

Exploratory Authoring of Interactive Content in a Live Environment

Philipp Otto, Jaqueline Pollak, Daniel Werner,
Felix Wolff, Bastian Steinert, Lauritz Thamsen,
Marcel Taeumel, Jens Lincke, Robert Krahn,
Daniel H. H. Ingalls, Robert Hirschfeld

Technische Berichte Nr. 101

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam



Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Philipp Otto | Jaqueline Pollak | Daniel Werner | Felix Wolff |
Bastian Steinert | Lauritz Thamsen | Marcel Taeumel | Jens Lincke |
Robert Krahn | Daniel H. H. Ingalls | Robert Hirschfeld

Exploratory Authoring of Interactive Content in a Live Environment

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.dnb.de/> abrufbar.

Universitätsverlag Potsdam 2016

<http://verlag.ub.uni-potsdam.de/>

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Das Manuskript ist urheberrechtlich geschützt.

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URN <urn:nbn:de:kobv:517-opus4-83806>
<http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-83806>

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-346-6

Vorwort

Bei der Erstellung von Visualisierungen gibt es im Wesentlichen zwei Ansätze. Zum einen können mit geringem Aufwand schnell Standarddiagramme erstellt werden. Zum anderen gibt es die Möglichkeit, individuelle und interaktive Visualisierungen zu programmieren. Dies ist jedoch mit einem deutlich höheren Aufwand verbunden.

Flower ermöglicht eine schnelle Erstellung individueller und interaktiver Visualisierungen, indem es den Entwicklungsprozess stark vereinfacht und die Nutzer bei den einzelnen Aktivitäten wie dem Import und der Aufbereitung von Daten, deren Abbildung auf visuelle Elemente sowie der Integration von Interaktivität direkt unterstützt.

Juni 2015

Philipp Otto	Bastian Steinert	Robert Krahn
Jaqueline Pollak	Lauritz Thamsen	Daniel H. H. Ingalls
Daniel Werner	Macel Taeumel	Robert Hirschfeld
Felix Wolff	Jens Lincke	

Inhaltsverzeichnis

Vorwort	v
1 Einleitung	1
2 Datenflusskonzept	3
2.1 Einleitung	3
2.2 Datenfluss	6
2.3 Komponenten	8
2.4 Dashboard	21
2.5 Zusammenfassung und Ausblick	24
3 Datenanalyse und -aufbereitung	27
3.1 Einleitung	27
3.2 Daten importieren und konvertieren	32
3.3 Daten erkunden	36
3.4 Daten transformieren	45
3.5 Zusammenfassung und Ausblick	53
4 Abbildung von Daten auf visuelle Elemente	57
4.1 Einleitung	57
4.2 Hintergrund	60
4.3 Abbildung von Daten auf visuelle Elemente	67
4.4 Implementierung	73
4.5 Zusammenfassung und Ausblick	79
5 Interaktionskonzepte	83
5.1 Einleitung und Motivation	83
5.2 Konzeptvorschläge	90
5.3 Implementierung	97
5.4 Ausblick und Diskussion	102
5.5 Zusammenfassung	106

1 Einleitung

Flower ist eine Programmierumgebung zur Erstellung interaktiver Visualisierungen. Es handelt sich um eine Webanwendung, die daher vollständig im Browser läuft. Das Web Application Framework „*Lively Kernel*“ [4] dient als Entwicklungs- und Laufzeitumgebung.

Bisher teilten sich die Tools zur Erstellung von Visualisierungen in zwei verschiedene Gruppen. Auf der einen Seite ist es möglich, schnell aus vorhandenen Daten ein Standarddiagramm zu erstellen. Hier ist der Programmieraufwand gering bis gar nicht vorhanden und nach wenigen Minuten ist eine Grafik fertig. Dieser Weg lässt jedoch kaum Spielraum für Individualität, weshalb die meisten Ergebnisse ähnlich aussehen.

Auf der anderen Seite können mit den bisherigen Tools individuelle und interaktive Visualisierungen angefertigt werden. Dafür muss Programmcode geschrieben werden, weshalb es mitunter Stunden dauert, bis ein Ergebnis vorzuweisen ist. Dieses kann dafür genau an die Vorstellungen des Programmierers angepasst werden. Der Freiheit sind keine Grenzen gesetzt.

Flower vereint diese beiden Lösungen und ermöglicht die schnelle Erstellung individueller, interaktiver Visualisierungen. Es beruht sowohl auf grafischen Ansätzen, bei denen Elemente direkt per Mausinteraktion manipuliert werden, als auch auf textueller Programmierung. Dadurch wird der Programmieraufwand deutlich verringert, ohne jedoch an Ausdruckskraft zu verlieren. Dem zugrunde liegt ein grafisches Datenflusskonzept, in dem einzelne Aufgaben jeweils in Komponenten gekapselt werden.

Flower deckt alle Schritte während der Visualisierungserstellung ab. Diese umfassen: das Analysieren und Aufbereiten der Daten, die Abbildung von Datenwerten auf grafische Elemente sowie die Integration von Interaktivität.

Nachdem im folgenden Kapitel das Konzept des visuellen Datenflusses erläutert wird, orientiert sich der Aufbau der restlichen Arbeit am Ablauf einer Visualisierungserstellung.

2 Datenflusskonzept

2.1 Einleitung

Flower ermöglicht die schnelle Erstellung von interaktiven und individuellen Visualisierungen. Abbildung 2.1 zeigt ein mit *Flower* erstelltes Programm, sowie die daraus resultierende Grafik (unterer Bereich der Abbildung).

Aufbau eines Flower-Programms

Flower verfolgt einen visuellen, datenflussorientierten Programmieransatz. Ein Programm (im Folgenden auch Datenfluss genannt) besteht dabei aus beliebig vielen Komponenten, im Beispielfall vier, die untereinander angeordnet werden. Komponenten sind Operatoren im Datenfluss. Sie erhalten Eingabedaten, prozessieren diese und geben die Ausgabedaten weiter.

Die Ausführungsreihenfolge des Datenflusses ist durch die relative Position der Komponenten zueinander gegeben. Der Datenfluss wird von oben nach unten ausgeführt. Die Verbindungen zwischen den Komponenten sind somit dynamisch und durch gestrichelte Linien gekennzeichnet. Jede Komponente gibt ihre Ausgabedaten an die darunterliegende Komponente weiter, welche diese als Eingabedaten erhält.

Das Endergebnis des Programms ist die fertige Visualisierung, welche im Beispiel in einer *Canvas*-Komponente am Ende des Datenflusses angezeigt wird.

Erstellung einer Visualisierung mit Flower

Um den typischen Arbeitsablauf von *Flower* zu zeigen, wird im Folgenden der Erstellungsprozess der Beispielvisualisierung erläutert. Das vollständige Programm und die resultierende Visualisierung sind in Abbildung 2.1 zu sehen.

Als Erstes müssen die Github Commit Daten in das Programm geladen werden. Dafür wird, über das in Abbildung 2.2 zu sehende Weltkontextmenü, ein *JSONFetcher* erzeugt. Dieser lädt Daten von einer beliebigen URL in das Programm. Um zu überprüfen, ob und welche Daten geladen wurden, wird darunter ein *JSONViewer* erstellt, der seine Eingabedaten anzeigt. Wie in Abbildung 2.3 im *JSONViewer* zu erkennen ist, wurden die Daten der letzten 100 Commits erfolgreich geladen. Jeder Eintrag enthält einen Zeitstempel¹, der benutzt werden kann, um den entsprechenden Wochentag des Commits herauszufinden.

Die Daten werden nun in einem *Script* mittels JavaScript reduziert und umgewandelt. Das Ergebnis ist ein Objekt, das die Commitanzahl pro Wochentag beinhaltet.

¹Zu finden unter `commit.committer.date`.

2 Datenflusskonzept

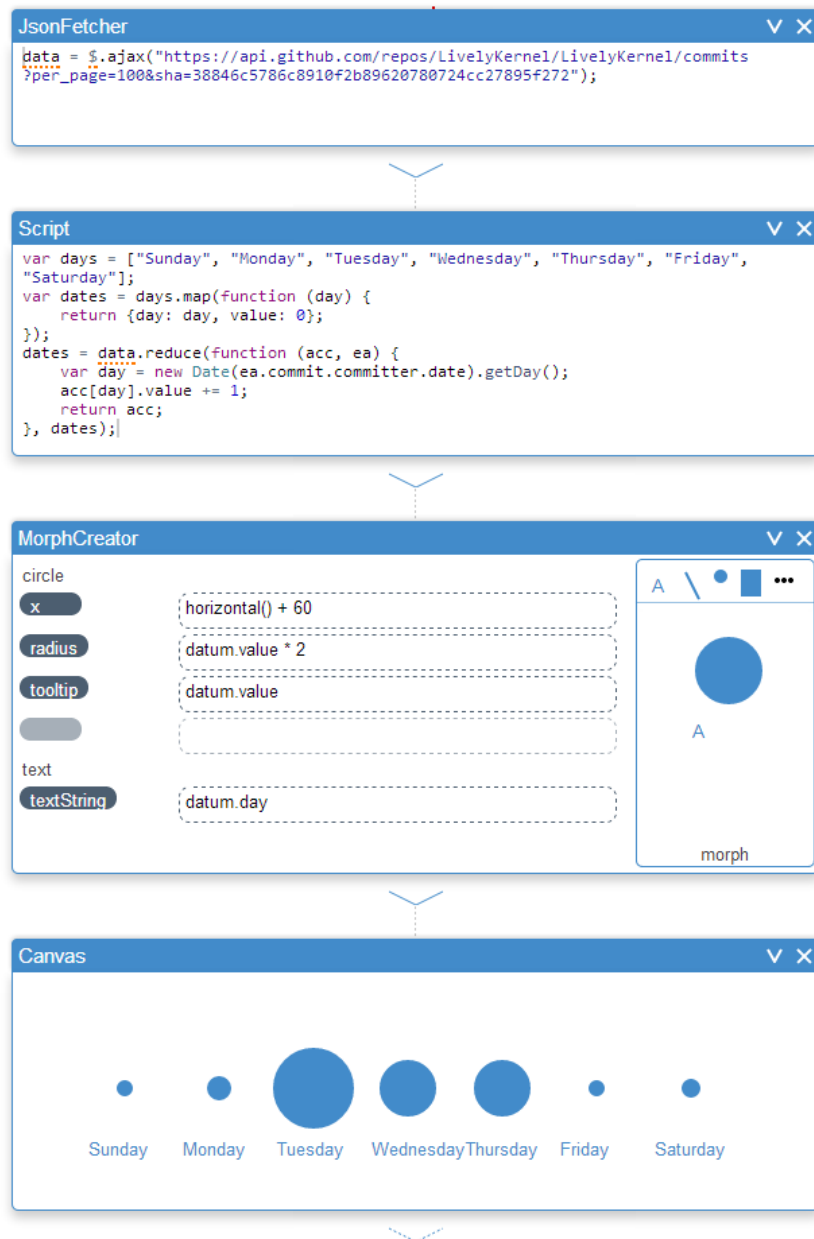


Abbildung 2.1: Komplettes *Flower*-Programm, das eine Visualisierung erstellt, die anhand der Fläche von Kreisen die Menge der Commits pro Wochentag zeigt. Als Vorbild dient das Github Punchcard Diagramm [1]. Berücksichtigt werden dabei die letzten 100 Commits von *Flower* vor dem 4. Juni 2014 [2].

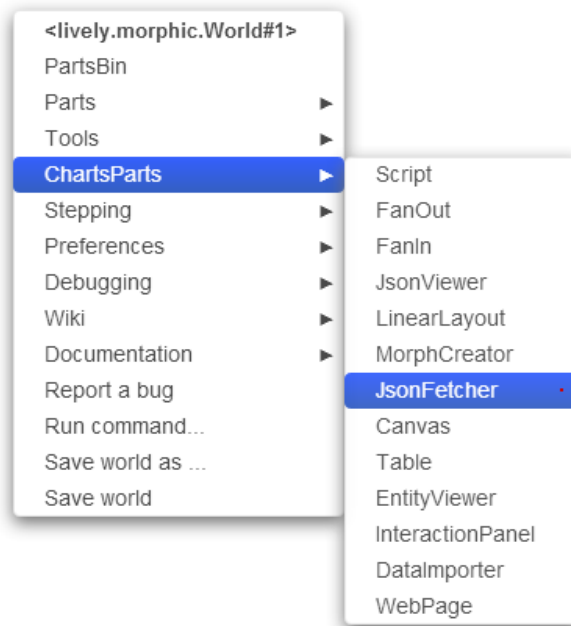


Abbildung 2.2: Erstellung der ersten Komponente über das Kontextmenü der Welt

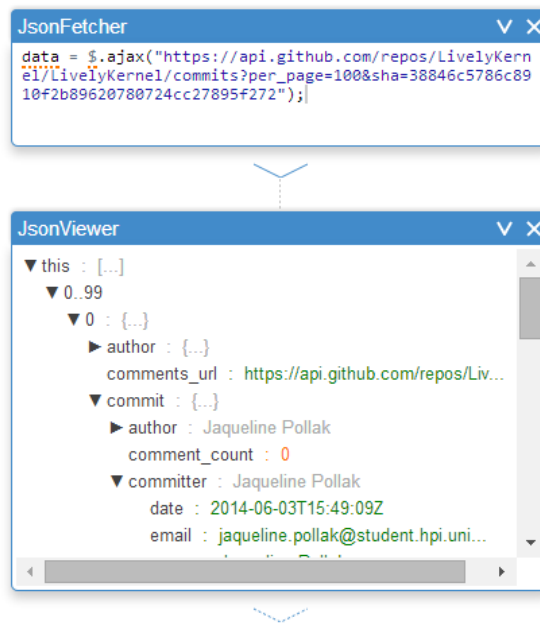


Abbildung 2.3: Laden der Daten

Im nächsten Schritt werden aus den transformierten Daten visuelle Elemente erzeugt, im Beispielfall Kreise. Diese werden mithilfe des sogenannten *MorphCreators* erstellt und nebeneinander angeordnet. Außerdem soll unter jedem Kreis eine Beschriftung mit dem jeweiligen Wochentag angezeigt werden, was ebenfalls mithilfe des *MorphCreators* geschieht.

Im letzten Schritt werden die soeben erstellten visuellen Elemente auf einer Zeichenfläche angezeigt, dem *Canvas*. Damit ist der Erstellungsprozess der Beispielvisualisierung abgeschlossen.

Gliederung des Kapitels In diesem Kapitel wird das zugrunde liegende Datenflusskonzept für die Erstellung von Visualisierungen erläutert. Daraufhin wird der konzeptionelle Aufbau von *Flower* erklärt und auf die Kommunikation der verschiedenen Bestandteile eingegangen. In Abschnitt 2.3 wird der Aufbau und die Funktionalität der Komponenten, den Grundbausteinen von *Flower*, dargelegt. Anschließend wird der Programmierumgebung ein neues Element hinzugefügt – das *Dashboard*. Abgeschlossen wird das Kapitel mit einer Zusammenfassung und Betrachtungen zu verwandten und zukünftigen Arbeiten.

2.2 Datenfluss

Der folgende Abschnitt erläutert vor allem das zugrunde liegende Datenflusskonzept von *Flower*. Er geht außerdem auf die Kommunikation der Komponenten miteinander ein und visualisiert den Ablauf derselben.

2.2.1 Datenflusskonzept

Wie bereits angesprochen, verfolgt *Flower* einen visuellen, datenflussorientierten Programmieransatz. Datenflussorientierte Programmierung ist ein Paradigma, das ein Programm als gerichteten Graph modelliert. Die Knoten eines Datenflussprogramms werden durch Komponenten repräsentiert und die Kanten durch die Verbindungen zwischen ihnen. Daten fließen über die Kanten von Knoten zu Knoten, wobei diese Operationen auf den Daten durchführen. Jeder Knoten im Datenfluss kapselt einen Bearbeitungsschritt, wodurch globaler Zustand vermieden wird. Knoten können verändert, ausgetauscht oder hinzugefügt werden. Ein Datenflussprogramm besteht aus Datenquellen, Knoten die Operationen repräsentieren und Datensinken. *Flower* funktioniert nach dem Push-Modell [3], bei dem Komponenten aktualisiert werden, sobald an einem Eingang Daten vorliegen.

Mit dem Datenflussmodell nutzt *Flower* dessen Ähnlichkeit zur Visualisierungspipeline aus, die als Datentransformationspipeline gesehen werden kann. Eingabedaten werden bearbeitet oder gefiltert und im nächsten Schritt auf visuelle Elemente gemappt. Diese werden wiederum bearbeitet und schlussendlich auf einem *Canvas* angezeigt. Die einzelnen Schritte dieser Pipeline werden in Kapitel 3 – „Datenanalyse und -aufbereitung“ – und Kapitel 4 – „Abbildung von Daten auf visuelle Elemente“ – näher erläutert.

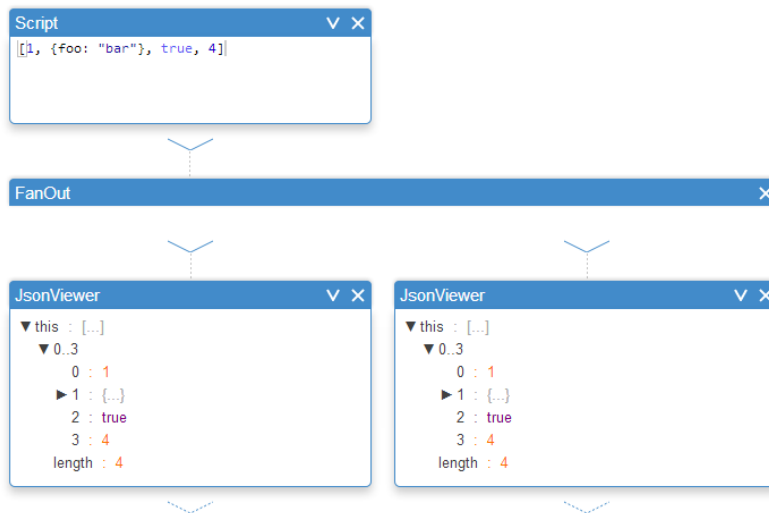


Abbildung 2.4: Die *FanOut* Komponente teilt den Datenfluss in mehrere Stränge auf.

Fans

FanOuts und *FanIns* sind spezielle Komponenten, die den Datenfluss in mehrere Stränge aufteilen beziehungsweise mehrere Stränge zusammenfügen können. Die Aufteilung des Datenflusses, wie das *FanOut* sie in Abbildung 2.4 vornimmt, ist dabei programmieretechnisch als Nacheinanderausführung mehrerer Programmstränge und nicht als parallele Ausführung² oder *if-else*-Konstrukt zu sehen. Sie ist eine Möglichkeit, sein Programm zu strukturieren und in logische Einheiten aufzuteilen. Das Zusammenfügen ist zum einen nach einer Auftrennung des Datenflusses nötig und wird zum anderen benutzt, wenn Daten aus mehreren Datenquellen geladen werden sollen.

2.2.2 Komponentenkommunikation

Wie bereits im Beispiel in Abschnitt 2.1 angemerkt, ist die Ausführungsreihenfolge des Datenflusses durch die relative Position der Komponenten zueinander gegeben. Ändert der Nutzer eine Komponente, setzt das eine Kettenreaktion in Gang, in der alle folgenden Komponenten ebenfalls aktualisiert werden. Der Ablauf der Kommunikation, der diese Propagation ermöglicht, ist in Abbildung 2.5 zu erkennen und im Folgenden aufgeschlüsselt:

1. Wird eine Komponente verändert, reevaluiert sie ihre Daten (1).
2. Sie sucht die direkt unter ihr liegende Komponente und benachrichtigt diese über die Veränderung (2).

²JavaScript ist single-threaded und erlaubt keine nebenläufige Ausführung.

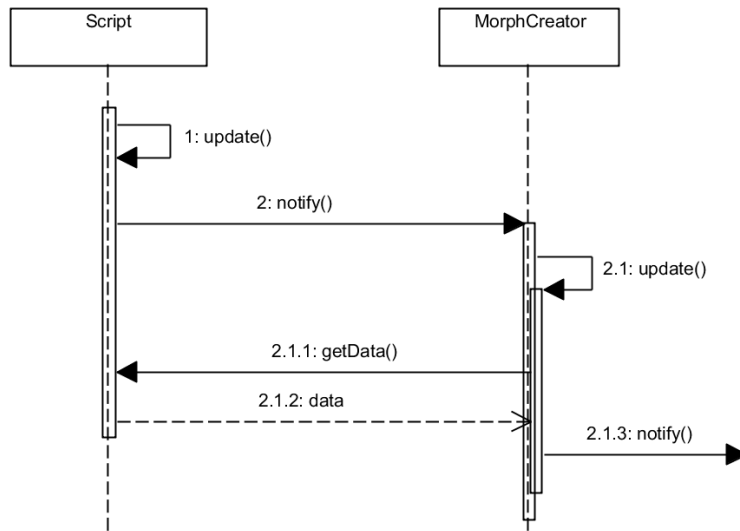


Abbildung 2.5: Sequenzdiagramm, das den Ablauf der Kommunikation zwischen zwei Komponenten zeigt.

3. Diese wird ebenfalls aktualisiert (2.1), wofür mittels `getData` die Eingangsdaten erneuert werden (2.1.1/2). Dies ist vor allem für das FanIn wichtig, da es Daten aus mehreren Quellen erhält.
4. Die nächste Komponente wird benachrichtigt (2.1.3).

So wird die Veränderung durch den gesamten Datenfluss propagiert, bis keine darunterliegende Komponente mehr gefunden wird. Ändert der Nutzer eine Komponente im Datenfluss, erhält er unmittelbares Feedback und sieht sofort das veränderte Ergebnis. Es ist keine manuelle Neuausführung des Datenflusses notwendig.

2.3 Komponenten

Komponenten sind die Operatoren im Datenfluss, die Daten als Eingabe erhalten, diese gegebenenfalls bearbeiten und anschließend weitergeben. Jedes Datenflussprogramm besteht aus mindestens einer Komponente, sie sind dessen Grundbausteine. Dieser Abschnitt geht zuerst auf den Aufbau derselben ein und widmet sich anschließend ihrem Layouting. Abschließend wird eine Möglichkeit vorgestellt, Teile eines Datenflusses wiederzuverwenden, und eine Übersicht über alle Komponententypen gegeben.

