset in="SourceGraphic" result="original"/>
                        <feMerge>
                                <feMergeNode in="blur"/>
                                <feMergeNode in="original"/>
                        </feMerge>
                </filter>
        </defs>

        <xsl:apply-templates />

 </svg>
</xsl:template>

<!-- SKY -->
<xsl:template match="sky">
        <image width="200" height="300">
                <xsl:attribute name="xlink:href"><xsl:value-of
select="@appearance" />.png</xsl:attribute>
        </image>
</xsl:template>


<!-- MAXTEMP -->
<xsl:template match="maxtemp">
        <g id="maxtemp">
                <ellipse style="fill:orange;" ry="25" rx="60" cy="40" cx="75"/>
                <text class="degree" y="40" x="75"><xsl:apply-
templates/></text>
        </g>
</xsl:template>


<!-- MINTEMP -->
<xsl:template match="mintemp">
        <g id="mintemp">
                <ellipse style="fill:blue;" ry="25" rx="60" cy="260" cx="125"/>
                <text x="125" y="260" class="degree"><xsl:apply-templates /
></text>
        </g>
</xsl:template>

</xsl:stylesheet>
```

weather.svg

```
<?xml version="1.0" encoding="US-ASCII"?>
<svg xmlns:svg="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" xmlns="http://www.w3.org/2000/svg"
height="300" width="200"><style type="text/css">

                .degree {
                        text-anchor: middle;
                        baseline-shift: -6pt;
                        font-family: Verdana sans-serif;
                        font-size:16pt;
                        filter: url(#textshadow);
                }
                ellipse {
                        stroke-width: 1mm;
                        stroke: black;
                        filter: url(#shadow);
                }
```

```
        </style><defs><filter id="shadow"><feGaussianBlur stdDeviation="2"
result="blur"/><feOffset result="original"
in="SourceGraphic"/><feMerge><feMergeNode in="blur"/><feMergeNode
in="original"/></feMerge></filter><filter id="textshadow"><feGaussianBlur
stdDeviation="1" result="blur"/><feOffset result="original"
in="SourceGraphic"/><feMerge><feMergeNode in="blur"/><feMergeNode
in="original"/></feMerge></filter></defs>
        <image height="300" width="200" xlink:href="sunny.png"/>
        <g id="mintemp"><ellipse cx="125" cy="260" rx="60" ry="25"
style="fill:blue;"/><text class="degree" y="260" x="125">15&#186;C</text></g>
        <g id="maxtemp"><ellipse cx="75" cy="40" rx="60" ry="25"
style="fill:orange;"/><text x="75" y="40" class="degree">38&#186;C</text></g>
```

Kapitel 3: Batikbeispiel

```
BatikSample.java
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import org.apache.batik.dom.svg.*;
import org.apache.batik.svggen.*;
import org.w3c.dom.*;

public class BatikSample extends Component{

    public Dimension getPreferredSize() {
        return new Dimension(200,200);
    }

    public Dimension getMinimumSize() {
        return getPreferredSize();
    }

    public void paint(Graphics g) {
        g.setColor(Color.blue);
        g.fillRect(1,1, 100, 100);
        g.setColor(Color.red);
        g.fillOval(99,99, 100,100);
    }

    public static void main(String[] args) throws Exception{
        BatikSample batik=new BatikSample();

        final Frame f=new Frame("Batik sample");
        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent event) {
                System.exit(0);
            }
        });
        f.add(batik);
        f.pack();
        f.show();

        DOMImplementation svgDom = SVGDOMImplementation.getDOMImplementation();
        String nameSpace = SVGDOMImplementation.SVG_NAMESPACE_URI;
        Document document = svgDom.createDocument(nameSpace, "svg", null);
        SVGGraphics2D graphics = new SVGGraphics2D(document);
        graphics.setSVGCanvasSize(batik.getPreferredSize());

        batik.paint(graphics);

        PrintWriter writer = new PrintWriter(System.out, true);
        graphics.stream(writer);
    }