2.3.1 Aufbau

Alle Komponenten haben die in Abbildung 2.6 erkennbaren folgenden Gemeinsamkeiten:

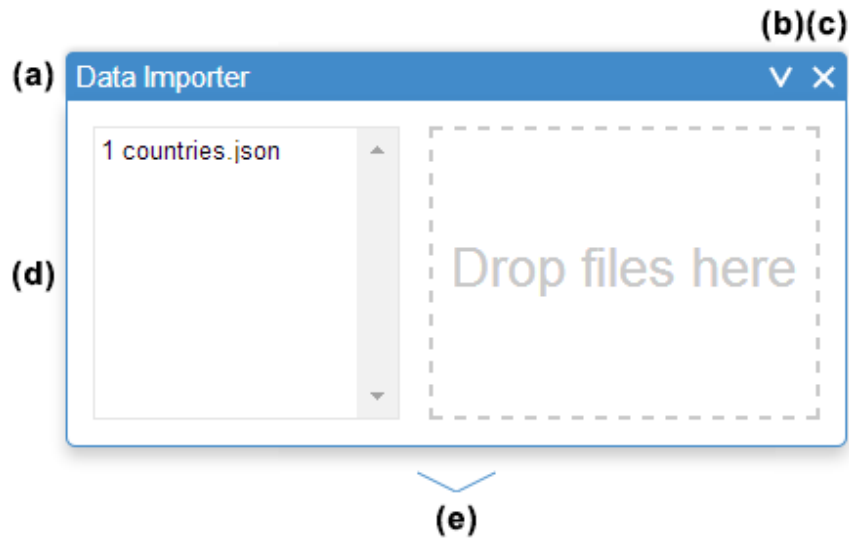


Abbildung 2.6: Komponente mit Datenflusscontainer

1. Komponentenheader mit Titel, der beim Klick die Komponente minimiert
2. Schaltfläche, um den Komponententyp zu wechseln
3. Schaltfläche, die die Komponente entfernt
4. Komponentenkörper mit Inhalt
5. Pfeil, um die Weitergabe von Daten an- und auszuschalten

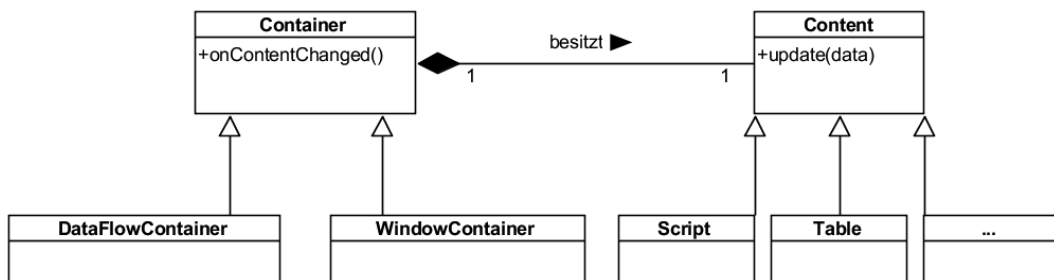


Abbildung 2.7: Klassendiagramm, das den Aufbau einer Komponente aus Container und Inhalt (Content) zeigt.

Implementierung

Abbildung 2.7 zeigt die Klassenstruktur der Komponenten, die eine Komposition aus Container und Inhalt sind. Dieses Design ermöglicht es, verschiedene

Container- und Inhaltstypen zu erstellen und diese beliebig miteinander zu kombinieren.

Der Container einer Komponente legt fest, wie die Komponente auf Einflüsse von außen reagiert. Dies umfasst die Benachrichtigung über neue Eingangsdaten und die Aufforderung sich dem Layout anzupassen. Der Inhalt legt fest, wie die Komponente Daten prozessiert und wie sie sich visuell präsentiert.

Es gibt in *Flower* zwei verschiedene Containertypen:

- den *Datenflusscontainer* (*DataFlowContainer*)
- den *Fenstercontainer* (*WindowContainer*)

Der *Datenflusscontainer*, der in Abbildung 2.6 zu sehen ist, implementiert Funktionalität zur Weitergabe der Daten und wird in das Layouting einbezogen (siehe Abschnitt 2.3.2).



Abbildung 2.8: Komponente mit Fenstercontainer

Der *Fenstercontainer*, der in Abbildung 2.8 zu sehen ist, bleibt davon unbeeinflusst und verändert auch selbst den Datenfluss nicht. Er dient zum Beispiel zur Anzeige eines temporären *JSONViewers*, der, wie in Abbildung 2.9 zu sehen, über dem Datenfluss liegt. Mithilfe des *JSONViewers* kann überprüft werden, welche Daten zwischen zwei Komponenten weitergereicht werden. Da der Container nicht am Datenfluss teilnimmt, besitzt er keinen Pfeil.

Eine Übersicht über die verschiedenen Inhaltstypen ist in Abschnitt 2.3.4 zu finden.

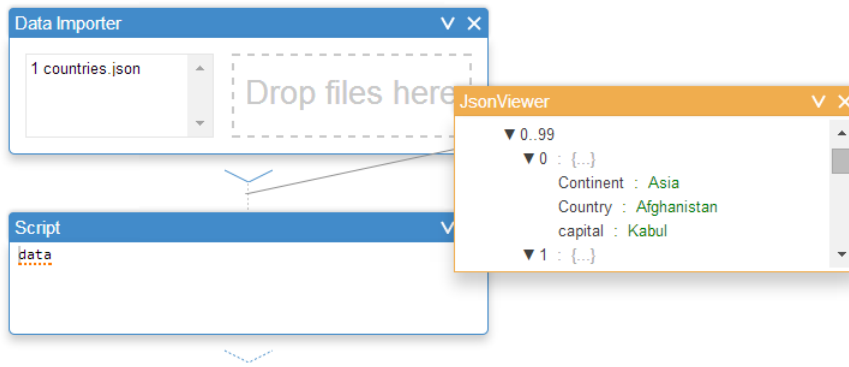


Abbildung 2.9: Ein temporärer *JSONViewer* mit Fenstercontainer wird zur Kontrolle der weitergereichten Daten eingesetzt.

Kommunikation zwischen Container und Inhalt

Um die Kopplung zwischen Container und Inhalt möglichst gering zu halten, existiert ein auf Abbildung 2.7 erkennbares, minimalistisches Interface zwischen beiden Teilen. Dem Inhalt wird über die `update`-Methode mitgeteilt, dass sich die Eingangsdaten geändert haben. Als Rückgabewert werden die prozessierten Daten erwartet. Ändert sich der Inhalt, wie zum Beispiel beim Eintippen von Code im *Script*, ruft dieser die `onContentChanged`-Methode der Komponente auf. `onContentChanged` propagiert diese Veränderung dann gegebenenfalls.

Austausch des Komponenteninhalts

Die strikte Trennung zwischen Container und Inhalt bietet einen weiteren Vorteil. Sie erlaubt es, den Typ einer Komponente zu ändern, nachdem sie erstellt wurde und Teil des Datenflusses ist. Über die Schaltfläche (b), die in Abbildung 2.6 zu erkennen ist, kann der Nutzer aus allen verfügbaren Komponententypen wählen. So kann beispielsweise ein *JSONViewer* schnell und unkompliziert in eine *Table* umgewandelt werden.

Wie dem Quelltext 2.1 zu entnehmen ist, müssen dazu programmatisch folgende Schritte durchgeführt werden.

1. Der neue Inhalt wird erstellt und der alte entfernt. (Zeilen 2–8)
2. Die neuen Referenzen in Inhalt und Komponente werden gesetzt. (9–10)
3. Der neue Inhalt wird hinzugefügt und der Titel der Komponente geändert. (14–17)

2.3.2 Layouting

Der visuelle Programmierstil von *Flower* in Verbindung mit der losen Bindung der Komponenten bietet mehrere Vorteile. Zum einen können Zwischenkomponenten eingefügt werden, ohne explizit neue Verbindungen setzen zu müssen. Zum

Quelltext 2.1: Auswechseln des Inhalts einer Komponente

```
1 swapContent: function(newContentName) {
2     var newContent = new lively.morphic.Charts[newContentName]();
3
4     newContent.setExtent(this.content.getExtent());
5     newContent.setPosition(this.content.getPosition());
6
7     // - this - is the current component
8     this.content.remove();
9     this.content = newContent;
10    this.content.component = this;
11
12    // ...
13
14    this.componentBody.addMorph(newContent);
15    this.content.update(this.data);
16
17    this.setDescription(this.content.description);
18 }
```

anderen kann die Anordnung von Komponenten im Datenfluss unkompliziert verändert werden, da keine Verbindungen zwischen den Komponenten bestehen, die erst noch gelöst werden müssten. Das Verschieben von Komponenten hat direkten Einfluss auf die Ausführungsreihenfolge beziehungsweise das Ergebnis der Datenflussauswertung. Bei der Entwicklung von *Flower* wurde viel Wert darauf gelegt, die Anordnung von Komponenten so einfach und intuitiv wie möglich zu gestalten.

Die Komponenten implementieren einen Layoutalgorithmus, der auf folgende Aktionen reagiert (auch *Layouttrigger* genannt):

- Verschieben
- Löschen
- Ändern der Größe
- Minimieren

Beim Verschieben von Komponenten wird ein in Abbildung 2.10 ersichtlicher Platzhalter angezeigt. Dieser nimmt die Form eines gestrichelten Rahmens an, der die Größe der Komponente hat. Der Platzhalter wird an der Stelle angezeigt, an der sich die Komponente in den Datenfluss einfügen würde, wenn sie in diesem Moment losgelassen werden würde. Andere Komponenten im Datenfluss rücken zur Seite und machen Platz für die umhergezogene Komponente. Wird eine Komponente minimiert oder aus dem Datenfluss entfernt, wird die entstandene Lücke von unten aufgefüllt, indem die Komponenten darunter so weit wie möglich hochrücken.

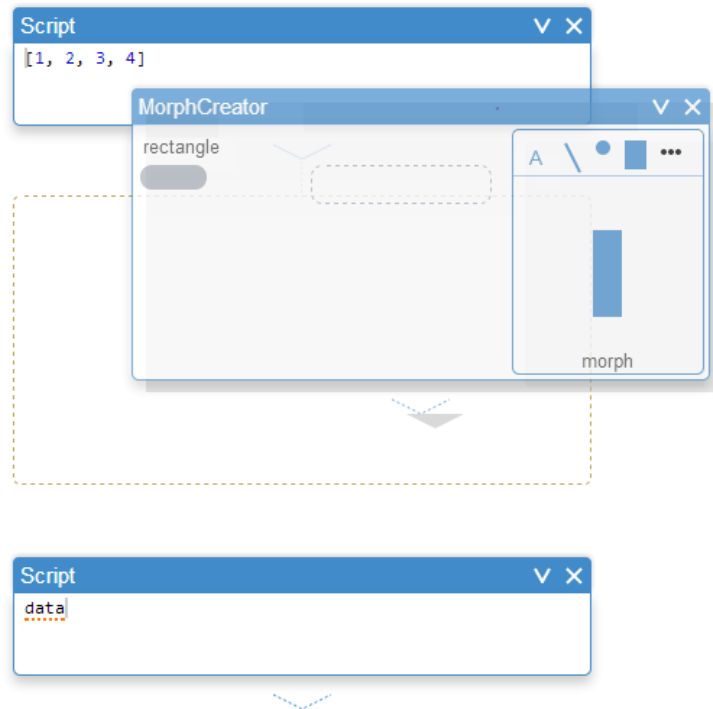


Abbildung 2.10: Komponenten zeigen beim Verschieben einen Platzhalter an.

Implementierung

Bei diesem automatischen Layouting muss beachtet werden, dass die relative Position der Komponenten zueinander Bestandteil des Programms ist und nicht ohne Weiteres verändert werden darf. Alle Verschiebungen von Komponenten im Datenfluss, deren Ursache das Layouting ist, müssen deshalb die Reihenfolge und Struktur des ursprünglichen Datenflusses erhalten. Immer wenn ein *Layouttrigger* auftritt, entsteht im Datenfluss entweder eine Lücke (zum Beispiel beim Löschen) oder es wird mehr Platz benötigt (zum Beispiel beim Vergrößern). Dies lässt sich in eine Distanz umrechnen, um die sich die nachfolgenden Komponenten verschieben müssen. Die Distanz ist negativ, wenn eine Lücke entstanden ist, beziehungsweise positiv, wenn mehr Platz benötigt wird. Sie wird anschließend durch den Datenfluss propagiert, wobei jede Komponente für sich entscheidet, wie sie darauf reagiert.

Bei der Implementierung des Layoutalgorithmus wurde eine objektorientierte Variante gewählt. Dies ermöglicht es, dass ein *FanOut* anders auf die Bewegungsaufforderung reagiert als ein *Script*. In Abbildung 2.11 ist zu erkennen, dass das *FanOut*, dadurch dass es breiter ist als andere Komponenten, gesonderte Logik für das Layouting implementieren muss. Wird das mittlere *Script* entfernt, kann das *FanOut* nicht wie andere Komponenten bis an den *DataImporter* heran hoch verschoben werden, weil noch andere Komponenten im Weg liegen (siehe Abbildung 2.11).

2 Datenflusskonzept

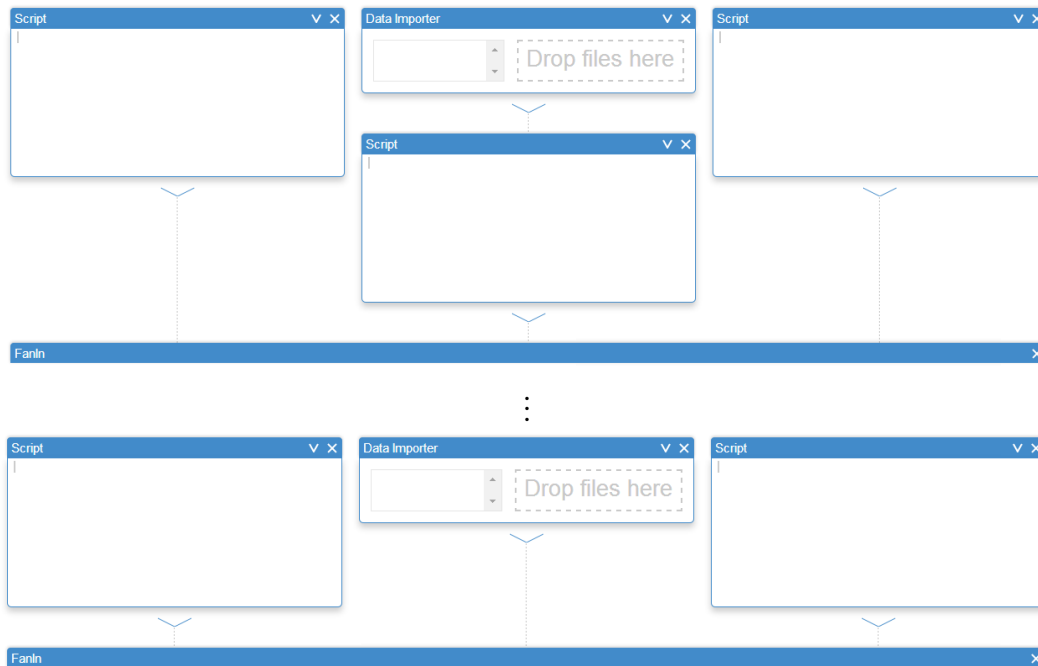


Abbildung 2.11: *FanIn* vor und nach dem Entfernen der mittleren *Script* Komponente. Das *FanIn* wird nach dem Entfernen des *Scripts* nicht bis an den *DataImporter* heran verschoben.

Im Folgenden wird auf den Ablauf des Verschiebens einer Komponente eingegangen. Quellcodeauszüge zeigen, wie in diesem Fall das Layouting durchgeführt wird.

Komponenten lassen sich mittels einer Drag-and-Drop Geste verschieben. Diese besteht technisch gesehen aus drei einzelnen Events:

1. dem Beginn der Drag-Geste: `onDragStart`,
2. dem Draggen an sich:³ `onDrag`,
3. dem Beenden der Drag-Geste: `onDragEnd` und `onDropOn`.

In `onDragStart` (siehe Quelltext 2.2) wird die Komponente im Datenfluss durch den Platzhalter ersetzt und die aktuelle Position aller Komponenten des Datenflusses zwischengespeichert. Dies ermöglicht es, nach jedem drag-Schritt den Ursprungszustand wiederherzustellen, was den Algorithmus erheblich vereinfacht.

In `onDrag` (siehe Quelltext 2.3) wird zuerst die Größe des Platzhalters berechnet sowie dessen Position auf die aktuelle Mausposition gesetzt. Anschließend wird `realignAllComponents` aufgerufen, wo das tatsächliche Layouting stattfindet. In

³Das drag-Event wird während der Bewegung mehrmals getriggert. Wie oft genau, hängt vom Mausmodell des Benutzers ab.

Quelltext 2.2: Speichern des Status Quo für das Layouting in onDragStart

```

1 // take the dragged component out of the layout
2 var componentsBelow = this.getComponentsInDirection(this.BELOW);
3 componentsBelow.each(function (c) {
4     c.move(-_this.getExtent().y - _this.componentOffset);
5 });
6 // save cached position for the automatic layouting
7 lively.morphic.Charts.DataFlowComponent.getAllComponents().each(function
    (ea) {
8     ea.cachedPosition = ea.getPosition();
9 });

```

Quelltext 2.3: Verschieben des Platzhalters in onDrag

```

1 onDrag: function($super) {
2     $super();
3     var previewMorph = $morph("PreviewMorph" + this);
4     var previewPos = this.calculateSnappingPosition();
5     var previewExtent = this.calculateSnappingExtent();
6     previewMorph.setPosition(previewPos);
7     previewMorph.setExtent(previewExtent);
8
9     this.realignAllComponents();
10 }

```

Quelltext 2.4: Verschieben von Komponenten, um den Platzhalter einzufügen

```
1 realignAllComponents : function() {
2     var previewMorph = $morph("PreviewMorph" + this);
3
4     // reset the position of all components
5     var all = lively.morphic.Charts.DataFlowComponent.getAllComponents();
6     all.each(function (ea) {
7         ea.setPosition(ea.getCachedPosition());
8     })
9
10    var componentsBelow = this.getComponentsInDirection(this.BELOW);
11    var componentsAbove = this.getComponentsInDirection(this.ABOVE);
12    var componentsToMove = [];
13
14    // select all components which the preview intersects
15    // this could also possibly be one of componentsAbove, so check them
16    // as well
17    componentsToMove = componentsAbove.concat(componentsBelow).filter(
18        function (c) {
19            return (c && previewMorph.globalBounds().intersects(
20                c.innerBounds().translatedBy(c.getPosition())
21            ));
22        });
23    /* actually move the components */
24 }
```

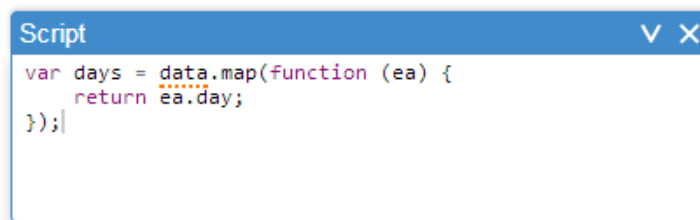
`realignAllComponents` wird anfänglich die Position aller Komponenten auf deren zwischengespeicherte Ursprungsposition zurückgesetzt, um den Status Quo wiederherzustellen. Anschließend werden mittels `GetComponentInDirection` alle umliegenden Komponenten bestimmt. Danach wird mit Hilfe eines Überschneidungstests zwischen dem Platzhalter und der jeweiligen Komponente bestimmt, welche Komponenten im Weg liegen. Diesen wird im letzten Schritt durch den Aufruf der `move`-Methode mitgeteilt, sich wegzubewegen.

In `onDragEnd` und `onDropOn` wird der Platzhalter entfernt, sowie den umliegenden Komponenten über das Ende des Verschiebevorgangs Bescheid gegeben.

2.3.3 Diagramme als Part

Flower bietet die Möglichkeit Datenflussteile unter Nutzern zu teilen und wiederzuverwenden. Komponenten können selektiert und daraufhin als Datenfluss im `PartsBin` gespeichert werden. Der gesicherte Datenfluss, der zum Beispiel ein Balkendiagramm erstellt, kann dann als Teil einer komplexeren Visualisierung wiederverwendet werden. Weiterhin kann dieser auch als Vorlage verwendet werden, die den eigenen Wünschen nach angepasst werden kann. Der `PartsBin` [5] ist ein Lively-Tool, welches darauf ausgelegt ist, Lively-Objekte unter Nutzern zu teilen. Die Möglichkeit, dort auch ganze Datenflüsse abzuspeichern, verstärkt deren Wiederverwendbarkeit und ermöglicht MashUps [6] aus bereits bestehenden Programmteilen. Das Konzept sollte weiter angepasst werden, sodass der Datenfluss nicht in seiner kompletten Größe wieder eingefügt wird, sondern gekapselt als eigene Komponente, siehe dazu die Betrachtungen im Abschnitt 2.5.3 *Zukünftige Arbeiten*.

2.3.4 Komponentenübersicht



```
Script
var days = data.map(function (ea) {
  return ea.day;
});
```

Abbildung 2.12: *Script*

Das **Script** enthält einen Code-Editor und ist in der Lage beliebigen JavaScript-Code auszuführen. Der Editor unterstützt die in Lively übliche Evaluierung von

beliebigem Quellcode⁴. Das *Script* stellt die erhaltenen Eingabedaten als lokale Variable `data` bereit. Gibt der Nutzer Code ein, führt das zur automatischen Neu-evaluierung des Datenflusses. Diese Neuevaluierung ist *debounced*⁵, um eine ruckelfreie Eingabe zu ermöglichen. Das automatische Ausführen des Codes kann mittels der Esc-Taste an- und ausgeschaltet werden. Das *Script* gibt den Wert der letzten JavaScript-Expression als Ausgabedaten weiter.

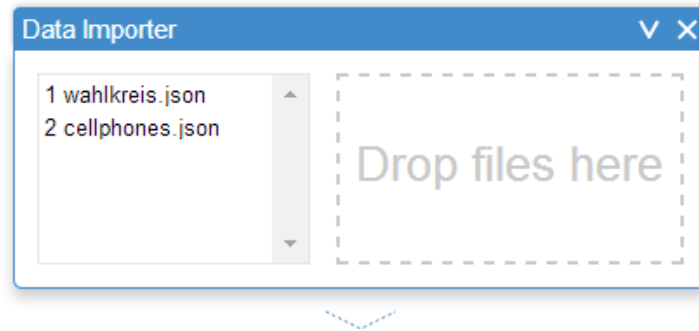


Abbildung 2.13: *DataImporter*

Der **DataImporter** lädt Daten vom lokalen Dateisystem, aus dem Internet oder der Zwischenablage in das Programm (siehe *DataImporter* in Abschnitt 3.2.2, Seite 35). Er besteht aus einem Ablage-Bereich, der mittels Drag-and-Drop Gesten mit Daten-URLs, Dateien und ausgeschnittenen Daten befüllt werden kann. Der *DataImporter* erkennt daraufhin automatisch das Format der Eingabedaten, fragt diese an, falls es sich um eine URL handelt, und wandelt sie in das JSON-Format [9] um. Eine Übersicht über alle Eingabedaten wird im linken Teil der Komponente angezeigt. Die Ausgabedaten bestehen aus einem Array, dessen Elemente die jeweils umgewandelten JSON-Daten sind.

Der **MorphCreator** bildet Daten auf visuelle Elemente ab (siehe *Abbildung von Daten auf visuelle Elemente* in Abschnitt 4.3, Seite 67). Er enthält einen Beispielmorph, der als Prototyp für die zu erstellenden visuellen Elemente dient. Die Komponente definiert außerdem ein deklaratives Mapping von Datenelementen oder beliebigen JavaScript-Expressions auf Eigenschaften des Morphs. Sie führt ein `map` auf den Eingabedaten aus, wobei für jedes Datum ein Morph nach Vorbild des Prototypmorphs erstellt wird. Es werden außerdem alle deklarierten Mappings auf den Morph angewendet. Die Ausgabedaten bestehen aus einem Array mit den erstellten Morphs.

⁴Unter Windows beispielsweise mittels Ctrl+P und Ctrl+D.

⁵Siehe <http://underscorejs.org/#debounce>; besucht am 01. Oktober 2014.