}
```

BatikSample_out.svg

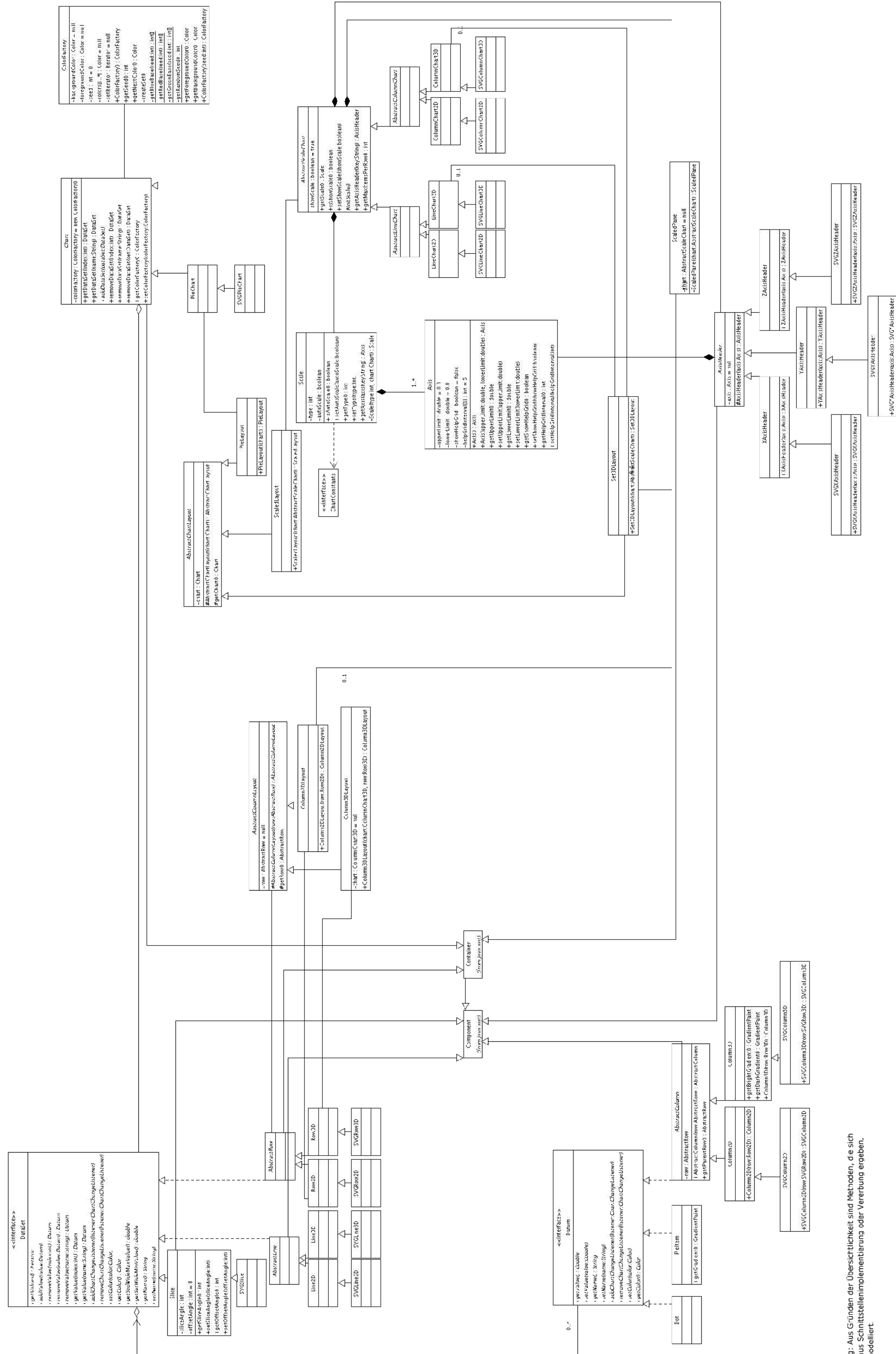
```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC '-//W3C//DTD SVG 1.0//EN' 'http://www.w3.org/TR/2001/REC-
SVG-20010904/DTD/svg10.dtd'>
<svg stroke-dasharray="none" shape-rendering="auto"
xmlns="http://www.w3.org/2000/svg" font-family="&apos;sansserif&apos;"
width="200" text-rendering="auto" fill-opacity="1"
contentScriptType="text/ecmascript" color-rendering="auto" color-
interpolation="auto" preserveAspectRatio="xMidYMid meet" font-size="12"
fill="black" xmlns:xlink="http://www.w3.org/1999/xlink" stroke="black" image-
rendering="auto" stroke-miterlimit="10" zoomAndPan="magnify" version="1.0"
stroke-linecap="square" stroke-linejoin="miter" contentStyleType="text/css"
font-style="normal" height="200" stroke-width="1" stroke-dashoffset="0" font-
weight="normal" stroke-opacity="1">
  <!--Generated by the Batik Graphics2D SVG Generator-->
  <defs id="genericDefs" />
  <g>
    <g fill="blue" stroke="blue">
      <rect width="100" x="1" height="100" y="1" stroke="none" />
      <circle fill="red" r="50" cx="149" cy="149" stroke="none" />
    </g>
  </g>
</svg>
```

Kapitel 3: Cocoon

chart.dtd

```
<?xml version="1.0" encoding="ISO8859_15"?>
<!ELEMENT chart (set)+ >
<!ELEMENT set (datum)+>
<!ELEMENT datum (value)>
<!ELEMENT value (#PCDATA)>
<!ATTLIST chart type (pie|column|line) #REQUIRED>
<!ATTLIST set name ID #REQUIRED>
<!ATTLIST datum name CDATA #REQUIRED>
```

Anhang B: Klassendiagramm



Achtung: Aus Gründen der Übersichtlichkeit sind Methoden, die sich direkt aus Schnittstellenimplementierung oder Vererbung ergeben, nicht modelliert.