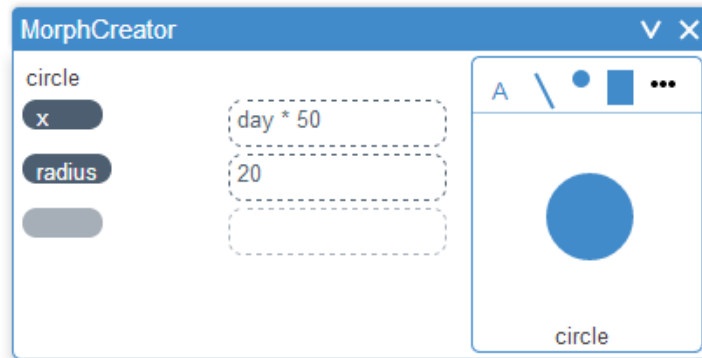


Abbildung 2.14: *MorphCreator*

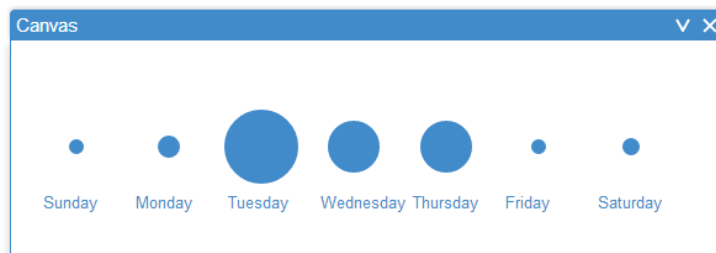


Abbildung 2.15: *Canvas*

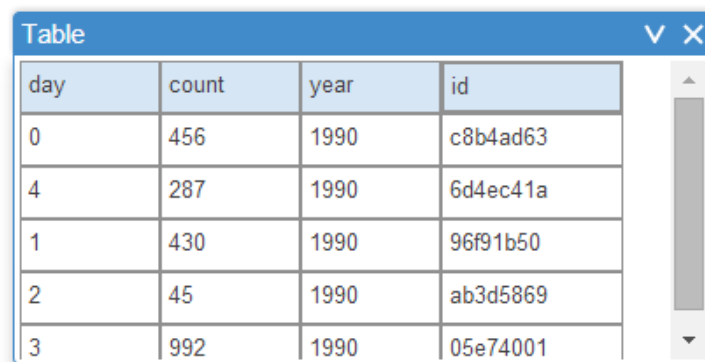
2 Datenflusskonzept

Das **Canvas** dient zur Anzeige von Morphs. Sowohl Eingabe- als auch Ausgabedaten bestehen aus einem Array von Morphs, das durch das Canvas nicht verändert wird.



Abbildung 2.16: *JSONViewer*

Der **JSONViewer** zeigt beliebige JavaScript-Objekte in einer baumartigen Liste an. Baumknoten lassen sich beliebig ein- und ausblenden. Eigenschaften des Prototyps werden nicht angezeigt. Der *JSONViewer* verändert die Eingabedaten nicht.



The screenshot shows a window titled "Table" displaying a table with four columns: "day", "count", "year", and "id". The table contains five rows of data.

day	count	year	id
0	456	1990	c8b4ad63
4	287	1990	6d4ec41a
1	430	1990	96f91b50
2	45	1990	ab3d5869
3	992	1990	05e74001

Abbildung 2.17: *Table*

Die **Table** zeigt JavaScript-Objekte in einer zweidimensionalen Tabelle an. Sie lässt sich spaltenweise sortieren. Die *Table* bietet über das Kontextmenü die Möglichkeit, spaltenweise aggregierte Werte anzuzeigen. Dazu gehören zum Beispiel

das Minimum, das Maximum und der Durchschnitt (siehe *Deskriptive Statistiken* in Abschnitt 3.3.3, Seite 38). Spalten lassen sich umbenennen und löschen. Die *Table* verändert die Eingabedaten nicht.



Abbildung 2.18: *FanOut*

Das **FanOut** teilt den Datenfluss in mehrere Stränge auf, wobei jeder Strang die duplizierten Eingabedaten erhält.

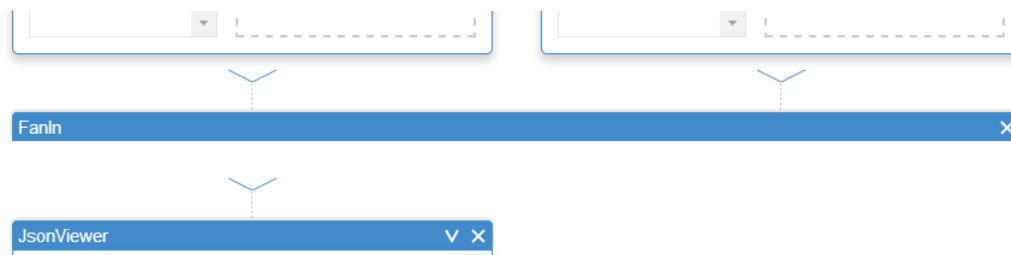


Abbildung 2.19: *FanIn*

Das **FanIn** führt mehrere Stränge des Datenflusses wieder zusammen. Das Ausgabedatum ist ein Array, dessen Elemente die jeweiligen Eingabedaten der verschiedenen Stränge sind.

Sowohl *FanIn* als auch *FanOut* bestehen visuell nur aus dem Komponentenheader, der Inhalt ist nicht sichtbar, weil er keinen Mehrwert bietet.

2.4 Dashboard

2.4.1 Motivation

Im Laufe des Bachelorprojekts wurden viele Visualisierungen mit *Flower* erstellt. Besonders bei komplexen Visualisierungen, kann es passieren, dass der Nutzer nicht alle Komponenten gleichzeitig im Blick hat. Bearbeitet der Nutzer eine Komponente im oberen Bereich der Lively-Welt, sind etwaige Auswirkungen nicht sofort

2 Datenflusskonzept

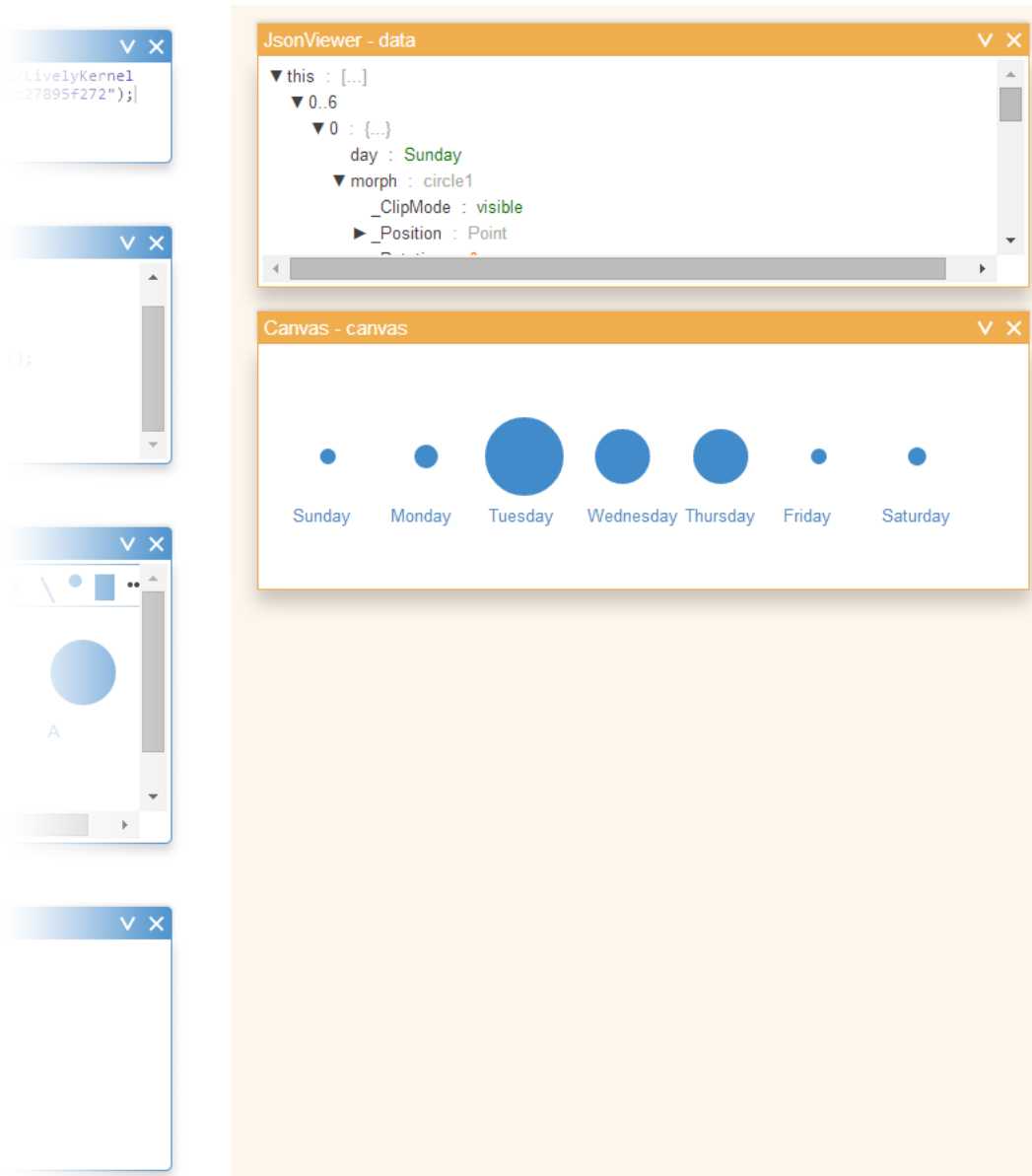


Abbildung 2.20: *Dashboard*, das einen *JSONViewer* und ein *Canvas* anzeigt

erkennbar, weil das *Canvas* nicht im sichtbaren Bereich der Welt liegt. Dadurch gehen die Vorteile des unmittelbaren Feedbacks verloren, denn das Ergebnis von Änderungen ist erst nach dem Scrollen ersichtlich.

Flower führt deswegen ein weiteres Benutzeroberflächenelement ein; das in Abbildung 2.20 sichtbare, leicht orange unterlegte *Dashboard*. Das *Dashboard* ist ein abgegrenzter Bereich auf der rechten Seite des Bildschirms, dessen Position sich beim Scrollen nicht verändert. Es ist somit immer sichtbar und verschwindet nie aus dem Blickfeld des Nutzers. Die Position wurde an den rechten Bildschirmrand gelegt, da ein Datenflussprogramm meist höher als breit ist. Die Breite des *Dashboard*s kann durch den Nutzer verändert werden, falls es den Datenfluss überdecken sollte. Das *Canvas*, auf dem die Visualisierung angezeigt werden soll, wird nun im *Dashboard* platziert und erlaubt es dem Nutzer, das Ergebnis seines Programms ständig im Blick zu haben.

Das *Dashboard* kann außerdem beliebig viele weitere Viewer anzeigen, zum Beispiel einen *JSONViewer*, der immer die aktuellen Eingabedaten anzeigt. Es bietet des Weiteren Platz für das *InteractionPanel*, welches den interaktiven Erstellungsprozess der Visualisierung unterstützt (siehe „Datenabhängige Interaktionen“ in Abschnitt 5.2.1, Seite 91). Um dem *Dashboard* beliebige Viewer hinzuzufügen, wird in jeder Komponente eine Variable namens `env`⁶ zugänglich gemacht. `env` ist ein JavaScript-Objekt, wobei dem *Dashboard* für jeden Objektschlüssel ein Viewer hinzugefügt wird. Jeder Viewer zeigt jeweils die ihm zugeordneten Daten an. Fügt der Nutzer ein neues Attribut zu `env` hinzu und weist diesem Daten zu, so wird im *Dashboard* ein weiterer Viewer hinzugefügt. Über die Codezeile:

```
env.canvas = [morph1, morph2, ..., morphX];
```

wird beispielsweise ein Canvas zum *Dashboard* hinzugefügt, das `morph1` bis `morphX` anzeigt.

Das *Dashboard* fungiert als Ergänzung zum Datenflusskonzept und bietet eine ständig sichtbare Übersicht über beliebige Inhalte des aktiven Programms.

2.4.2 Implementierung

Die `env` Variable ist eine Instanzvariable des Dashboards. Sie ist trotzdem in allen Komponenten zugänglich und kann von dort aus verändert werden. Wird `env` das erste Mal benutzt, ohne dass ein Dashboard existiert, wird dieses automatisch erstellt. Immer wenn eine Komponente verändert wird, wird zuerst der gesamte Datenfluss ausgewertet und anschließend das Dashboard aktualisiert.

Diese Aktualisierung geschieht über mehrere Schritte, die im Quelltext 2.5 nachzuvollziehen sind:

1. Für jedes Attribut (jeden Key) von `env` wird durchgeführt: (Zeilen 3–11)
 - a) Es wird überprüft, ob bereits ein Viewer für diesen Key existiert. (4–5)

⁶Kurz für *environment*.

Quelltext 2.5: Updaten des Dashboards

```
1 update: function() {
2   var _this = this;
3   Properties.own(this.env).each(function(key) {
4     var viewer = _this.getViewerForKey(key);
5     if (!viewer) {
6       viewer = _this.addViewer(key);
7     }
8
9     viewer.update(_this.env[key]);
10    viewer.updated = true;
11  });
12  this.removeUnusedViewers();
13 }
```

- b) Falls noch kein Viewer existiert, wird über `addViewer` ein neuer hinzugefügt. (6)
 - c) Alle Viewer für die Keys aus `env` werden mit den jeweils zugehörigen Daten aktualisiert. (9–10)
2. Alle Viewer, die in diesem Durchlauf nicht aktualisiert wurden, werden entfernt, da deren Key offensichtlich aus `env` entfernt wurde. (12)

Die Methode `addViewer` versucht anhand der anzuzeigenden Daten zu bestimmen, welche Komponente am besten zur Darstellung geeignet ist. Enthalten die Elemente der Daten Morphs, wird ein *Canvas* ausgewählt. Ist dies nicht der Fall, aber es handelt sich um eine flache Datenstruktur, wird eine *Table* angezeigt. Trifft keiner der beiden Fälle zu, wird ein *JSONViewer* hinzugefügt.

2.5 Zusammenfassung und Ausblick

2.5.1 Zusammenfassung

Flower ermöglicht es Nutzern schnell individuelle und interaktive Visualisierungen zu erstellen. Der visuelle, datenflussorientierte Programmieransatz unterstützt den Nutzer dabei vor allem auf konzeptioneller Programzebene. Er nutzt die Ähnlichkeit zur Visualisierungspipeline aus und ermöglicht eine saubere Strukturierung des Programmes.

Flower ist komponentenbasiert und verzichtet auf feste Verbindungen. Dies macht Programme in Kombination mit dem umgesetzten Layoutalgorithmus flexibel und veränderbar. Die verschiedenen Komponententypen helfen dem Nutzer bei immer wiederkehrenden und aufwendigen Arbeitsschritten der Erstellung interaktiver

Grafiken. Gleichzeitig ist es jedoch mit dem *Script* möglich, uneingeschränkt Programmcode zu schreiben. Datenflüsse können unter Nutzern geteilt werden und sind wiederverwendbar.

Das *Dashboard* hilft den wichtigsten Teil des Programmes, das Endergebnis, im Blick zu behalten und kann zur interaktiven Erstellung des Datenflusses genutzt werden.

2.5.2 Verwandte Arbeiten

Lively Fabrik

Lively Fabrik [7] setzt einen sehr ähnlichen Ansatz der visuellen Datenflussprogrammierung um, der noch durch Scripting erweitert werden kann. Komponenten werden im Gegensatz zu *Flowers* Komponenten über feste Pins miteinander verbunden und können somit beliebig angeordnet werden. Erkenntnisse aus diesem Forschungsprototyp sind zurück in den Lively Kernel geflossen. Sie haben maßgeblich zur Implementierung des `lively.connect` beigetragen, von welchem in *Flower* oft⁷ Gebrauch gemacht wird.

Pure Data

Pure Data [8] ist eine quelloffene, visuelle Datenflussprogrammiersprache. Sie wird vor allem im Bereich der Multimedia-Bearbeitung eingesetzt und dient dort als Prototyping-Umgebung. Komponenten sind fest miteinander verbunden und führen ähnlich wie in *Flower* jeweils eine spezifische Aufgabe aus. Pure Data ermöglicht es Nutzern, eigene Objekte in beliebigen Programmiersprachen zu implementieren und diese als externe Komponenten einzubinden. Die Entwicklung an der Programmiersprache wird bis heute fortgesetzt und vor allem über ein Online-Wiki⁸ koordiniert.

2.5.3 Zukünftige Arbeiten

Gruppierung von Komponenten

Bis jetzt ist es dem Nutzer lediglich möglich Datenflussteile im `PartsBin` abzuspeichern, anderen zur Verfügung zu stellen und wiederzuverwenden. Die Größe des abgespeicherten Datenflussteils ist danach dieselbe und kann auch nicht verändert werden. Dieser Arbeitsschritt trägt somit nicht zur Komplexitätsbewältigung bei und verbessert auch nicht die Übersicht in langen und komplexen Datenflüssen. In dieser Richtung besteht noch Verbesserungspotential.

Es könnte die Möglichkeit geben, Datenflussteile zu neuen Komponenten zu gruppieren. Der gruppierte Datenfluss würde als eigene neue Komponente wiederverwendbar sein und vom ursprünglichen Datenfluss abstrahieren. Diese Funktionalität würde es Nutzern zum einen ermöglichen, eigene Komponenten zu schrei-

⁷14 Anwendungsfälle in der aktuellen Version vom 26. Juni 2014.

⁸<http://puredata.info/>; besucht am 25. Juni 2014.

ben und zu verbreiten, sowie zum anderen stark zur Verringerung der Komplexität des Datenflusses beitragen. Sie kann zur Beschleunigung der Visualisierungserstellung führen, indem Nutzer veröffentlichte Komponenten in ihrem Datenfluss wiederverwenden. Eine nachträgliche Editierung der gruppierten Komponenten sollte trotzdem möglich sein. Dies könnte über ein „Aufklappen“ der Komponente geschehen, das den ursprünglichen Datenfluss wieder zeigt.

Nutzerunterstützung

Obwohl *Flower* viel Unterstützung bei der Erstellung von Visualisierungen bietet, ist der Einstieg für Nutzer ohne Vorkenntnisse darüber bisher schwierig. Es existiert außer dieser Arbeit keine Dokumentation, in der Komponenten und Hilfsfunktionen erklärt sind. Hilfreich wäre vor allem die Möglichkeit kontextsensitive Hilfen oder Tipps einzublenden. Vorstellbar ist, dass beim Starten des Programms der Vorschlag erscheint, einen *DataImporter* zu erstellen. Danach könnte der Vorschlag kommen die Daten zu betrachten, in einem *Script* zu verändern oder im *MorphCreator* auf visuelle Elemente abzubilden. Befolgt der Nutzer immer nur die Vorschläge, könnte eine simple Beispielgrafik entstehen. So werden selbst Nutzer, die noch nie eine Visualisierung erstellt haben, an den generellen Ablauf der Visualisierungspipeline herangeführt.

Komponenten könnten außerdem die Möglichkeit bieten, zur Verfügung stehende Hilfsfunktionen samt kurzer Erklärung anzuzeigen. *Flower* würde dadurch entdeckbarer werden und Nutzer wären nicht auf eine pur textuell verfasste Dokumentation angewiesen. Diese Methode bietet außerdem die Möglichkeit den Nutzer Schritt für Schritt an die Visualisierungserstellung heranzuführen, statt ihn von Anfang an mit Informationen zu überfluten.

3 Datenanalyse und -aufbereitung

3.1 Einleitung

3.1.1 Motivation

Der Prozess der Visualisierungserstellung beginnt mit einer Menge von Daten und mündet in deren grafischer Darstellung. Das Ziel dabei ist es, Eigenschaften und Besonderheiten, Ereignisse und Trends, die in den Zahlen versteckt sind, zutage zu fördern und den Betrachter mit geeigneten grafischen Mitteln auf diese aufmerksam zu machen.

Der Weg dorthin ist allerdings nicht so geradlinig, wie es vorerst klingen mag. Der Entwickler wird viele verschiedene Aufbereitungen ausprobieren müssen, um am Ende eine Visualisierung zu erhalten, mit der er zufrieden ist und die seine Aussage unterstreicht. Genau in diesem Prozess gilt es den Nutzer zu unterstützen.

Der erste Schritt während der Visualisierungserstellung ist es, die zugrunde liegenden Daten zu erkunden und geeignet aufzuarbeiten. Dieser Arbeitsschritt ist Gegenstand des folgenden Kapitels. Die späteren Kapitel werden geeignete Daten voraussetzen. Daher bildet die Datenaufbereitung das Fundament einer gelungenen Visualisierung.

Im Folgenden wird an einem Beispiel der Prozess der Datenaufbereitung in *Flower* überblicksartig vorgestellt. Anschließend wird der Fokus auf den einzelnen Schritten liegen und deren Umsetzung im Detail betrachtet und diskutiert.

3.1.2 Datenaufbereitung am Beispiel

In diesem Abschnitt wird der Ablauf der Datenvorverarbeitung an einem konkreten Beispiel nachvollzogen. Es steht im Kontext der Fragestellung, ob die Siegwahrscheinlichkeit von Fußball-Bundesliga-Mannschaften bei bestimmten Witterungsbedingungen von ihrer generellen Siegwahrscheinlichkeit abweicht. Abbildung 3.1 beantwortet diese Frage und beruht auf den Daten, die im Datenfluss in Abbildung 3.2 erzeugt werden.

Es werden dafür zwei Datenquellen benötigt: zum einen die Ergebnisse der Mannschaften aus den vergangenen Spielzeiten, zum anderen zu jedem Spieltag das Wetter am entsprechenden Spielort.

Für erstere Daten erwies sich eine britische Wett-Datenbank¹ als äußerst nützlich, da sie gut strukturierte und vollständige Daten mit allen nötigen Informationen

¹<http://www.football-data.co.uk/germanym.php>; besucht am 01. Oktober 2014.

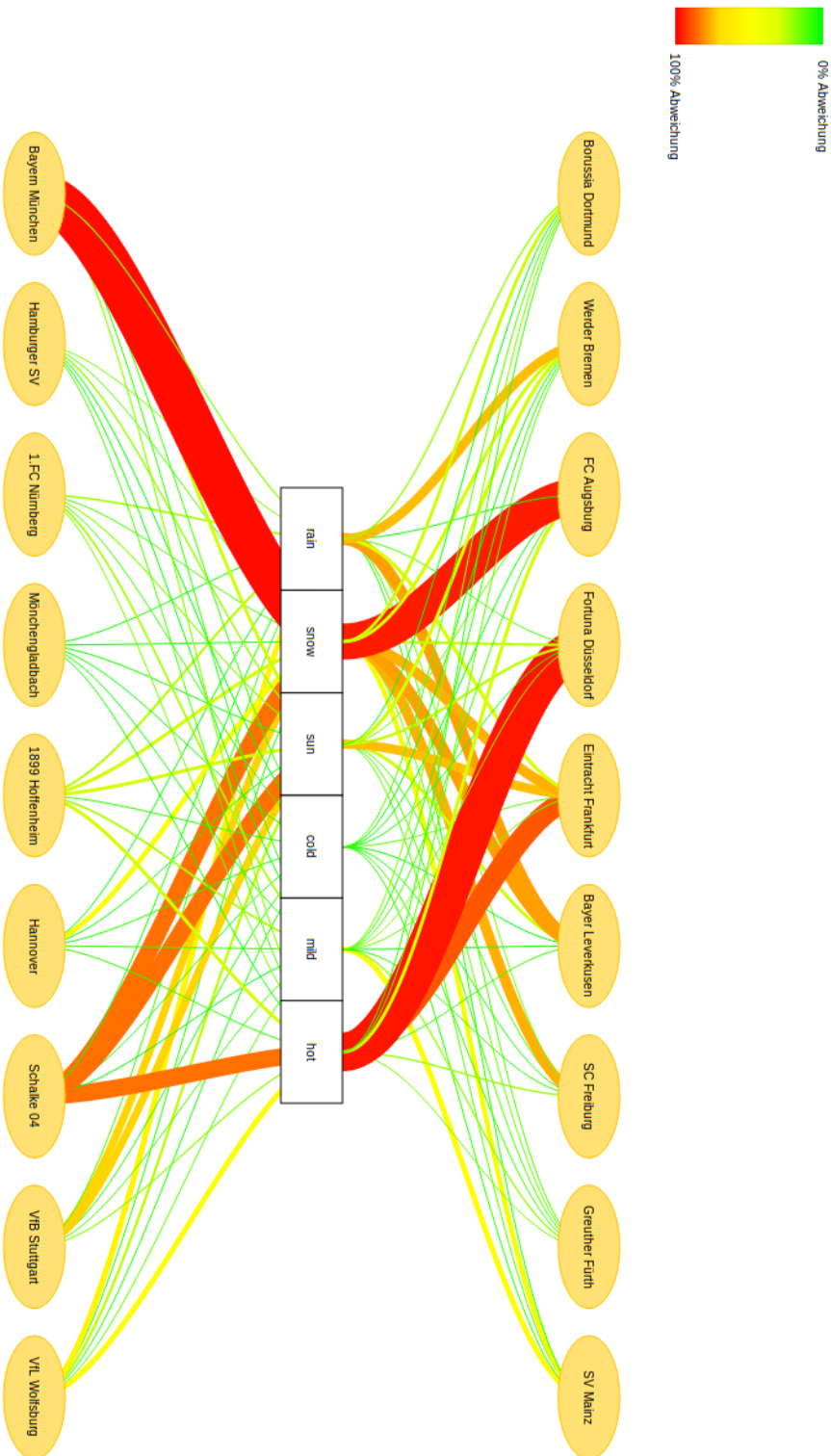


Abbildung 3.1: Abweichung der Siegwahrscheinlichkeiten von Fußballmannschaften abhängig vom Wetter

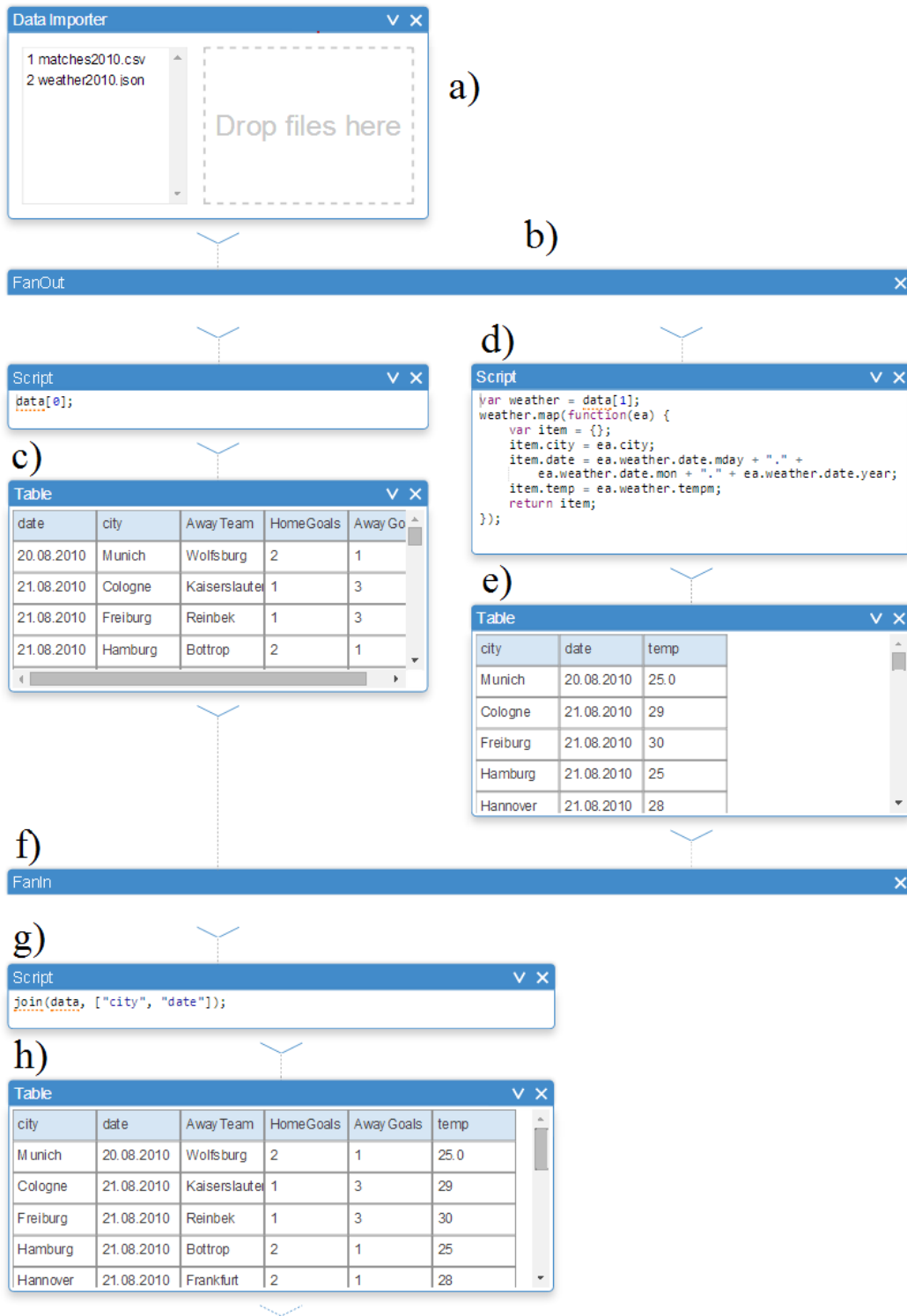


Abbildung 3.2: Beispielhafte Datenaufbereitung in Flower

3 Datenanalyse und -aufbereitung

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
1	Div	Date	HomeTeam	AwayTeam	FTHG	FTAG	FTR	HTHG	HTAG	HTR	HS	AS	HST	AST	HF	AF	HC	AC	HY	AY	HR	AR	B365H	B365
2	D1	20.08.2010	Bayern Munich	Wolfsburg	2	1	H	1	0	H	17	11	5	5	6	25	9	3	1	3	0	0	1,57	
3	D1	21.08.2010	FC Köln	Kaiserslautern	1	3	A	1	0	H	10	17	4	6	10	25	5	6	1	2	1	0	2,1	
4	D1	21.08.2010	Freiburg	St Pauli	1	3	A	0	0	D	9	17	3	7	12	13	3	6	0	0	0	0	2,2	
5	D1	21.08.2010	Hamburg	Schalke 04	2	1	H	0	0	D	18	13	6	4	11	17	7	4	2	0	0	1	2,38	
6	D1	21.08.2010	Hannover	Ein Frankfurt	2	1	H	1	1	D	13	17	7	3	9	14	2	6	0	1	0	0	2,6	
7	D1	21.08.2010	Hoffenheim	Werder Bremen	4	1	H	4	1	H	10	10	7	2	24	16	5	5	1	2	0	0	2,88	
8	D1	21.08.2010	M'gladbach	Numberg	1	1	D	1	1	D	12	10	5	5	15	18	6	5	1	1	0	0	1,91	
9	D1	22.08.2010	Dortmund	Leverkusen	0	2	A	0	2	A	14	15	6	4	20	13	8	1	0	0	0	0	2,38	
10	D1	22.08.2010	Mainz	Stuttgart	2	0	H	1	0	H	17	14	6	5	16	13	5	4	4	1	0	0	3,1	
11	D1	27.08.2010	Kaiserslautern	Bayern Munich	2	0	H	2	0	H	11	20	2	3	24	13	7	7	0	1	1	0	7	4
12	D1	28.08.2010	Ein Frankfurt	Hamburg	1	3	A	1	0	H	12	14	7	4	12	12	4	6	2	1	0	0	3,1	
13	D1	28.08.2010	Numberg	Freiburg	1	2	A	1	1	D	17	5	4	3	12	28	4	2	2	1	0	0	1,91	
14	D1	28.08.2010	Schalke 04	Hannover	1	2	A	0	1	A	15	11	5	4	10	22	15	1	2	2	0	0	1,33	4
15	D1	28.08.2010	St Pauli	Hoffenheim	0	1	A	0	0	D	12	8	4	3	16	16	5	9	2	0	0	0	3,2	4

Abbildung 3.3: Ergebnisse der Fußball-Bundesliga in der Saison 2010/11

kostenlos zur Verfügung stellt. Ein Auszug aus den Daten ist in Abbildung 3.3 zu sehen. Es handelt sich dabei um eine Datei im CSV-Format, welche in den Drop-Bereich des *DataImporters* (Abbildung 3.2, Schritt a) verschoben wird. Der *DataImporter* ist eine Komponente, die in Abschnitt 3.2.2 detailliert vorgestellt wird. Nach dem Importieren stehen die Daten in der nächsten Komponente zur Verfügung.

Nun sind noch die Wetterdaten für die Spielorte an den entsprechenden Tagen nötig. Die Daten können von einer Wetter-API² abgefragt und lokal in einer Datei, zum Beispiel *weather2010.json*, gespeichert werden. Anschließend kann diese Datei mit Hilfe des *DataImporters* dem Datenfluss zugeführt werden.

Im nächsten Datenflusselement (b) spaltet sich der Datenfluss in zwei Stränge auf. Die linke Seite kümmert sich um die Spielergebnisse. Sie werden in einer Tabelle (c) angezeigt, mit Hilfe derer alle nicht benötigten Spalten gelöscht werden können. Aus Gründen der Übersichtlichkeit werden die Spalten *FTHG* und *FTAG* in *HomeGoals* und *AwayGoals* umbenannt. Der Name der Spalte *HomeTeam* wird zu *city* geändert.

Die rechte Seite verarbeitet die Wetterdaten. Diese haben eine verschachtelte Struktur (siehe Abbildung 3.4), die in einer Tabelle nicht geeignet angezeigt und daher auch nicht bearbeitet werden kann. Alternativ besteht hier die Möglichkeit, ein *Script* (d) zu nutzen. Dort kann beliebiger JavaScript-Code auf die Daten angewandt werden. Im Beispiel selektiert dieser Code mit Hilfe der *map*-Funktion die Datenwerte *city*, *date* und *temp*. Zudem formt er das Datumsformat nach TT.MM.JJJJ um, damit es für den Menschen besser lesbar ist. Die Tabelle (e) darunter zeigt an, ob die Umformungen das Gewünschte bewirkt haben.

Die *FanIn*-Komponente (f) verbindet im nächsten Schritt die beiden Datenstränge wieder miteinander. Im darauf folgenden *Script* (g) können dann die beiden Datenquellen miteinander verschmolzen werden. Die *join*-Funktion erleichtert diese Aufgabe – sie vereinigt die Daten über die gemeinsamen Attribute *city* und *date*. Hier wird jetzt auch deutlich, warum *HomeTeam* in den Spieldaten in *city* umbenannt wurde.

²Beispielsweise <http://www.wunderground.com>; besucht 01. Oktober 2014.


```
[
  {
    "city": "Munich",
    "weather": {
      "date": {
        "pretty": "4:20 PM CEST on August 20, 2010",
        "year": "2010",
        "mon": "08",
        "mday": "20",
        "hour": "16",
        "min": "20",
        "tzname": "Europe/Berlin"
      },
      "utcdate": {
        "pretty": "2:20 PM GMT on August 20, 2010",
        "year": "2010",
        "mon": "08",
        "mday": "20",
        "hour": "14",
        "min": "20",
        "tzname": "UTC"
      },
      "tempm": "25.0",
      "tempf": "77.0",
      "dewptm": "16.0",
      "dewptf": "60.8",
      "hum": "57",
      "wspd": "1.9",
      "wspd_i": "1.2",
      "wgust": "-9999.0",
      "wgust_i": "-9999.0",
      "wdird": "0",
      "wdire": "Variable",
      "vis": "10.0"
    }
  }
]
```

Abbildung 3.4: Auszug aus der Datenstruktur der abgefragten Wetterdaten

Durch diese Vereinigung entstehen die Daten, die in der letzten Tabelle (h) zu sehen sind. Mit diesen kann jetzt die Visualisierungserstellung fortgesetzt werden. In den folgenden Abschnitten werden die hier sehr oberflächlich beschriebenen Funktionen detaillierter beleuchtet und ihre Funktionsweise erläutert.

3.2 Daten importieren und konvertieren

Dieser Abschnitt wird sich damit beschäftigen, auf was für Dateiformate der Programmierer während der Datenbeschaffung treffen kann und wie diese als Grundlage für die Visualisierung homogen genutzt werden können. Dazu werden kurz vier häufige Formate (csv, xls, XML und JSON) mit ihren jeweiligen Eigenschaften vorgestellt. Anschließend wird der *DataImporter* eingeführt. Diese Komponente ermöglicht das Importieren und Konvertieren von Daten innerhalb von *Flower*.

3.2.1 Dateiformate und deren Verwendung

csv

Die Abkürzung *csv* steht für *comma separated values* und bezeichnet ein Dateiformat für Textdateien, welches unter anderem für die Übermittlung und Konvertierung von Daten zwischen Tabellenkalkulationsprogrammen genutzt wird. Aufgrund dessen, dass es keine formale und global anerkannte Definition für diesen Dateityp gibt, gehen die genauen Implementierungen allerdings auseinander. Die wichtigsten Eigenschaften lassen sich jedoch in wenigen Punkten zusammenfassen [13]:

1. Jede Zeile in der Datei enthält einen Datensatz.
2. Die einzelnen Felder im Datensatz sind durch Kommas voneinander getrennt.
3. Jede Zeile, das heißt jedes Datum, sollte die gleiche Anzahl an Feldern besitzen.
4. In der ersten Zeile kann ein Tabellenkopf definiert werden, indem die jeweiligen Spaltenbezeichnungen als Felder eines Datensatzes hinzugefügt werden.

Eine Beispieldatei, die den Stundenplan eines Schülers beinhaltet, könnte wie folgt aufgebaut sein:

```
Mo,Di,Mi,Do,Fr
Mathe,Geschichte,Deutsch,Deutsch,Sport
Mathe,Erdkunde,Englisch,Deutsch,Sport
Latein,Musik,Religion,Mathe,Kunst
Sport,Englisch,,Mathe,
```

xls(x)

Hierbei handelt es sich um das Format des Tabellenkalkulationsprogramms Microsoft Excel, wobei *xls* nach der Version *Excel 2003* von *xlsx* abgelöst wurde. *xls*

ist im Verhältnis zum zuvor beschriebenen *csv*-Format deutlich umfangreicher in seiner internen Struktur. Es beruht auf dem Konzept von Datenströmen. So existiert beispielsweise für jede Tabelle, die in einer Datei gespeichert ist, ein separater Datenstrom. Das programmatische Lesen von Datenwerten ist mit einigem Aufwand verbunden und ist von *Microsoft* in einem Algorithmus definiert. Der interessierte Leser sei an dieser Stelle auf die Dokumentation von *Microsoft* verwiesen [10].

XML

Ausgeschrieben heißt dieses Dateiformat *Extensible Markup Language*. Es beschreibt hierarchische Datenstrukturen, die sowohl von Maschinen als auch von Menschen lesbar sein sollen. Im Folgenden eine Beispieldatenstruktur:

Quelltext 3.1: XML-Daten, welche Eigenschaften von zwei Personen beinhalten

```

1 <?XML version="1.0" encoding="UTF-8" ?>
2 <Daten>
3   <Person id="1">
4     <Name>Peter</Name>
5     <Alter>31</Alter>
6     <Kinder>
7       <Kind>Claudia</Kind>
8       <Kind>Dieter</Kind>
9     </Kinder>
10    <Telefon>
11      <Festnetz>030/29837429</Festnetz>
12      <Mobil>0176/23479283</Mobil>
13    </Telefon>
14  </Person>
15  <Person id="2">
16    <Name>Hannah</Name>
17    <Alter>43</Alter>
18    <Kinder>
19      <Kind>Paula</Kind>
20      <Kind>Katja</Kind>
21      <Kind>Henriette</Kind>
22    </Kinder>
23    <Telefon>
24      <Mobil>0151/84989874</Mobil>
25    </Telefon>
26  </Person>
27 </Daten>

```

Logisch ist ein *XML*-Dokument aus einem oder mehreren *Elementen* aufgebaut. Jedes *Element* wird dabei von *start-* und *end-tags* beschränkt. In der oberen Datenstruktur sind das zum Beispiel `<Daten>` und `</Daten>`. *Tags* haben dabei mindestens ein Attribut, nämlich einen Namen. Weitere Attribute können innerhalb eines *Tags* definiert werden. Dies ist bei `<Person id='1'>` am Attribut *id* zu erkennen. Um ein wohlgeformtes Element zu definieren, muss der Name im *end-tag* mit dem im *start-tag* übereinstimmen. Sie bilden in gewisser Weise öffnende und schließende

Klammer um ein Element. Dies ist allerdings nur eine von mehreren Bedingungen, die erfüllt werden sollten. Näheres ist in der W3C Empfehlung nachzulesen [14].

Neben der Wohlgeformtheit eines Dokuments kann es auch noch valide sein, wenn es einer angegebenen Grammatik entspricht, zum Beispiel einem *XML-Schema*. Diese Schemata erlauben es, sowohl einfache als auch komplexe Datentypen zu definieren. Details dazu sind in der Folge dieser Arbeit jedoch nicht von Bewandtnis. Bei tieferem Interesse sollte die Empfehlung des W3C [15, 16] zurate gezogen werden.

JSON

Dies ist ein Dateiformat, welches es ermöglicht, die Daten ähnlich wie in *XML* hierarchisch aufzubauen. Das Ziel ist dabei ebenfalls, von Menschen und Maschinen gleichermaßen gut zu lesen und zu schreiben zu sein. Die Abkürzung steht für *JavaScript Object Notation*, welche bereits auf die Verwandtschaft mit JavaScript hinweist. Es basiert dabei auf einer Teilmenge der JavaScript Syntax, genauer gesagt auf zwei Grundstrukturen [9]:

1. einer Sammlung von Schlüssel/Wert-Paaren, in JavaScript als Object bezeichnet und
2. einer Liste von Werten, gemeinhin als Array bekannt.

Diese Strukturen sind Programmiersprachen-übergreifend verfügbar, was JSON zu einem sprachunabhängigen Format macht. Es ist also möglich, Inhalte einer JSON-Datei innerhalb beliebiger Programmiersprachen einzulesen.

Aus den Grundstrukturen von JSON lassen sich nun Datenstrukturen erstellen, wie zum Beispiel die folgende:

```
var data = [{
  "Name": "Peter",
  "Alter": 31,
  "Kinder": ["Claudia", "Dieter"],
  "Telefon": {
    "Festnetz": "030/29837429",
    "Mobil": "0176/23479283"
  }
}, {
  "Name": "Hannah",
  "Alter": 43,
  "Kinder": ["Paula", "Katja", "Henriette"],
  "Telefon": {
    "Mobil": "0151/84989874"
  }
}]
```

Hier wird ersichtlich, dass sich aus den beiden Grundstrukturen durch deren Komposition beliebig tief geschachtelte Datenstrukturen erzeugen lassen. Ein beispielhafter Zugriff auf die Handynummer von Peter könnte dabei so aussehen:

```
var nummer = data[0].Telefon.Mobil;  
// nummer == "0176/23479283"
```

Da die Syntax einer JSON-Datei eine Teilmenge der JavaScript-Syntax ist, ist der Inhalt einer JSON-Datei bereits valides JavaScript. Das macht die Integration dieser Daten sehr komfortabel. Die Hilfsfunktion `JSON.parse` nimmt als Argument einen String und versucht diesen zu evaluieren. Handelt es sich bei dem String um valides JSON, gibt sie den Inhalt als JavaScript-Objekt zurück. Hier ein kleines Anwendungsbeispiel:

```
var obj = JSON.parse('{ "a":5, "b":7 }');  
// obj == {a: 5, b:7}
```

Vorkommen dieser Formate

Die oben vorgestellten Formate decken nicht die Gesamtheit der möglichen Dateiformate ab, die dem Programmierer bei der Visualisierungserstellung über den Weg laufen können. Jedoch waren alle Datenquellen, auf die wir im Laufe unserer Erarbeitung gestoßen sind, stets in einem dieser Formate vorhanden. Das führt zu der Annahme, dass bei Unterstützung dieser Formate die meisten Anwendungsfälle abgedeckt werden.

3.2.2 DataImporter

Motivation

So vielfältig die Möglichkeiten auch sind, die sich durch all die öffentlich im Internet zugänglichen Datenquellen bieten, so vielfältig sind auch die Zugänge zu diesen Daten und deren Formate. Im anfänglichen Fußball-Beispiel gab es zum einen eine *csv*-Datei als Download, zum anderen wurde mit Hilfe einer Wetter-API eine JSON-Datei generiert. Diese und weitere häufige Dateiformate wurden bereits im vorherigen Abschnitt tiefergehend behandelt.

Dem Programmierer soll es ermöglicht werden die gängigsten Datenformate homogen zu nutzen, sodass er sich um Konvertierungen nicht kümmern muss.

Funktionalität

Der *DataImporter* bietet dabei die Möglichkeit, von den Dateiformaten *csv*, *xls* und JSON zu abstrahieren. Er konvertiert diese Formate nach JSON und interpretiert sie anschließend als JavaScript-Objekte, welche im Datenfluss genutzt werden können. Die Daten, die er an die nächste Komponente weiterreicht, sind ein Array, in dem jeder Eintrag die Daten einer importierten Datei referenziert. Wurden beispielsweise drei Dateien importiert, so beinhaltet `data` in der nächsten Komponente initial ein dreielementiges Array.

Um Daten zu importieren, existieren die folgenden Möglichkeiten:

1. *Drag-and-Drop* der Datei in den Drop-Bereich
2. *Copy-and-Paste* einer URL zu dieser Datei in den Drop-Bereich

3. Copy-and-Paste einer Zeichenkette, welche die Daten selbst enthält

Nach dem Import von Daten erscheint eine neue Zeile in der Liste auf der linken Seite. Wenn ein Dateiname gefunden wird, wird dieser angezeigt, ansonsten die ersten Zeichen der Daten.

3.3 Daten erkunden

Im Anschluss an das Importieren von Daten ist der nächste Schritt, herauszufinden, was im Detail in diesen Daten enthalten ist. Dazu sind Aspekte wie Struktur, Wertebereiche einzelner Attribute und konkrete Datenelemente interessant, da diese Kennwerte einen Überblick über die Gesamtdaten geben können. Der folgende Abschnitt beschreibt die Möglichkeiten, die *Flower* dafür bietet.

3.3.1 JSONViewer

Die Daten im Datenfluss sind JavaScript-Objekte, die Struktur dieser Objekte ist hierarchisch. Hierarchische Strukturen sind in der Informatik weit verbreitet und bekannt, weshalb zum Anzeigen auch auf bekannte Konzepte zurückgegriffen wird. Es kommt eine interaktive Baumstruktur zum Einsatz, wie sie zum Beispiel auch zur Anzeige von Verzeichnisbäumen genutzt wird. Der Nutzer ist in der Lage, Teilbäume per Klick auf- und zuzuklappen und so verschiedene Detailtiefen darzustellen. Dabei werden auch die verschiedenen Datentypen der Attributwerte verschiedenfarbig dargestellt.

Abbildung 3.5 zeigt einen *JSONViewer* in zwei Zuständen. Die Eingabedaten sind dabei die Wetterdaten vor ihrer Bearbeitung (siehe Abbildung 3.2). Auf der linken Seite ist eine Übersicht über alle Elemente dieser Datenstruktur zu erkennen. Auf der rechten Seite wurde das erste Element zwei Hierarchiestufen aufgeklappt und offenbart so eine größere Detailtiefe.

Diese Darstellungsweise ist für jedes Objekt möglich. Da die Anzeigestruktur beliebig weit expandiert werden kann, können jegliche Werte inspiziert werden. Zudem eignet sie sich dazu, einen Eindruck vom Aufbau einzelner Datenelemente zu gewinnen. Zusätzlich ist zu erkennen, wie viele Elemente in der gesamten Datenstruktur vorhanden sind.

Dem gegenüber steht jedoch, dass es kaum möglich ist, einen Überblick über einen Werteverlauf eines Attributes zu bekommen. Im Wetter-Beispiel müssen, um den Wertebereich von *tempm* zu erkunden, alle Elemente zwei Ebenen tief aufgeklappt werden. Im Anschluss kann durch alle Daten gescrollt und jeweils der Attributwert herausgesucht werden.

Eine Frage ist, wie viele Elemente initial beim Anzeigen neuer Daten aufgeklappt werden sollen und um wie viele Ebenen jeweils. Dabei gilt es zu beachten, dass es bei großen Datenmengen durchaus einige Sekunden dauern kann, bis alle Elemente komplett expandiert sind. Aus Performancegründen wäre es naheliegend, keines der Elemente automatisch zu öffnen. Es ergäbe sich dann anfänglich die Ansicht

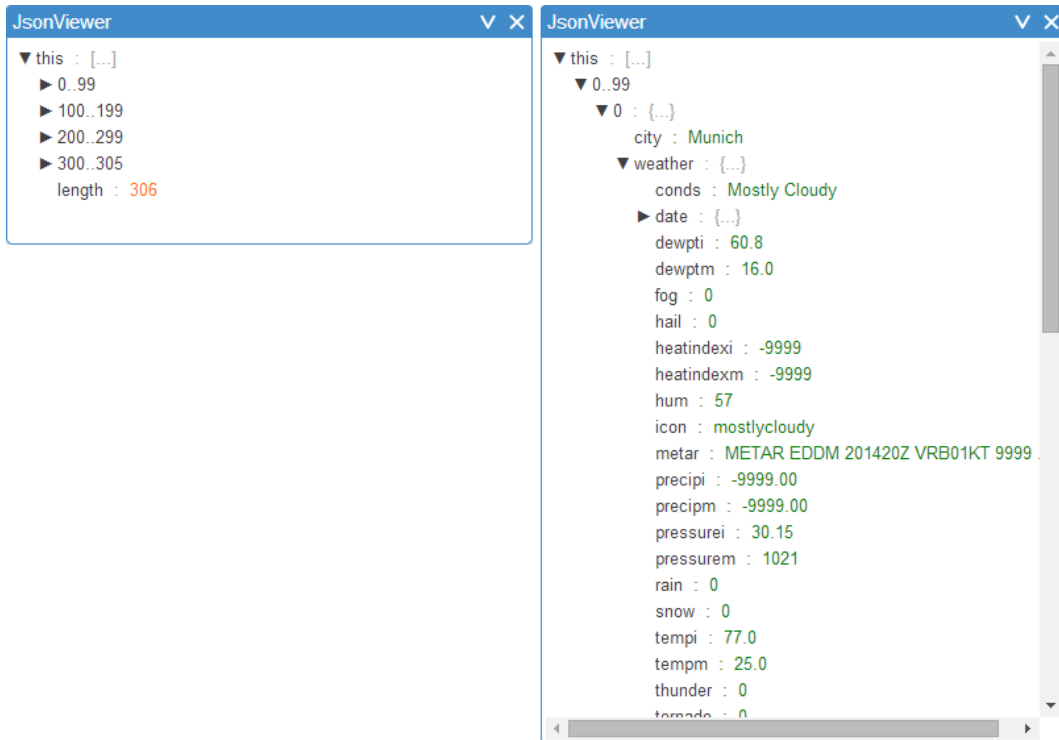


Abbildung 3.5: JSONViewer, der die Wetterdaten für 306 Spiele in zwei verschiedenen Detailtiefen anzeigt

aus Abbildung 3.5 links. Dabei ist aber festzustellen, dass dies für die meisten Anwendungsfälle nicht nützlich ist. Der Inhalt der Elemente ist oft entscheidender als die Tatsache, dass sie vorhanden sind oder ihre bloße Anzahl. Daher ist das gewählte Standardverhalten, immer das erste Element zu öffnen. Weil damit im Verhältnis zum Expandieren aller Elemente eine Menge Rechenzeit gespart wird, wird dieses Element dann drei Ebenen tief geöffnet. So kann es in vielen Fällen bereits ohne einen weiteren Klick komplett inspiziert werden.

Implementierung

Die *JSONViewer*-Klasse erbt von *Content* (siehe Abbildung 2.7, Seite 9) und stellt somit einen Inhalt für *Window*- und *DataflowComponents* dar. Er implementiert dabei die Identitätsfunktion. Das heißt, er verändert die Eingabedaten nicht, sondern reicht sie unverändert weiter.

Um die Logik der Baumansicht nicht neu implementieren zu müssen, nutzt er einen *lively.morphic.Tree*. Wird *update* auf dem *JSONViewer* aufgerufen, so übersetzt er diesen Aufruf dementsprechend, dass der *Tree* die Daten anzeigen kann.

3.3.2 Table

Neben den tief verschachtelten Datenstrukturen existieren auch flache Datenstrukturen. Die Einfachheit kann in einem *JSONViewer* jedoch nicht gut ausgenutzt

werden, um eine bessere Übersicht über die Inhalte der Daten zu gewähren – wohl aber in einer Tabelle. Als Beispiel dient die Datenstruktur, die man nach der Umformung der Wetterdaten erhält (siehe Abbildung 3.2, Schritt d). Das JavaScript-Objekt dahinter hat die folgende Struktur:

Quelltext 3.2: Struktur der Wetterdaten nach der Umformung

```
var data = [{
  "city": "Munich",
  "date": "20.08.2010",
  "temp": 25
},{
  "city": "Cologne",
  "date": "21.08.2010",
  "temp": 29
},
  // ...
}
```

Es handelt sich um ein Array gleichartiger Objekte, in dem die Objekte jeweils primitive Attributwerte besitzen. Das bedeutet, dass alle Inhalte bereits in der ersten Hierarchiestufe der einzelnen Elemente liegen. Um daraus eine Tabelle zu erzeugen, werden alle vorkommenden Attribute der Datenobjekte extrahiert. Diese Attribute werden als Spaltenbezeichnungen in der Tabelle genutzt. Anschließend wird für jedes Datum eine Tabellenzeile mit seinen Werten befüllt. Die Tabelle hat demnach neben der Kopfzeile so viele Zeilen wie das Datenarray Elemente und so viele Spalten wie die Daten verschiedene Attribute in der ersten Hierarchiestufe.

Da alle vorkommenden Attribute für die Spaltennamen genutzt werden, kann es bei verschiedenartigen Daten vorkommen, dass manche Werte zu einigen Objekten nicht existieren, siehe Abbildung 3.6. Deutschland hat (zumindest formal) keine Königin. Kommt es zu diesem Fall, weist ein Strich im Tabellenfeld auf das Fehlen eines Wertes hin.

Des Weiteren kann es passieren, dass sich nicht nur primitive Werte in den Daten befinden, sondern auch Objekte. Diese können in einer Tabelle allerdings nicht dargestellt werden. Das ist der Fall beim Attribut *Flagge* dieses Beispiels. Interessiert sich der Nutzer für solche Werte, muss er die Ansicht zu einem *JSONViewer* ändern³.

3.3.3 Deskriptive Statistiken

Ein weiterer Schritt der Datenerkundung ist die Anwendung von statistischen Methoden auf die Datenreihen.

³Dieser Wechsel ist unproblematisch, siehe *Austausch des Komponenteninhalts* in Abschnitt 2.3.1, Seite 8.

Land	Kennzeichen	Flagge	Königin
Deutschland	D	[object Object]	-
Großbritannien	GB	[object Object]	Elisabeth II

Abbildung 3.6: Tabelle, die verschiedenartige Datenobjekte anzeigt

„Statistik ist die Lehre von Methoden zur Gewinnung, Charakterisierung und Beurteilung von zahlenmäßigen Informationen über die Wirklichkeit (Empirie).“ [17]

Die statistischen Methoden lassen sich in die folgenden zwei Bereiche einteilen [18]:

Die *Deskriptive Statistik*, auch beschreibende Statistik genannt, fasst Datenmengen mittels Kenngrößen zusammen und verzichtet dabei auf verallgemeinernde Aussagen.

Die *Induktive Statistik*, auch schließende oder folgernde Statistik, baut auf der Wahrscheinlichkeitstheorie auf und versucht, Aussagen über eine Teilmenge für eine größere Gesamtheit von Daten zu verallgemeinern.

Bei der Datenerkundung geht es darum, große Datenmengen in geeigneter Weise fassbar und verständlich zu machen. Die *deskriptive Statistik* bietet Kennwerte, welche große Mengen von Werten in einzelnen Zahlen zusammenfassen können. Beispielsweise ist es mit dem Minimum und dem Maximum möglich einen Wertebereich abzugrenzen, ohne alle konkreten Werte anschauen zu müssen. Weitere bekannte Kenngrößen sind Durchschnitt, Standardabweichung und Varianz, die jeweils bestimmte Eigenschaften von Daten in einem Wert darstellen. Um dem Nutzer die Möglichkeit dieser Datenzusammenfassung zu bieten, sollen deskriptive Statistiken komfortabel berechenbar sein. *Induktive Statistiken* verfolgen andere Ziele und werden in der Datenerkundung im Folgenden keine Rolle spielen.

Ausgangspunkt für die Berechnung ist die *Table*, in welcher die einzelnen Spalten die Datenreihen darstellen, auf denen die Kalkulationen stattfinden. Abbildung 3.7 zeigt die Statistiken für die geschossenen Tore von Heimmannschaften in 306 Spielen. Zunächst wurde dazu ein Kontextmenü auf der Kopfzelle *HomeGoals* geöffnet und dort auf *Statistics* geklickt. Daraufhin öffnet sich die *StatisticTable*, welche die Ergebnisse vorhandener statistischer Funktionen tabellarisch anzeigt. Dies kann man für verschiedene Spalten der Daten machen und so die statistischen Kennwerte miteinander vergleichen. In diesen 306 Spielen, was einer Saison entspricht, fielen im Durchschnitt 1.65 Heimtore, dagegen 1.27 Auswärtstore. Die Heimmannschaften waren demnach was die Durchschnittstore angeht erfolgreicher.

Selbstdefinierte Funktionen

Neben den bereits vordefinierten statistischen Funktionen soll zusätzlich die Möglichkeit existieren, eigene Funktionen zu definieren. So ist der Programmierer in der Lage, auf seinen spezifischen Daten Funktionen anzuwenden, die genau auf

3 Datenanalyse und -aufbereitung

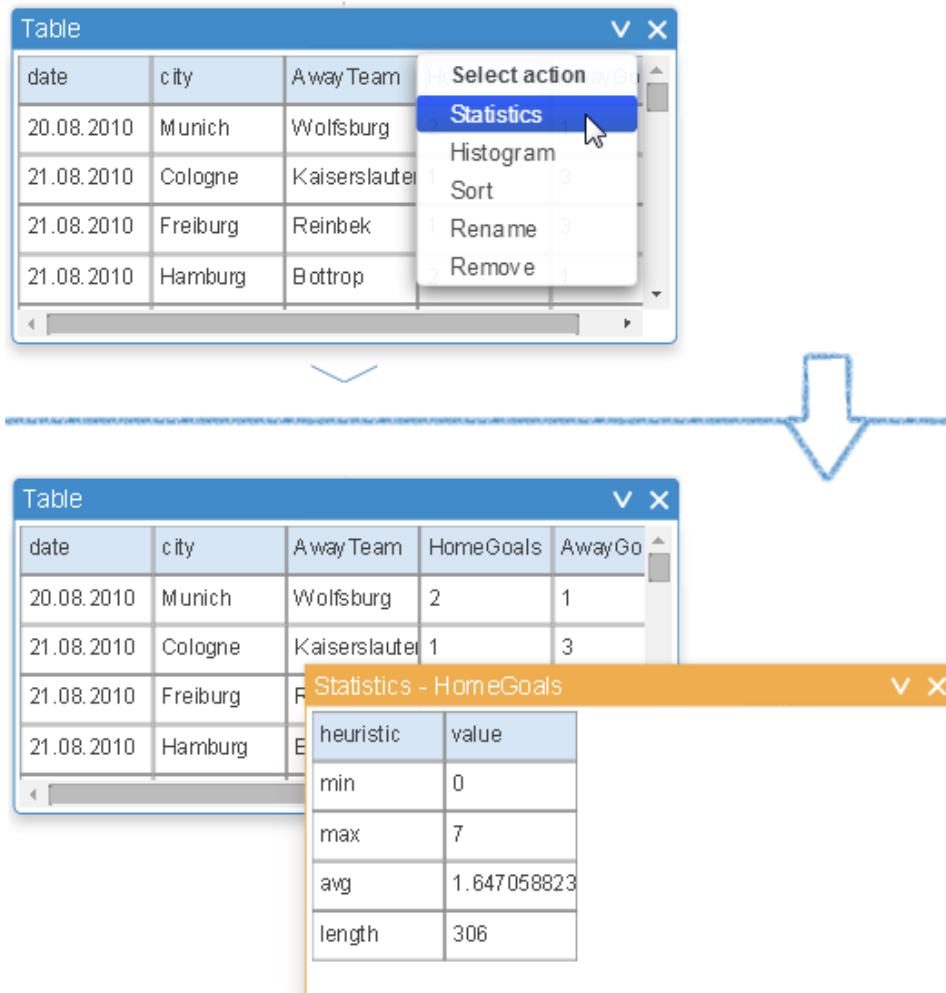


Abbildung 3.7: Statistiken für die Tore der Heimmannschaften

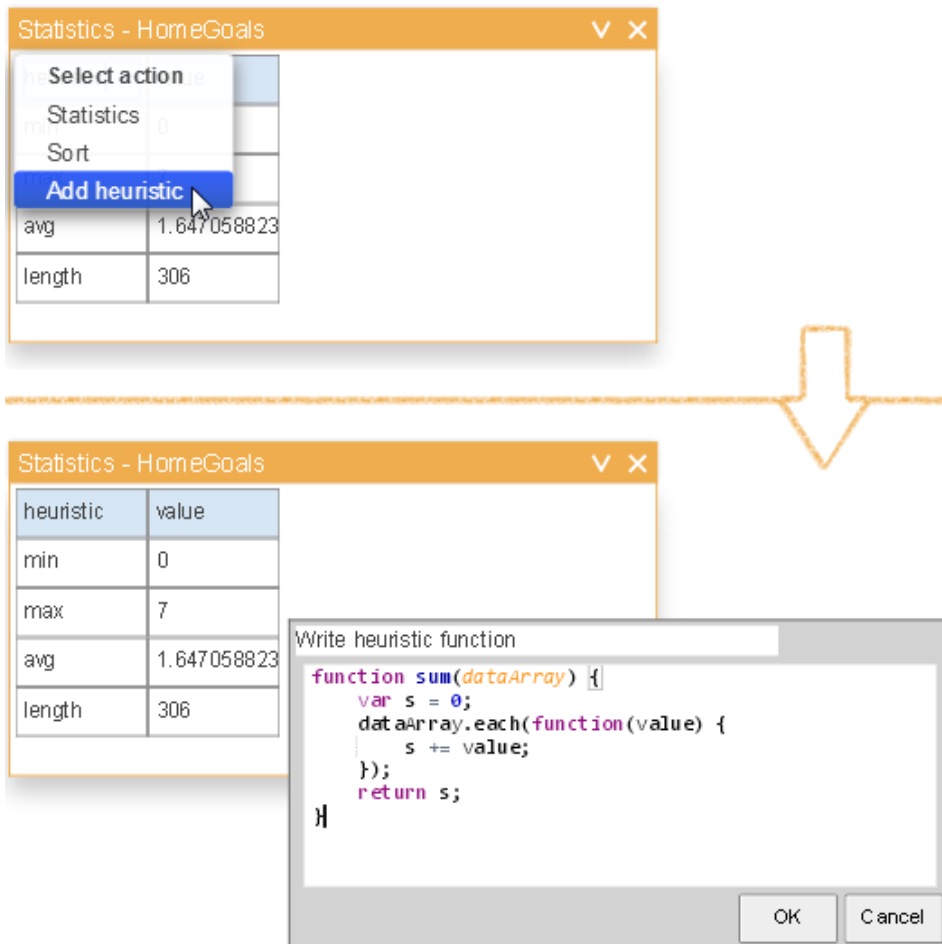


Abbildung 3.8: Hinzufügen einer neuen statistischen Funktion

diesen Anwendungsfall zugeschnitten sind. Eine mögliche Fragestellung wäre, wie oft eine Heimmannschaft mehr als drei Tore geschossen hat. Der Ablauf dafür ist in Abbildung 3.8 zu erkennen. Um diese Fragestellung zu beantworten, wird auf einer Kopfzelle der *StatisticTable* ein Kontextmenü geöffnet und der Punkt *Add heuristic* gewählt. Daraufhin öffnet sich ein Dialog, in dem eine Funktion geschrieben werden kann. Als Standard ist dort bereits eine Funktion vorhanden, die alle Daten aufsummiert (siehe Abbildung 3.8). Diese erhält als Argument ein Array mit den Werten der Spalte, für welche die *StatisticTable* geöffnet wurde – im Beispielfall die Tore der Heimmannschaften. Diese Werte werden zur Berechnung der neuen statistischen Kenngröße genutzt, welche die Funktion am Ende zurückgibt.

Um die Spiele zu zählen, in denen die Heimmannschaft mehr als drei Tore geschossen hat, wird die folgende Funktion implementiert:

```
1 function gt3(dataArray) {
2     var s = 0;
3     dataArray.each(function(value) {
4         if (value > 3) s++;
5     });
6     return s;
7 }
```

Wird der Dialog anschließend bestätigt, erscheint eine neue Zeile in der *StatisticTable* mit der Heuristik, die den Namen der eingegebenen Funktion trägt, und dem entsprechenden Wert. In diesem Fall erfüllen 28 Spiele die Bedingung.

Implementierung

Eine Anforderung an die Implementierung ist, dass die Menge der vorberechneten statistischen Funktionen leicht erweiterbar ist. Um das umzusetzen, ist es wichtig, dass nicht jede dieser Funktionen sozusagen *hard-coded* ausgeführt werden muss. Dazu wird eine Menge an statistischen Funktionen an einer Stelle definiert, welche dann alle nacheinander auf die Daten angewandt werden können. Die Menge dieser Funktionen wird in der Funktion `getStatisticFunctions` angelegt. Der folgende Quelltext lässt ihre Struktur erkennen.

```
1 getStatisticFunctions() {
2     ...
3
4     // calculate the average value
5     var avg = function(dataArray) {
6         // sum up the arrays elements
7         var sum = ...;
8
9         return sum / dataArray.length;
10    }
11
12    // calculate the maximum
13    var max = function(dataArray) {
14        return dataArray.max();
15    }
16 }
```

```

17     ...
18
19     return [
20         ...
21         {name: "avg", calculation: avg},
22         {name: "max", calculation: max},
23         ...
24     ];
25 }

```

Der Aufrufer bekommt ein Array von Objekten zurück, welche jeweils den Namen des statistischen Wertes und dessen Berechnungsvorschrift enthalten. Im Falle von `max` kann in der Berechnungsvorschrift auf die Implementierung in der Klasse `Array` zurückgegriffen werden. Für den Durchschnitt muss die Logik selbst implementiert werden.

Um die statistischen Werte zu berechnen, wird die Menge an Funktionen wie folgt genutzt:

```

1 createStatistics(data) {
2     var result = [];
3     var _this = this;
4     var calculation;
5
6     this.functionSet.each(function(ea) {
7         eval(_this.getStatisticsContext() + " calculation = " + ea.
            calculation);
8         var item = {
9             heuristic: ea.name,
10            value: calculation(data)
11        };
12        result.push(item);
13    });
14
15    this.update(result);
16 }

```

Die Instanzvariable `functionSet` wurde im Konstruktor der `StatisticTable` mittels der Funktion `getStatisticFunctions` initialisiert. Über dieses Array wird iteriert und für jede Funktion ein neues Objekt erzeugt. Dieses enthält zum einen den Namen der Statistik, zum anderen den Wert, der sich aus der Anwendung der Berechnungsvorschrift auf die aktuellen Daten ergibt. Das `result`-Array entspricht der Datenstruktur, welche in einer Tabelle dargestellt werden kann⁴. Da dies im Kontext einer `StatisticTable` passiert, können im letzten Schritt die berechneten Daten angezeigt werden.

Um in den Funktionen bereits berechnete statistische Werte nutzen zu können, wird die Funktion vor ihrer Berechnung im Kontext dieser Werte evaluiert (Zeile 7).

⁴Siehe Abschnitt 3.3.2.

3 Datenanalyse und -aufbereitung

Die Methode `getStatisticsContext` liefert dabei eine Zeichenkette zurück. Eine Beispielausprägung mit den Werten für Minimum und Durchschnitt sieht so aus:

```
var context = this.getStatisticsContext();  
// context == "var min = 3; var avg = 13;"
```

Dies ist vor allem sinnvoll in Funktionen, die der Nutzer selbst definiert. Ein Anwendungsfall dafür ist die Berechnung der Standardabweichung, in der man den Durchschnitt aller Werte benötigt.

Bestätigt der Nutzer die Eingabe einer neuen Funktion, wird ein *Callback* ausgeführt. Dieser bekommt als Argument eine Zeichenkette, die die eingegebene Funktion beinhaltet. Er ist wie folgt implementiert:

```
1 function(functionString) {  
2     // evaluate the function in context of statistical values  
3     eval(this.getStatisticsContext() + " var heuristicFunction = " +  
         functionString);  
4  
5     var item = {  
6         heuristic: heuristicFunction.name,  
7         value: heuristicFunction(columnData)  
8     }  
9  
10    this.data.push(item);  
11    this.functionSet.push({  
12        name: heuristicFunction.name,  
13        calculation: heuristicFunction  
14    });  
15    this.update(this.data);  
16 }.bind(this);
```

Auch hier wird in Zeile 3 die neue Funktion in den Kontext der statistischen Werte gesetzt. Die Evaluierung der Funktion folgt in Zeile 7. Um die Statistik in der Tabelle anzeigen zu können, wird in Zeile 5 ein neuer Eintrag erzeugt. Er besteht aus dem Namen der Funktion, die der Nutzer eingegeben hat, und ihrem Wert, berechnet über den aktuellen Daten. Die Variable `this.data` enthält in diesem Moment ein Array aus Tupeln der bereits berechneten Statistiken. Das neu erzeugte Tupel wird an dieses Array angehängt (Zeile 10) und die Tabelle anschließend aktualisiert.

Es ist problemlos möglich, die Daten, auf denen die Statistiken berechnet wurden, zu ändern und die Statistiken neu zu berechnen. In der Instanzvariable `functionSet` befinden sich initial die vordefinierten statistischen Funktionen. Fügt der Nutzer eigene Funktionen hinzu, so werden diese in die Liste aufgenommen, wie im *Callback* Zeile 11 zu sehen. Bei Änderung der Eingabedaten kann nun die Methode `createStatistics` erneut aufgerufen werden. Hier werden jetzt automatisch auch die vom Nutzer hinzugefügten Funktionen mitberechnet. Da sie in der Reihenfolge aufgerufen werden, in der sie auch definiert wurden, können Abhängigkeiten von zuvor berechneten Kennwerten problemlos aufgelöst werden.

3.4 Daten transformieren

Bis hierhin wurden in diesem Kapitel Techniken vorgestellt, um die Daten zu importieren und zu erkunden. Im folgenden Teil soll es nun darum gehen, wie sie entsprechend der Bedürfnisse aufbereitet werden können. Für die dafür nötigen Transformationsschritte gibt es zwei Komponenten: Die *Table* und das *Script*.

3.4.1 Transformationen in einer Tabelle

- **Sortierung** Die Daten können nach beliebigen Attributen sortiert werden. Die Sortierung kann zum einen das Finden von bestimmten Datenelementen vereinfachen. Hat man eine Datenstruktur bestehend aus einer Liste von Ländern und deren Einwohnerzahlen, kann man sie nach den Ländernamen ordnen. Anschließend fällt es leichter ein bestimmtes Land darin zu entdecken. Zum anderen wird eine Sortierung auch den folgenden Datenfluss beeinflussen, denn die geordneten Daten werden an die nächste Komponente weitergereicht.
- **Umbenennung** Spalten in der Tabelle können beliebig umbenannt werden. Wie bei der Sortierung werden auch hier die veränderten Daten an die nächste Komponente weitergereicht. Die Umbenennung ist zum einen nützlich, um unintuitive Attributnamen zu vermeiden. So enthalten die Fußballdaten zum Beispiel das Attribut *FTHG*, welches die Tore der Heimmannschaft angibt. Findet der Nutzer diesen Namen nicht eingängig, kann er ihn ändern. Die Umbenennung kann zum anderen auch vor einer *join*-Operation sinnvoll sein, wenn Datensätze über ein gemeinsames Attribut miteinander verbunden werden sollen. Davon wurde in Abbildung 3.2 Gebrauch gemacht. Näheres dazu folgt im Abschnitt 3.4.4.
- **Selektion** Ziel der Selektion ist es, Attribute aus den Daten zu entfernen und so lediglich die für den konkreten Anwendungsfall benötigten Daten übrig zu behalten. Um dies zu erreichen, besteht die Möglichkeit Spalten aus der Tabelle zu löschen. Diese Operation führt dazu, dass das Spaltenattribut aus den weitergereichten Daten entfernt wird.

Die Tabelle bietet Transformationsschritte an, die ohne Programmierung vollzogen werden können. Das kann den Arbeitsablauf gegenüber der programmatischen Datentransformation beschleunigen. Voraussetzung dafür ist, dass die Daten geeignet in einer Tabelle angezeigt werden können⁵.

Im Fußball-Wetter-Beispiel auf Seite 29 wurde im Schritt (c) von den Möglichkeiten Gebrauch gemacht, die die Tabelle bietet. Es wurden sowohl überflüssige Spalten gelöscht als auch passendere Namen vergeben.

⁵Näheres dazu in Abschnitt 3.3.2 auf Seite 37.

3.4.2 Implementierung

Aus Performancegründen ist es nicht praktikabel, dass jede Komponente die Eingabedaten komplett kopiert, auf den kopierten Daten ihre Transformationsschritte durchführt und diese Daten an die nächste Komponente weiterreicht. Die Alternative ist, mit Referenzen zu arbeiten. Dabei kann es jedoch zu unerwarteten Seiteneffekten kommen, was gerade bei der oben beschriebenen Selektion problematisch ist. Im impliziten Datenfluss scheint es so zu sein, dass eine weiter unten liegende Komponente eine höhere nicht beeinflusst, da sie ihre veränderten Daten nur nach unten weitergeben kann. Werden jetzt aber in einer Tabelle bestimmte Attribute aus den Daten gelöscht, so können das unter Umständen dieselben Daten sein, auf denen auch andere Komponenten arbeiten. Das liegt daran, dass immer nur Referenzen auf die kompletten Daten weitergereicht werden. Die Attribute würden aus den referenzierten Daten gelöscht und wären so auch für darüber liegende Komponenten nicht mehr verfügbar. Diese Seiteneffekte gilt es möglichst zu vermeiden. Um das zu erreichen, wurde sich dafür entschieden, dass jede Komponente selbst dafür verantwortlich ist, entsprechende Kopieroperationen durchzuführen. Die Umsetzung des Löschens einer Spalte wird hier als Beispiel für diese Problematik vertieft. Dabei speichert sich die Tabelle den entsprechenden Spaltennamen in einem Array namens `removedColumns`. In der anschließend ausgelösten Aktualisierung führt sie die nötigen Kopieroperationen durch und entfernt diese Spalten aus den duplizierten Daten.

Um Seiteneffekte auf den vorausgehenden Datenfluss zu vermeiden, führt die Tabelle ein `map`⁶ auf den Daten durch (Quelltext 3.3, Zeile 3). Dabei wird pro Datum ein neues Objekt erzeugt, welches die Attribute enthält, die nicht gelöscht wurden (Zeile 7). Die Komponenten darüber werden also nicht vom Löschen beeinflusst.

Quelltext 3.3: Löschen einer Spalte in der Tabelle

```

1 removeColumns(data) {
2   var _this = this;
3   this.component.data = data.map(function(ea) {
4     var item = {};
5     Properties.own(ea).each(function(key) {
6       // if key isn't from a removed attribute, add it to the new item
7       if (_this.removedColumns.indexOf(key) < 0) {
8         item[key] = ea[key];
9       }
10    });
11    return item;
12  });
13  return this.component.data;
14 }
```

⁶https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map; besucht am 01. Oktober 2014.

3.4.3 Transformationen in einem Script

Eines der Projektziele ist, den Entwickler in seinen Möglichkeiten nicht einzuschränken. Demnach muss es auch einen Weg geben Transformationen durchzuführen, die von der Tabelle nicht abgedeckt werden. Genau diese Aufgabe übernimmt das *Script*. Dort kann beliebiger Code ausgeführt werden (siehe Abschnitt 2.3.4, Seite 17). Welche Probleme damit gelöst werden können, ist auf Seite 29 in Schritt (d) ersichtlich.

Aus den Wetterdaten⁷ gilt es die Stadt, das Datum und die Temperatur zu extrahieren. Dazu wird die im vorherigen Abschnitt bereits erwähnte `map`-Funktion von JavaScript-Arrays wie folgt genutzt:

```

1 var minWeather = weather.map(function(ea) {
2   var item = {}, w = ea.weather;
3   item.city = ea.city;
4   item.date = w.date.mday + "." + w.date.mon + "." + w.date.year;
5   item.temp = w.tempm;
6   return item;
7 });

```

`weather` ist dabei das Array, das die ursprünglichen Wetterdaten enthält. Der Variablen `minWeather` werden die vereinfachten Daten zugewiesen. Diese haben die Struktur, die in Abbildung 3.2 in der Tabelle (e) zu erkennen ist. Mit ein bisschen Programmieraufwand können auf diese Art und Weise gezielt Attribute aus tiefer verschachtelten Datenstrukturen selektiert werden. Die Konkatenation mehrerer Datenfelder zu einer Zeichenkette ist in diesem Fall ebenfalls sehr nützlich. So kann eine Datumsangabe erzeugt werden, die mit der aus den Spieldaten übereinstimmt.

In Skripten muss beachtet werden, dass nur Referenzen auf die Daten existieren. Die Daten werden bei der Weitergabe von der vorherigen Komponente nicht kopiert. Wenn sie direkt manipuliert werden, hat das Seiteneffekte auf andere Komponenten, die auf den gleichen Daten arbeiten. Der Programmierer ist daher angehalten, möglichst funktional zu programmieren oder nötige Kopieroperationen selbst durchzuführen.

3.4.4 Join-Operation

Eine Visualisierung basiert oft auf mehreren Datenquellen. Deshalb sollte es eine Möglichkeit geben, Daten miteinander zu verbinden. Diese Aufgabe übernimmt die `join`-Funktion. Ihre Signatur ist wie folgt:

```
join(dataArrays, joinAttribute, optSubtreeNames)
```

Sie bekommt eine Liste von Datenquellen und Attributen und als dritten optionalen Parameter eine Liste von Namen für entstehende Teilbäume. Diese wird später im Detail beschrieben.

⁷Abbildung 3.4, Seite 31.



Abbildung 3.9: Einfacher Join, der die Ergebnis- und Wetterdaten zu einer flachen Datenstruktur verbindet

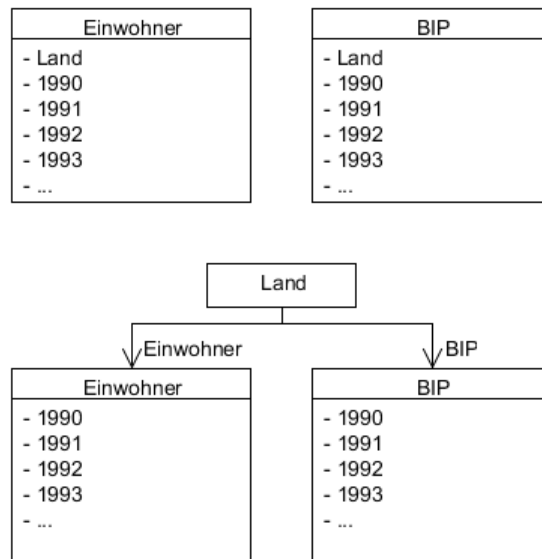


Abbildung 3.10: Erweiterter Join, welcher die Einwohnerzahlen und das Bruttoinlandsprodukt (BIP) einzelner Länder über verschiedene Jahre in einer Datenstruktur vereint

Eine Anwendung des Joins ist in Abbildung 3.9 zu sehen. Sie zeigt oben die Ausgangsdatenstruktur und unten die Struktur nach der *join*-Operation. Der Aufruf ist folgender:

```
join([Ergebnisse, Wetter], ["Stadt", "Datum"]);
```

Sollten Attribute mit gleichem Namen in mehreren Datensätzen vorkommen, so überschreiben sich diese Attribute gegenseitig. Nur einer der Werte wird in den zusammengeführten Daten vorkommen. Ist dies nicht erwünscht, so sollten die Attribute im Vorhinein umbenannt werden.

Einen weiteren Anwendungsfall für die *join*-Operation bieten die Daten in Abbildung 3.10. Sie geben für verschiedene Länder über verschiedene Jahre die Einwohnerzahlen und das Bruttoinlandsprodukt an. Diese Struktur ist zum Beispiel in den Daten von Hans Rosling vorzufinden [20]. Hier ist zu sehen, dass alle Attribute in beiden Datensätzen die gleichen Namen tragen. Ein Join der beiden Tabellen über das Attribut *Land* mit dem folgenden Aufruf führt nicht zu dem gewünschten Ergebnis.

```
join([Einwohner, BIP], ["Land"]);
```

Stattdessen werden alle Attribute der Tabelle *Einwohner* von denen in *BIP* überschrieben. Das Ergebnis ist also genau die *BIP*-Tabelle. Damit das nicht passiert, soll die Datenstruktur wie in Abbildung 3.10 unten entstehen. Ihr Aufbau ist folgender:

1. Jedes Land hat jeweils eine Referenz auf *Einwohner* und eine auf *BIP*.
2. *Einwohner* und *BIP* halten die jeweiligen Daten für die verschiedenen Jahre.

Einwohner und *BIP* können in dieser Struktur als Teilbäume eines jeden Datums gesehen werden. Aus diesem Grund heißt der dritte, optionale Parameter der *join*-Funktion *optSubtreeNames*. Ist hier ein Array mit Strings angegeben, so wird ein erweiterter Join durchgeführt. Die Teilbäume sind dann jeweils mit den *SubtreeNames* referenzierbar. Der korrekte Aufruf, der die angestrebte Datenstruktur erzeugt, lautet:

```
join([Einwohner, BIP], ["Land"], ["Einwohner", "BIP"]);
```

Es ist nicht automatisch möglich, aus den Daten geeignete *SubtreeNames* zu extrahieren, da sie lediglich Zuordnungen von Jahreszahlen auf Datenwerte enthalten, nicht aber Tabellennamen wie „Einwohner“. Aus diesem Grund müssen die *SubtreeNames* explizit angegeben werden.

3.4.5 Aggregation von Daten

In den Diagrammen von Hans Rosling [20] ist es möglich, sich Datenwerte – wie die Bevölkerung für ganze Kontinente – anzuzeigen. Mit einem Klick auf das Symbol für den Kontinent kann dieser in seine einzelnen Länder zerlegt und sich diese einzeln angeschaut werden. Die Rückrichtung ist auch möglich. Der Kontinent

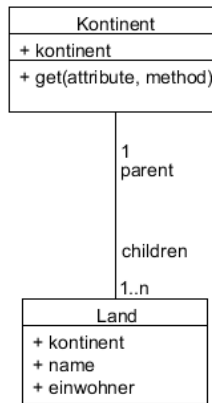


Abbildung 3.11: Schematischer Aufbau von Daten nach Anwendung der Aggregationsfunktion

wird dabei in seine einzelnen Bestandteile aufgeteilt, das heißt in einer anderen Detailstufe angezeigt. Um solche Effekte umzusetzen, ist eine geeignete Datenstruktur im Hintergrund nötig. In der aktuellen Struktur existieren nur Objekte für Länder, nicht aber für Kontinente.

Die Ausgangsdaten sind Länder mit ihrem Namen, einem Kontinent und einer Einwohnerzahl. Die Grundidee ist es, diese Daten nach dem Attribut *kontinent* zu gruppieren und dabei das Attribut *einwohner* aufzusummieren. Um dies zu erreichen, ist der folgende Aufruf nötig:

```
var kontinente = aggregateBy(staaten, "kontinent");
```

Dabei ist *staaten* die Liste von den oben beschriebenen Ländern. Um die Ansicht zwischen Kontinenten und Ländern wechseln zu können, ist eine geeignete Navigierbarkeit in den Daten nötig. Deshalb wird durch das Aggregieren eine Baumstruktur aufgebaut, die dies ermöglicht (siehe Abbildung 3.11). Diese hat auf der unteren Ebene eine Menge von Ländern, die jeweils eine Referenz auf ihr zugehöriges Kontinent-Objekt haben (*parent*). Die Kontinente haben neben ihrem Namen eine Liste von Ländern (*children*) und eine Methode *get*. Diese kann genutzt werden, um auf aggregierte Attribute der Ausgangsdaten zuzugreifen. Die Signatur ist wie folgt:

```
get(propertyName, optAggregationFunction);
```

Der Parameter *propertyName* gibt den Namen des gewünschten Attributs an, *optAggregationFunction* ist entweder „sum“, „avg“ oder eine beliebige, vom Nutzer erstellte Funktion. Ist sie nicht angegeben, wird „sum“ verwendet. *get* berechnet die gewählte Aggregation auf dem entsprechenden Attribut aller Kinder und gibt diesen Wert zurück. Um die Einwohner des ersten Kontinents in der Liste abzurufen, ist folgender Aufruf nötig.

```
continents[0].get("einwohner", "sum");
```

Eine solche Datenstruktur kann bei der Erstellung von Diagrammen wie der *Gapminder World* [20] genutzt werden. Bei einem Klick auf einen Kreis, der einen Kontinent darstellt, werden statt des Eltern-Knotens dann die Kind-Knoten (Länder) angezeigt. Das funktioniert auch in der Rückrichtung. Auf diese Art und Weise können visuelle Elemente separiert und aggregiert werden.

3.4.6 Entitäten-Modell

Der Inhalt dieses Abschnitts ist durch Commit-Daten motiviert, die von der Github-API⁸ abgefragt werden können. Sie sind folgendermaßen aufgebaut:

```

1 var data = [
2   {
3     author: {name: "author1"},
4     id: "commitsha1",
5     files: [{name: "file1"}, {name: "file2"}, {name: "file3"}]
6   },{
7     author: {name: "author2"},
8     id: "commitsha2",
9     files: [
10      {name: "file2"},
11      {
12        name: "file4",
13        commitSpecificInfo: 1
14      }
15    ]
16  },{
17    author: {name: "author3"},
18    id: "commitsha3",
19    files: [
20      {
21        name: "file4",
22        commitSpecificInfo: 2
23      }
24    ]
25  },{
26    author: {name: "author3"},
27    id: "commitsha4",
28    files: [{name: "file5"}]
29  }
30 ]

```

Es existiert pro Commit ein Objekt. Jeder Commit hat einen Autor, eine ID und eine Liste an Dateien, die geändert wurden.

Problemstellung

JSON-Daten, wie sie hier vorliegen, haben einige Einschränkungen:

⁸<https://api.github.com/>; besucht am 01. Oktober 2014.

- Es ist nicht ohne Weiteres möglich, *zyklische Datenstrukturen* zu beschreiben. Die Daten sind hierarchisch aufgebaut.
- Eine Folge davon ist, dass die *Navigierbarkeit* von Elementen einer tieferen Hierarchiestufe zu denen einer höheren nicht gegeben ist. Ohne über alle Daten zu iterieren, kann im Beispiel mit der bloßen Referenz auf ein `file`-Objekt programmatisch nicht ermittelt werden, welche Autoren diese Datei geändert haben. Es fehlen dafür Referenzen von `files` auf `author`.
- Objekte, die *mehrfach referenziert* werden, existieren in Form von *mehreren Instanzen* in den Daten, obwohl sie semantisch das gleiche Datenelement darstellen. Ein Beispiel dafür ist `file4`, welches in zwei verschiedenen Commits vorkommt und dabei von zwei verschiedenen Objekten repräsentiert wird.

Daten-Entitäten

Um die beschriebenen Probleme zu überwinden, wurde das Modell der *Entities* eingeführt. Damit wird eine vollständige Navigierbarkeit zwischen den Datenelementen gewährleistet, was auch Zyklen beinhaltet. Das ist möglich, da die notwendigen Referenzen ergänzt werden. Weiterhin wird dafür gesorgt, dass mehrfach referenzierte Objekte nur einmal existieren. Bei der Erstellung der *Entities* werden identische Objekte erkannt und zu einer Instanz zusammengefasst.

Hat der Programmierer eine Entität für `files` erzeugt, die beispielsweise den Namen *File* trägt, kann er mit den folgenden Zeilen alle Autoren herausfinden, die eine Datei mit dem Namen `file4` bearbeitet haben.

```
var authors = File.getAll().find(function(file) {  
    return file.name == "file4";  
}).getCommits().pluck("author");
```

Hier wird ausgehend von der Menge aller `files` die Datei mit dem Namen `file4` gesucht. Anschließend werden alle Commits abgefragt, in denen diese Datei bearbeitet wurde. Von diesen Commits können nun die Autoren selektiert werden. In diesem Beispiel wird die Navigierbarkeit von den Dateien über die Commits hin zu den Autoren sichtbar.

Um anhand von JSON-Daten Entitäten zu erzeugen, kann die *EntityFactory* genutzt werden.

```
var Factory = new lively.morphic.Charts.EntityFactory();  
var Commit = Factory  
    .createEntityTypeFromList("Commit", data, "id");
```

In diesem Quellcode wird zunächst eine *Factory* erzeugt. Diese erstellt ausgehend von den Daten eine Entität namens *Commit*. Die ersten beiden Parameter beinhalten den Namen der Entität und die Ausgangsdaten. Der dritte gibt den eindeutigen Bezeichner für ein Commit-Objekt an. Dieser ist dafür nötig, um zu unterscheiden, ob es sich bei zwei Commit-Objekten um verschiedene Elemente handelt.

Aus dieser Entität können nun weitere Entitäten extrahiert werden. So führt der folgende Aufruf zur Erzeugung der Entitäten *Author* und *File*, die beide aus *Commit* hervorgehen.

```
var Author = Commit
    .extractEntityFromAttribute("Author", "name", "author");
var File = Commit
    .extractEntityFromList("File", "name", "files");
```

- Ein `Author` ist in den `Commits` unter dem Attribut `author` zu finden und wird durch `name` eindeutig identifiziert.
- Ein `File` befindet sich in den `Commits` *in einer Liste* unter dem Attribut `files`. Es wird eindeutig durch `name` gekennzeichnet.

Für diese Aufgabe stehen zwei verschiedene Methoden zur Verfügung.

- `extractEntityFromAttribute` erwartet, dass das Zielobjekt als einzelnes Objekt unter dem gegebenen Attribut zu finden ist.
- `extractEntityFromList` hingegen sucht die Zielobjekte in einer Liste, die von dem angegebenen Attribut referenziert wird.

Mit der Erzeugung der Entitäten werden doppelte Objekte erkannt und zusammengefasst. Zusätzlich wird ihnen eine Zugriffsmethode hinzugefügt, welche es erlaubt, auf höhere Hierarchiestufen zuzugreifen. Auf einer Instanz der `File`-Entität kann beispielsweise `getCommits()` aufgerufen werden, da die Dateien aus den `Commits` extrahiert wurden. Es wird eine Liste mit `Commits` zurückgegeben, welche diese Datei referenzieren. Auf diese Weise kann über die Datenstrukturen navigiert werden.

Entities haben ein Verständnis davon, wie verschiedene Datenelemente zusammenhängen. Das ist dadurch möglich, dass der Nutzer bei ihrer Erstellung zusätzliche Informationen wie zum Beispiel IDs angibt. Somit kann der Nutzer auf einer höheren Abstraktionsebene mit den Daten arbeiten.

3.5 Zusammenfassung und Ausblick

3.5.1 Zusammenfassung

Datenquellen im Internet sind heterogen. Das heißt, die Datenstruktur ist oft unvorhersehbar und auch die Dateiformate sind unterschiedlich. Das führt dazu, dass vor der Erstellung einer Visualisierung die Daten erst einmal aufbereitet werden müssen. Oft muss man auf verschiedene Tools für verschiedene Schritte ausweichen. Bei Betrachtung des anfänglichen Beispiels mit den Fußballdaten wurden die Spieldaten (*csv*-Datei) zunächst mit *Microsoft Excel* geöffnet. Die gewünschten Spalten konnten anschließend markiert und in das Onlinetool *Mr. Data Converter* [11] kopiert werden, welches die Konvertierung nach JSON übernahm. Daraufhin wurden die Daten lokal als JSON-Datei abgespeichert. Auf diese Art und Weise ist der Arbeitsablauf unterbrechungsintensiv und dadurch zeitaufwendiger, als wenn

alle Aufgaben in einer Arbeitsumgebung erledigt werden können. Um die Kontextwechsel des Programmierers zu minimieren, bietet *Flower* Unterstützung sowohl beim Importieren und Konvertieren, bei der Analyse, als auch bei der Umformung der Daten. Dabei können Standardaufgaben per Mausinteraktion gelöst werden und so Programmieraufwand gespart werden.

Es existieren zudem Hilfsfunktionen, die beispielsweise dafür genutzt werden können, um mehrere Datenquellen zu verbinden. Durch den Einsatz von Skripten im Datenfluss ist es dem Nutzer möglich, auch beliebigen JavaScript-Code zu schreiben. Somit sollte es nicht dazu kommen, dass er für bestimmte Schritte das Tool wechseln muss.

Durch die Integration aller Datenaufbereitungsschritte in das Visualisierungstool ist es möglich, auch in späteren Schritten der Visualisierungserstellung noch die Ausgangsdaten anzupassen. Da diese Änderungen durch den Datenfluss propagiert werden, wirken sie sich direkt auf die Visualisierung aus. Somit hat der Programmierer unmittelbares Feedback zu seinen Datenanpassungen.

3.5.2 Verwandte Arbeiten

Microsoft Excel

Das Tabellenkalkulationsprogramm *Excel* stellt einige Möglichkeiten zur Datenaufbereitung und -analyse zur Verfügung. Zum einen können aus den Daten Standarddiagramme erzeugt werden, welche zum Beispiel zur Analyse der Werteverteilung genutzt werden können. Dies ist auch eine erstrebenswerte Funktion für *Flower*. Näheres dazu im Abschnitt „Zukünftige Arbeiten“.

Zum anderen können statistische Berechnungen auf Datenreihen durchgeführt werden. Dafür gibt es eine Reihe von vordefinierten Funktionen, die genutzt werden können. Es ist zusätzlich möglich, eigene Funktionen in der Programmiersprache *Visual Basic for Applications*, kurz *VBA*, zu implementieren und diese analog zu den vordefinierten zu nutzen. Funktionsergebnisse können in Zellen abgelegt werden. Auf diese Art und Weise ist es auch möglich, die vorhandenen Daten mit Ergebnissen von Berechnungen anzureichern und so neue Spalten hinzuzufügen. Hierbei sind Parallelen zu den deskriptiven Statistiken in *Flower* zu sehen.

Excel ermöglicht die Datenanalyse und die Erstellung von Standarddiagrammen und bietet dafür sehr mächtige Werkzeuge. Soll aus den Daten eine individuelle Visualisierung erstellt werden, können sie exportiert und in einem anderen Tool verwendet werden. *Flower* hingegen integriert die Datenanalyse und -aufbereitung in den Erstellungsprozess individueller Visualisierungen. Es ermöglicht die direkte Nutzung der Daten in den nächsten Schritten, was ihre Anpassung zu einem späteren Zeitpunkt erleichtert.

Datenanalyse mit R

R ist eine Programmiersprache und -umgebung für statistische Berechnungen und Diagramme. Sie beinhaltet unter anderem viele vordefinierte Funktionen aus dem Bereich der deskriptiven Statistik. Zudem bietet sie die Möglichkeit, mit nur einem Aufruf Diagramme wie zum Beispiel ein Histogramm zu erstellen. Viel Wert

wurde darauf gelegt, dass dem Nutzer sinnvolle Standards für die Eigenschaften der Diagramme gegeben werden. Dennoch behält er die volle Kontrolle über diese Parameter, wenn er sie explizit angeben möchte. Die Erstellung individueller Visualisierungen ist in *R* nicht vorgesehen. Die Benutzeroberfläche besteht vornehmlich aus einer Kommandozeile, in der Befehle geschrieben und ausgewertet werden können. Es ist aber auch möglich, ganze Skripte zu implementieren und auszuführen [12].

3.5.3 Zukünftige Arbeiten

Dateiexport

Da *Flower* weitreichende Datenumformungen und das Zusammenführen mehrerer Datenquellen ermöglicht, ist ein nächster Schritt, diese umgeformten Daten auch für eine spätere Verwendung zu speichern. Damit ist auch die reine Datenaufbereitung ein Anwendungsfall für *Flower*. Um das zu erreichen, soll der Nutzer in der Lage sein, Daten in sein lokales Dateisystem oder den Lively PartsBin [5] zu exportieren. Die Dateien können dann auch mit anderen Nutzern geteilt werden und wieder als Eingabe in einem Datenfluss dienen.

XML Integration

Der *DataImporter* ist in der Lage, gängige Dateiformate nach JSON zu konvertieren. Das Dateiformat *XML* ist dabei noch nicht mit inbegriffen. Da es jedoch in einigen Datenquellen vorkommt und zu den bekannteren Formaten gehört, ist die Unterstützung ein logischer Schritt.

Histogramme

In Abschnitt 3.3.3 wurden Möglichkeiten beschrieben, um Daten mit Hilfe von statistischen Kennwerten zu analysieren. Ein weiteres Werkzeug zur Analyse von Datenverteilungen sind Histogramme. Um ein Histogramm zu erstellen, müssen die Daten zunächst in Klassen eingeteilt werden. Anschließend wird für jede Klasse ein Balken erzeugt, dessen Höhe von der Anzahl der Elemente in dieser Klasse abhängt. Mit Hilfe dieser grafischen Darstellung lassen sich beispielsweise unterschiedliche Regionen in Daten erkennen. J. W. Tukey nutzt in seinem Artikel *Exploratory Data Analysis* [19] Histogramme, um verschiedene Altersgruppen der Bevölkerung in Vietnam abzugrenzen. Diese sind in Abbildung 3.12 dargestellt. Eine entscheidende Stellschraube ist dabei die Größe der Klassen, in welche die Elemente einsortiert werden. Daher sollte eine Implementierung die Klassengröße zum Beispiel mit einem Slider oder Spinner ändern können. Ein Histogramm sollte in der *Table* für einzelne Spalten erstellt werden können. Dies ist auch die Stelle, an der die deskriptiven Statistiken angezeigt werden können.

Im Zuge der Implementierung von Histogrammen sollte auch die Umsetzung weiterer verwandter Diagrammartentypen zur Datenerkundung in Betracht gezogen werden. Es handelt sich dabei beispielsweise um Linien- oder Kreisdiagramme, welche den Fokus auf andere Eigenschaften der Daten legen.

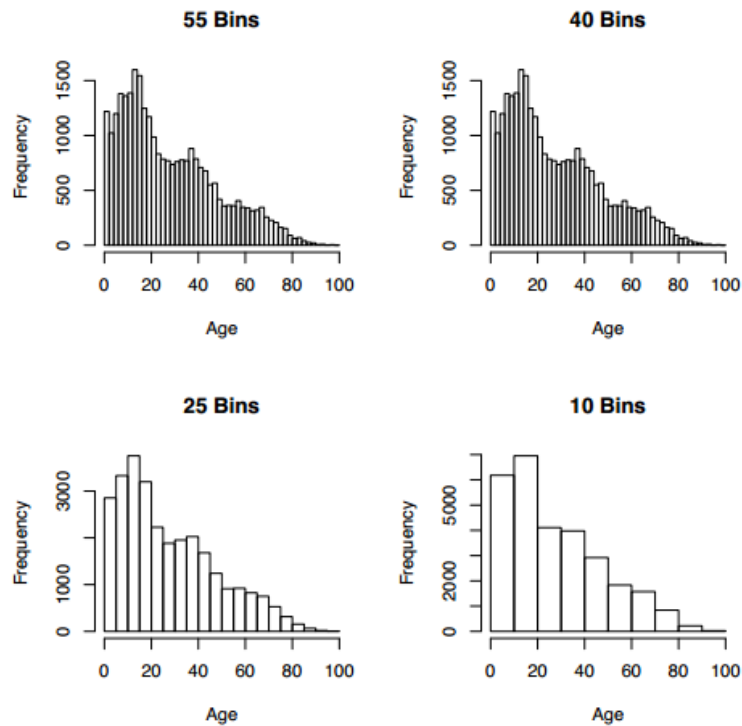


Abbildung 3.12: Histogramme mit identischen Daten und unterschiedlichen Klassengrößen. Quelle: J. W. Tukey [19]

Analyse und Transformationen im JSONViewer

Die Aktionen, die in einer Tabelle über das Kontextmenü auf den Kopfzellen möglich sind, sollten auch im *JSONViewer* funktionieren. Dies umfasst beispielsweise Transformationen wie das Löschen einzelner Attribute aus allen Datenelementen. Momentan ist für diese Aktion ein Skript notwendig, in dem über die Daten iteriert und die gewünschte Aktion durchgeführt wird. Das Schreiben einer Schleife benötigt aber deutlich mehr Zeit und ist auch fehleranfälliger. Hinzu kommt, dass es sich dabei auch um eine wiederkehrende Aufgabe handelt. Diese sollten dem Nutzer so weit wie möglich abgenommen werden. Zudem ist auch die Integration der deskriptiven Statistiken und der Analysediagramme, wie zum Beispiel von Histogrammen, an dieser Stelle möglich.

4 Abbildung von Daten auf visuelle Elemente

4.1 Einleitung

4.1.1 Motivation

Für das Untersuchen und Verstehen großer und meist unbekannter Datenmengen sind insbesondere Visualisierungen der Daten geeignet. Statt die Daten in ihrer blanken Form als Zahlen oder Text zu inspizieren, geben grafische Darstellungen oft eine übersichtlichere Auskunft. Schlüsse aus den Daten können somit leichter gezogen und Zusammenhänge sowie Korrelationen leichter entdeckt werden.

Bei der explorativen Analyse von Daten entstehen viele verschiedene Visualisierungen, die unterschiedliche Aspekte darstellen können. Um diesen Prozess zu erleichtern, gestaltet *Flower* die Abbildung von Daten auf visuelle Elemente möglichst einfach, intuitiv und schnell. Ein integraler Bestandteil ist dabei der *MorphCreator*, der den Nutzer bei genau dieser Abbildung unterstützt. Genauere Details dazu finden sich in Abschnitt 4.3. Auf Interaktionskonzepte im *MorphCreator* wird in Kapitel 5 eingegangen.

Um einen Einblick in den Prozess der Visualisierung zu erlangen, werden in den folgenden Abschnitten Beispielvisualisierungen untersucht. In Abschnitt 4.2 wird auf Grundlagen der Visualisierung eingegangen.

4.1.2 Untersuchung beispielhafter Visualisierungen

Für ein grundlegendes Verständnis für den Prozess der Visualisierung werden zwei beispielhafte Visualisierungen untersucht. Quelltext 4.1 zeigt einen Datensatz [35] in seiner JSON-Darstellung [9]. Er enthält den Kontinent, die Einwohnerzahl, den Energieverbrauch und die Arbeitslosenrate zu einzelnen Ländern der Welt. Die tabellarische Darstellung ist in Tabelle 4.1 zu sehen.

Zu diesem Datensatz können unterschiedliche Visualisierungen konstruiert werden, die unterschiedliche Aspekte betonen.

- a) *Abbildung 4.1* zeigt ein Balkendiagramm, welches die Einwohnerzahl der einzelnen Länder darstellt. Der Kontinent, der Energieverbrauch und die Arbeitslosenrate sind hingegen nicht abgebildet. Die einzelnen Länder werden durch Rechtecke symbolisiert, deren Höhen proportional zur Einwohnerzahl der Länder sind. Weiterhin sind die Rechtecke mit den jeweiligen Ländernamen beschriftet.

Quelltext 4.1: Auszug des referenzierten Datensatzes zu verschiedenen Ländern in JSON-Darstellung

```

1 var countries = [
2   // ...
3   {
4     "population" : 82516297,
5     "name" : "Germany",
6     "unemployed" : 4.800000191,
7     "continent" : "Europe",
8     "energy" : 331169028
9   },
10  // ...
11 ]

```

Tabelle 4.1: Tabellarischer Auszug des referenzierten Datensatzes zu verschiedenen Ländern

Land	Kontinent	Einwohner	Energieverbrauch (Öleinheiten)	Arbeitslosigkeit (%)
Australien	Australien	21 119 988	119 755 172	0,699 999 988
Dänemark	Europa	5 469 130	19 762 281	0,699 999 988
Deutschland	Europa	82 516 297	331 169 028	4,800 000 191
Finnland	Europa	5 290 431	36 795 024	1,5
Kanada	Nordamerika	32 977 334	271 731 208	0,400 000 006
Österreich	Europa	8 309 783	33 389 497	1,200 000 048

Tabelle 4.2: Gegenüberstellung der genutzten visuellen Variablen und deren Abhängigkeiten

Visualisierung	Visuelles Attribut	Datenattribut	
Abbildung 4.1	Farbe	—	} dateninvariant
	Breite	—	
	y-Koordinate	—	
	x-Koordinate	Index	
Abbildung 4.2	Höhe	Einwohnerzahl	} datenvariant
	Beschriftung	Kontinent	
	x-Koordinate	Energieverbrauch	
	y-Koordinate	Arbeitslosenrate	
	Radius	Einwohnerzahl	
	Farbe	Kontinent	
	Tooltip	Kontinent	

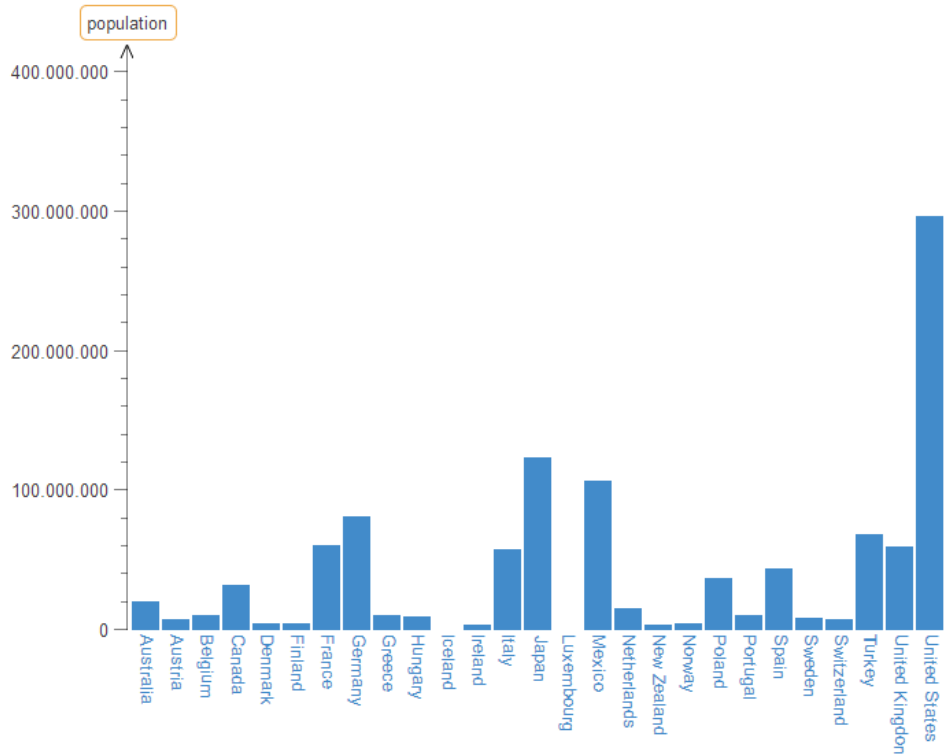


Abbildung 4.1: Ein Balkendiagramm für die Einwohnerzahl verschiedener Länder

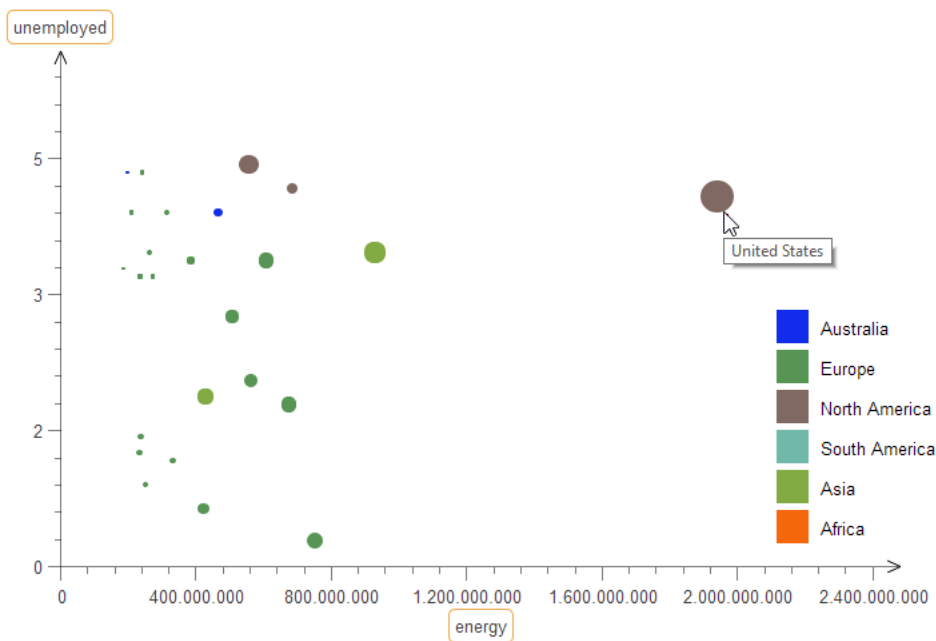


Abbildung 4.2: Ein von *Gapminder World* [20] inspiriertes Diagramm (im Folgenden auch *GW-Diagramm* genannt)

Was bei dem Diagramm beispielsweise schnell auffällt, ist die hohe Einwohnerzahl der USA.

- b) In *Abbildung 4.2* (im Folgenden auch *GW-Diagramm* genannt) hingegen, wird sowohl die Einwohnerzahl als auch der Energieverbrauch und die Arbeitslosenrate dargestellt. Die einzelnen Länder werden durch Kreise symbolisiert, deren Radius proportional zur Einwohnerzahl ist. Die x-Koordinate der Kreise ist ein Maß für den Energieverbrauch des Landes, während die y-Koordinate ein Maß für die Arbeitslosenquote des Landes ist. Die Farben der Kreise dienen zusammen mit einer Legende der Zuordnung zu dem jeweiligen Kontinent. Zu den einzelnen Kreisen existieren aus Platzgründen keine Beschriftungen, die das repräsentierte Land erläutern. Stattdessen wurden *Tooltips* verwendet, welche erscheinen, sobald der Nutzer den Cursor über die Kreise bewegt. Auf die Erstellung der *Tooltips* wird im späteren Kapitel 5 eingegangen.

In der Visualisierung ist, ähnlich wie im Balkendiagramm, erkennbar, dass die Vereinigten Staaten eine vergleichsweise große Einwohnerzahl haben. Zusätzlich lässt sich schnell entnehmen, dass der Energieverbrauch der USA am größten ist und die Arbeitslosenquote ebenfalls vergleichsweise hoch ist.

Eine Gegenüberstellung der beiden Visualisierungen hinsichtlich ihrer genutzten visuellen Attribute ist in *Tabelle 4.2* zu finden.

Auch wenn beide Visualisierungen im Aufbau verschieden sind, liegt ihnen dennoch der gleiche Datensatz zugrunde. Zusätzlich lassen sich weitere Gemeinsamkeiten feststellen:

- Jedem Datenelement (und somit jedem Land) wird ein visuelles Element zugewiesen (Rechteck beziehungsweise Kreis).
- Es existieren Eigenschaften visueller Elemente, die unabhängig von den Datenelementen sind (siehe *Tabelle 4.2*, oberer Abschnitt). Sie sind *dateninvariant*.
- Es existieren Eigenschaften visueller Elemente, die durch die Datenelemente geprägt sind (siehe *Tabelle 4.2*, unterer Abschnitt). Jene sind *datenvariant*.

Es liegt demnach eine Abbildung von Datenelementen auf visuelle Elemente vor. Die unterschiedlichen Ausprägungen der grafischen Elemente können dabei direkt von den Daten abhängen.

4.2 Hintergrund

Im vorherigen Abschnitt wurde anhand von Beispielvisualisierungen festgestellt, dass visuelle Elemente anhand verschiedener Eigenschaften variiert werden können. Das zugrunde liegende Konzept wird im Abschnitt „Visuelle Variablen“ vorgestellt. Anschließend wird darauf eingegangen, wie diese vom Menschen wahrgenommen werden und wie sie mit Hilfe von Skalen und Legenden dargestellt werden können.

4.2.1 Visuelle Variablen

Für die Abbildung von Daten auf visuelle Elemente stehen verschiedene *visuelle Variablen* zur Verfügung. Eine visuelle Variable lässt sich dabei wie folgt definieren:

„A visual variable is a visual property that can be varied in order to convey (encode) information.“ [38]

Übersetzt bedeutet dies, dass eine visuelle Variable eine visuelle Eigenschaft ist, welche variiert werden kann, sodass Informationen ausgedrückt (kodiert) werden können.

So wurde zum Beispiel im *GW-Diagramm* von der Position, Farbe und Radius eines Kreises Gebrauch gemacht. Eine Auflistung aller visuellen Variablen [39] mit exemplarischen Ausprägungen findet sich in Abbildung 4.3.





























Form				
Position				
Größe				
Helligkeit				
Farbe				
Orientierung				
Textur				

Abbildung 4.3: Visuelle Variablen nach Bertin [39] mit beispielhaften Ausprägungen

Für die geometrische *Form* der visuellen Elemente lassen sich beliebig komplexe Konstrukte erstellen, indem verschiedene Basiselemente miteinander kombiniert werden. So besteht in Abbildung 4.3 eine Beispielform aus einer Ellipse, einem gefüllten Pfad und einem Text. Zu den typischen Basiselementen gehören Rechtecke, Ellipsen, Linien, Texte und Pfade.

4.2.2 Wahrnehmungstypen

Im vorhergehenden Abschnitt wurden die verschiedenen visuellen Variablen vorgestellt, welche für eine Visualisierung genutzt werden können. Bei der Auswahl der passenden visuellen Variablen kann es helfen, zu prüfen, ob sie für den Anwendungsfall geeignet ist. Nach Bertin [39] existieren verschiedene Eigenschaften, die visuelle Variablen erfüllen können.

Selektiv Bei einer selektiven visuellen Variablen ist es möglich, alle Instanzen einer gegebenen Ausprägung gedanklich zu gruppieren. Es muss möglich sein, alle Ausprägungen bis auf die Zielausprägung zu ignorieren. Ein Beispiel ist in Abbildung 4.4 zu finden. Im linken Bereich kann kaum erkannt werden, welche Form die Buchstaben *W* bilden. Im rechten Bereich hingegen ist dies aufgrund der Farbe recht einfach für den Menschen zu erkennen.

Alle Variablen außer der Form sind selektiv.

Assoziativ Bei einer assoziativen Variablen ist es möglich, alle Variationen der Variablen gedanklich zu gruppieren. Die Variationen der Variablen dürfen dabei nicht die Sichtbarkeit der jeweiligen Elemente einschränken. In Abbildung 4.5 sind drei Blöcke aus Kreisen sichtbar. Die Kreise im linken Block bilden ein uniformes Gitter. Bei Variation der Helligkeit (mittlerer Bereich) können nicht alle Elemente des Gitters mühelos wahrgenommen werden, da die sehr hellen Kreise kaum sichtbar sind. Die Helligkeit ist deshalb keine assoziative Variable. Bei Variation der Farbe hingegen (rechter Bereich) ist die Wahrnehmung nicht eingeschränkt. Die Farbe ist deshalb eine assoziative Variable.

Weiterhin sind Orientierung, Textur, Form und Position assoziativ. Nicht assoziativ ist neben der Helligkeit auch die Größe, da sehr kleine visuelle Elemente ähnlich schwer wahrgenommen werden.

Quantitativ Wenn der Unterschied zweier Ausprägungen einer visuellen Variablen numerisch interpretiert werden kann, so ist die visuelle Variable *quantitativ*. So kann in Abbildung 4.6 leicht erkannt werden, dass das erste schwarze Rechteck halb so hoch ist wie das zweite schwarze Rechteck. Eine ähnlich quantitative Einschätzung zwischen den verschiedenen farbigen beziehungsweise hellen Rechtecken ist hingegen nicht möglich.

Position und Größe sind die einzigen quantitativen Variablen.

Geordnet Wenn die Variable einer natürlichen, intuitiven Ordnung unterliegt, ist es eine *geordnete* Variable. In Abbildung 4.6 ist zum Beispiel das erste schwarze Rechteck eindeutig *kleiner* als das zweite. Weiterhin ist das erste graue Rechteck eindeutig *dunkler* als das zweite. Zwischen dem blauen und gelben Rechteck hingegen kann intuitiv keine derartig geordnete Relation festgestellt werden.

Position, Größe und Helligkeit sind geordnet.

Eine Übersicht, welche Variablen welche Eigenschaften haben, findet sich in Tabelle 4.3.

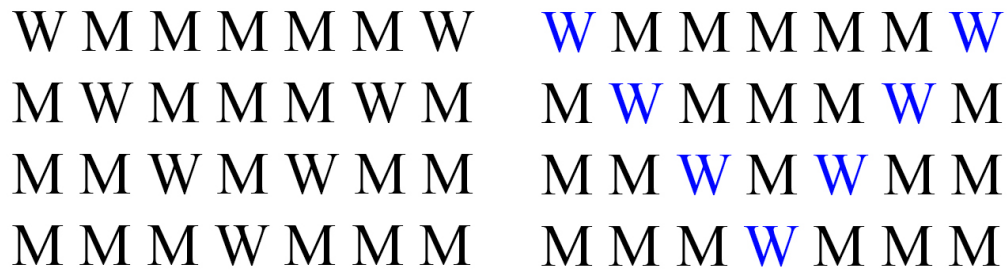


Abbildung 4.4: Beispiel für die nicht selektive Variable *Form* und für die selektive Variable *Farbe*

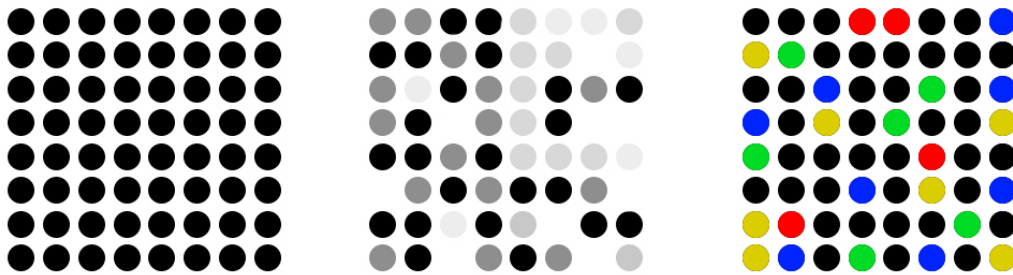


Abbildung 4.5: Beispiel für Assoziativität visueller Variablen

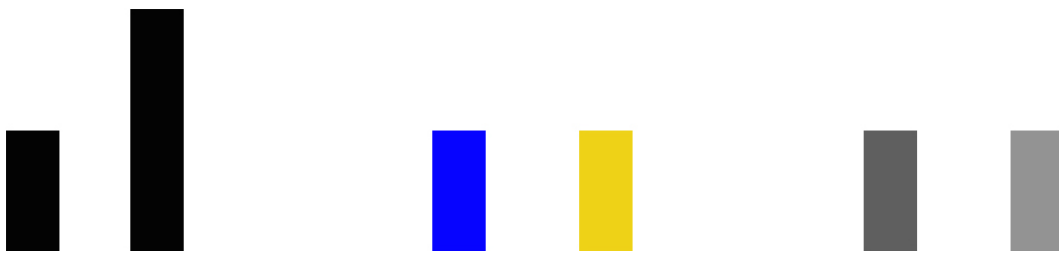


Abbildung 4.6: Beispiel für Quantität und Ordnung visueller Variablen

Tabelle 4.3: Übersicht zu den Eigenschaften visueller Variablen

Variable	Selektiv	Assoziativ	Quantitativ	Geordnet
Form	nein	ja	nein	nein
Position	ja	ja	ja	ja
Größe	ja	nein	ja	ja
Helligkeit	ja	nein	nein	ja
Farbe	ja	ja	nein	nein
Orientierung	ja	ja	nein	nein
Textur	ja	ja	nein	nein

4.2.3 Skalen und Legenden

Häufige Hilfsmittel (siehe Beispielvisualisierungen [21]) für das Verständnis einer Visualisierung sind Skalen und Legenden. In erster Linie können damit zwei Aspekte erläutert werden:

1. Die Beschriftung der Skalen/Legenden verdeutlicht, welche Datenelemente auf welche visuellen Variablen abgebildet wurden. Dadurch kann im *GW-Diagramm* (Seite 59) anhand der Beschriftung *unemployed* erkannt werden, dass die y-Achse die Arbeitslosenquote abbildet.
2. Weiterhin ist es möglich, die Abbildungen auf die (ungefähren) Datenwerte zurückzuführen, die der Grafik zugrunde liegen. Dadurch kann im *GW-Diagramm* beispielsweise erkannt werden, dass die höchsten Arbeitslosenquoten bei etwa 5% liegen.

Es folgen mehrere Beispiele, die unterschiedliche Szenarien und die dabei genutzten Skalen und Legenden darstellen.

Skalen für Position In der bereits bekannten Visualisierung auf Abbildung 4.2 (Seite 59) werden Länder auf Kreise abgebildet. Die x-Koordinate der Kreise ist dabei von der Einwohnerzahl und die y-Koordinate von der Arbeitslosenrate abhängig. Dies kann der Betrachter anhand der beschrifteten x- und y-Achse erkennen. Weiterhin können die Achsen verwendet werden, um den einzelnen Ländern konkrete Werte zuzuordnen.

Skalen für Größen mit variabler Position Bei dem *GW-Diagramm* wurde der Radius der Kreise genutzt, um die Einwohnerzahl der Länder darzustellen. Um es dem Betrachter zu ermöglichen, den Radius auf konkrete Werte abzubilden, wird eine neue Skala hinzugefügt. Das Resultat ist rechts oben in Abbildung 4.7 zu sehen.

Skalen für Größen mit konstanter Position Bei einem simplen Balkendiagramm, wie es in Abbildung 4.1 (Seite 59) zu sehen ist, befinden sich die unteren Kanten aller Rechtecke auf der gleichen y-Koordinate. Dadurch kann die y-Achse als Skala genutzt werden, um die unterschiedlichen Höhen der Rechtecke zu symbolisieren.

Legenden für diskrete Farbspektren Im *GW-Diagramm* wurden Farben verwendet, um die zu den Ländern gehörenden Kontinente zu kennzeichnen. Somit erhält jeder Kontinent eine eigene charakteristische Farbe, die sich eindeutig von den anderen unterscheiden lässt. Um dem Betrachter die Möglichkeit zu geben, diese Zuordnung nachzuvollziehen, wird eine diskrete Legende (siehe rechter unterer Teil der Abbildung) hinzugefügt.

Legenden für kontinuierliche Farbspektren Wie die Verwendung eines kontinuierlichen Farbspektrums aussehen kann, veranschaulicht Abbildung 4.8. Hier

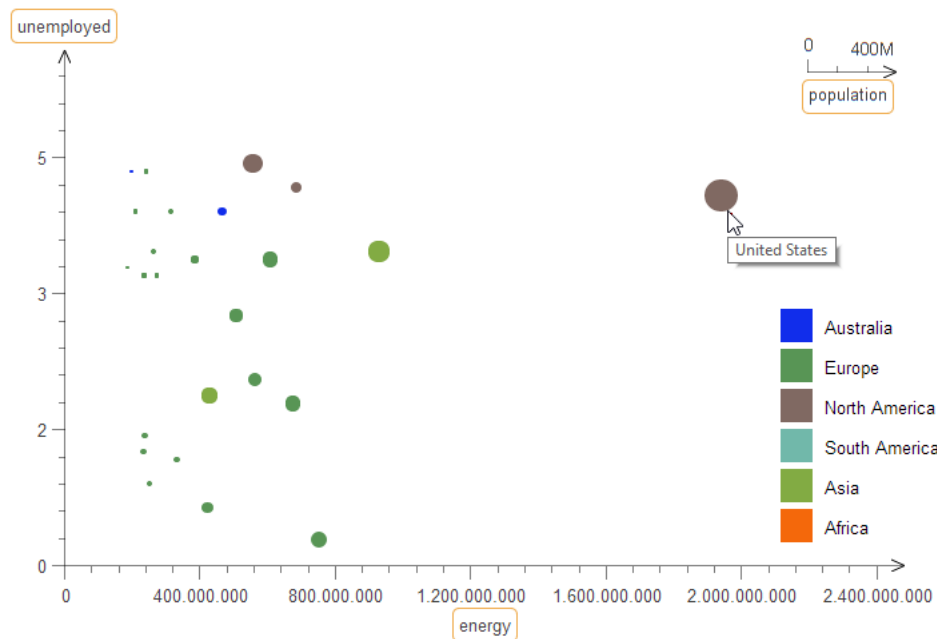


Abbildung 4.7: GW-Diagramm mit zusätzlicher Skala für die Einwohnerzahlen der Länder

werden die Farben Grün und Rot genutzt, um eine niedrige beziehungsweise hohe Arbeitslosenrate darzustellen. Für die Darstellung der Legende wird ein Farbverlauf verwendet. Eine diskrete Darstellung der einzelnen Abstufungen zwischen Grün und Rot ist potentiell auch denkbar. Allerdings ist die Darstellung als Farbverlauf wesentlich kompakter.

Weitere diskrete Legenden Für andere visuelle Variablen können stets diskrete Legenden konstruiert werden, da sie es erlauben konkrete Ausprägungen einer Variablen in Miniaturansicht darzustellen. Dies ist vor allem aus Karten bekannt, in denen verschiedene Symbole beziehungsweise Formen in Legenden nachgeschlagen werden können. Analog kann auch bei der Orientierung und Textur vorgegangen werden.

Zusammenfassung Im Rückblick auf die vorhergehenden Beispiele lässt sich erkennen, dass Skalen vor allem für Positionen und Größen geeignet sind. Alle anderen visuellen Variablen lassen sich mit diskreten Legenden beschreiben, da sie stets konkrete Ausprägungen der Variablen darstellen können. Teilweise lassen sich auch kontinuierliche Legenden verwenden, insofern kontinuierliche Ausprägungen einer visuellen Variablen genutzt wurden. Hier sei die Farbe als Beispiel genannt, für die eine kontinuierliche Legende in Form eines Farbverlaufs konstruiert werden kann.

Eine tabellarische Zusammenfassung findet sich in Tabelle 4.4.

4 Abbildung von Daten auf visuelle Elemente

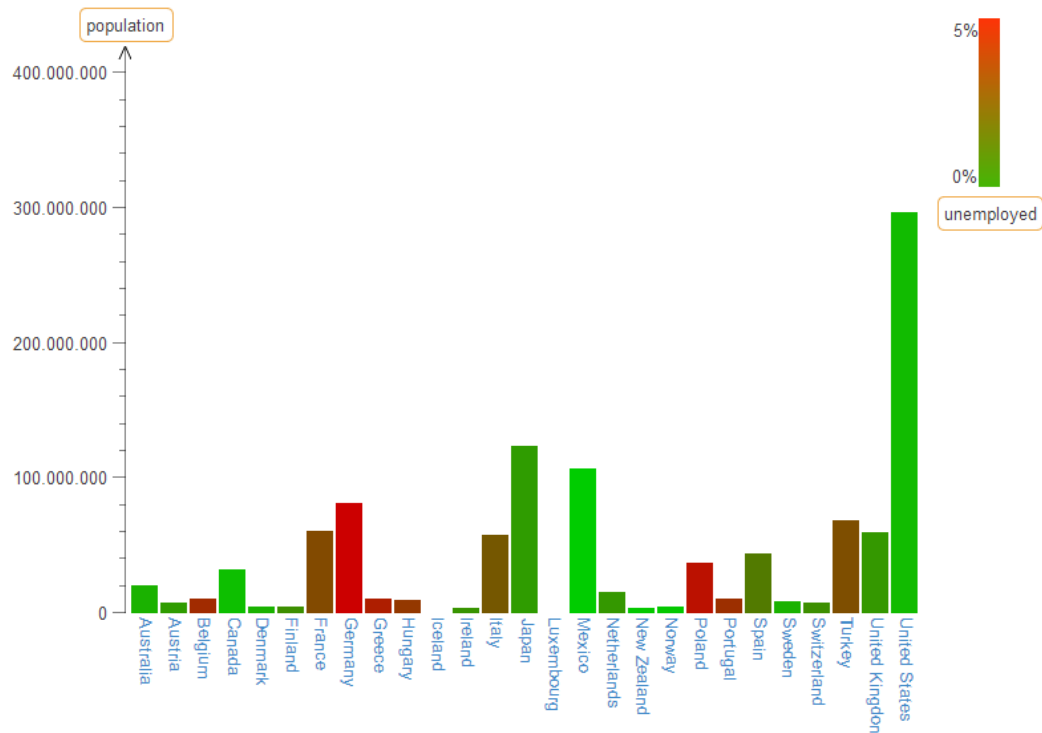


Abbildung 4.8: Farben und eine kontinuierliche Legende für die Darstellung der Arbeitslosenquoten

Tabelle 4.4: Zuordnung von visuellen Variablen zu einer geeigneten Skala beziehungsweise Legende

Visuelle Variable	Adäquate Legende/Skala
Form	diskrete Legende
Position	Skala
Größe	Skala
Helligkeit	diskrete oder kontinuierliche Legende
Farbe	diskrete oder kontinuierliche Legende
Orientierung	diskrete Legende
Textur	diskrete oder kontinuierliche Legende

4.3 Abbildung von Daten auf visuelle Elemente

Um die in Abschnitt 4.1.2 vorgestellte Visualisierung in *Flower* zu erstellen, kann ein *MorphCreator* verwendet werden. Die grundsätzliche Idee des *MorphCreators* ist es, eine Menge gleichartiger Daten auf eine Menge an gleichartigen Elementen abzubilden.

Im Kontext der Beispielvisualisierungen bedeutet dies, dass eine Menge an Ländern auf eine Menge an Rechtecken beziehungsweise Kreisen abgebildet wird. Wie der *MorphCreator* für die Visualisierungen genutzt werden kann, folgt in den nächsten Abschnitten.

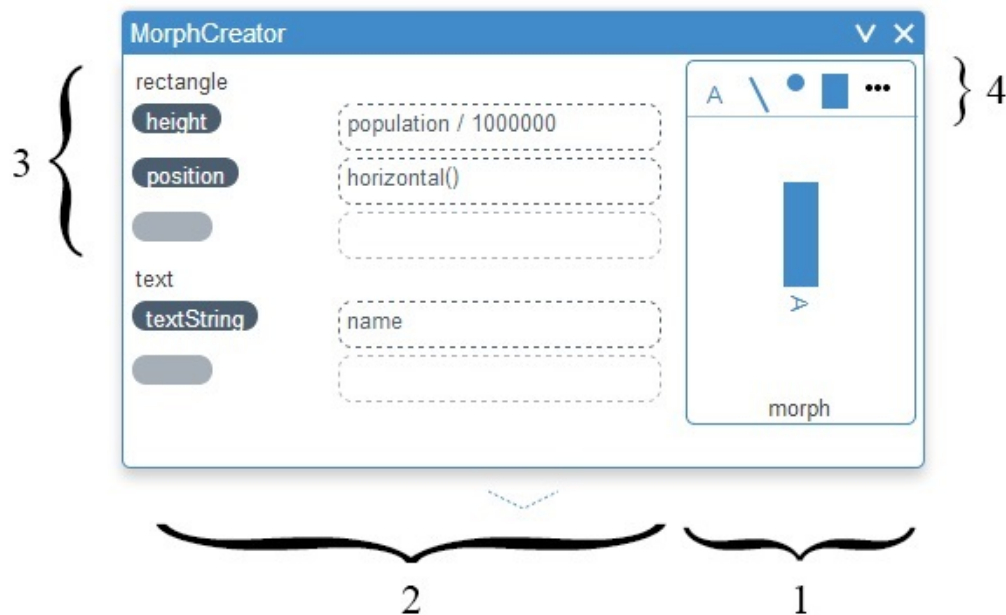


Abbildung 4.9: *MorphCreator* zur Erstellung eines Balkendiagramms

4.3.1 Beispielhafte Visualisierungserstellung

Für das Erstellen des *Balkendiagramms* kann ein *MorphCreator*, wie in Abbildung 4.9 dargestellt, verwendet werden. Als Vorlage für die einzelnen Balken wird ein Rechteck und ein Text definiert (siehe Markierung 1). Das Rechteck und der Text sind in der Umgebung des *Lively Kernels* jeweils ein *Morph* [40]. Diese können leicht händisch vom Nutzer angepasst werden. Der Text ist ein *Submorph*¹ des Rechtecks.

¹Ein *Submorph* ist ein *Morph*, der in einem anderen enthalten ist.

Für jedes Datenelement wird diese Vorlage (auch *Prototyp* genannt) kopiert. Weiterhin werden die Kopien anhand der Zuordnungen (im Folgenden auch *Mappings* genannt) angepasst (siehe Markierung 2). In dem konkreten Fall wird die Einwohnerzahl (*population*) jedes Landes auf die Höhe der einzelnen Rechtecke abgebildet. Eine Division durch 500 000 bewirkt, dass dem Radius eine dem Ausgabemedium angemessene Pixelanzahl zugewiesen wird.

Die x-Koordinate der Rechtecke wird mit der Hilfsfunktion `horizontal` spezifiziert, wodurch die Rechtecke horizontal nebeneinander angeordnet werden. Zusätzlich wird der Text unter dem Rechteck auf den Namen des jeweiligen Landes gesetzt.

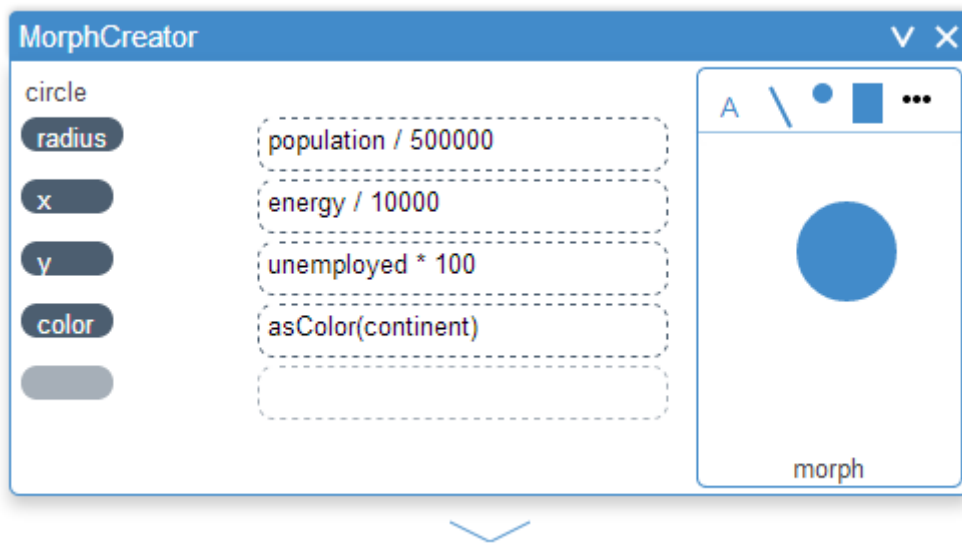


Abbildung 4.10: *MorphCreator* zur Erstellung des GW-Diagramms

Bei dem *GW-Diagramm* dient ein Kreis-Morph als Prototyp für die Repräsentationen der einzelnen Länder. Der Radius der Kreise wird anhand der Einwohnerzahl der korrespondierenden Länder gesetzt. Die x- und y-Koordinate werden abhängig von dem Energieverbrauch und der Arbeitslosenrate des Landes gesetzt. Hier wird erneut eine Skalierung der Werte anhand von Multiplikation und Division durchgeführt. Weiterhin wird aus dem Namen jedes Landes eine Farbe generiert. Dies geschieht durch Nutzung der Hilfsfunktion `asColor`, auf welche in Abschnitt 4.3.3 eingegangen wird.

4.3.2 Deklarative Abbildungen

Mithilfe des *MorphCreators* kann der Nutzer eine Überföhrungsfunktion von Datum zu Morph ausdrücken. Diese Überföhrungsfunktion besteht dabei aus:

- a) einem Prototyp-Morph, welcher das prinzipielle Aussehen der visualisierten Daten definiert (siehe Abbildung 4.9.1) – dieser spezifiziert dateninvariante Eigenschaften, welche über die *Halos* [40] vom Nutzer manipuliert werden können, neue Morphs können mithilfe der Toolbox in Markierung 4 der Abbildung erstellt werden, auf diese Art können einem Prototyp auch weitere Submorphs hinzugefügt werden;
- b) einer Menge an Deklarationen, die spezifizieren, welche visuellen Eigenschaften des Morphs nach welchen Vorschriften gesetzt werden – Jene Deklarationen sind demnach vor allem für datenvariante Eigenschaften zuständig, siehe Abbildung 4.9.2.

Verfügt der Prototyp-Morph über Submorphs, erscheinen auf der linken Seite verschiedene Kategorien. Eine ist in Abbildung 4.9 als Markierung 3 gekennzeichnet. Diese Kategorie enthält Deklarationen für das Rechteck. Darunter ist eine eigene Kategorie für den Text.

Die in b) genannten Deklarationen sind nach folgendem Schema aufgebaut:

$$\text{Morphattribut} \leftarrow \text{Ausdruck}$$

Das *Morphattribut* ist dabei eine der in Abschnitt 4.2.1 aufgelisteten visuellen Eigenschaften des Prototyp-Morphs. Eine Ausnahme ist die Form, die nicht datenabhängig bestimmt werden kann. Stattdessen wird sie fest durch die Form des Prototyp-Morphs vorgegeben. Auf diese Einschränkung wird auch in Abschnitt 4.5.2 „Zukünftige Arbeiten“ eingegangen.

Der *Ausdruck* ist ein JavaScript-Ausdrucks, welcher Zugriff auf das abzubildende Datum sowie auf all seine Eigenschaften besitzt. Diese Ausdrücke beinhalten vor allem mathematische Transformationen einzelner oder mehrerer Werte. Weiterhin können Hilfsfunktionen verwendet werden, welche im nächsten Abschnitt vorgestellt werden.

Die Transformation von Daten auf Morphs wird bei jeder Änderung des Prototyp-Morphs und der Deklarationen initiiert. Dies stellt sicher, dass der Nutzer bei Anpassungen unmittelbares Feedback erhält und so prüfen kann, ob das gewünschte Ergebnis erreicht wurde.

4.3.3 Hilfsfunktionen

Bei dem Abbilden von Daten auf visuelle Elemente sind oft ähnliche Arbeitsschritte notwendig. Dazu gehören zum Beispiel das Anordnen der Elemente oder auch das Erzeugen von Farbverläufen. Um den Nutzer von möglichst viel repetitiver Arbeit zu befreien, werden im *MorphCreator* verschiedene Hilfsfunktionen bereitgestellt. Diese werden im Folgenden vorgestellt.

Positionierung visueller Elemente

Wie viele Visualisierungen (siehe Beispielformen [21]) erkennen lassen, werden oft ähnliche Positionierungsarten für visuelle Elemente verwendet. Dazu gehören:

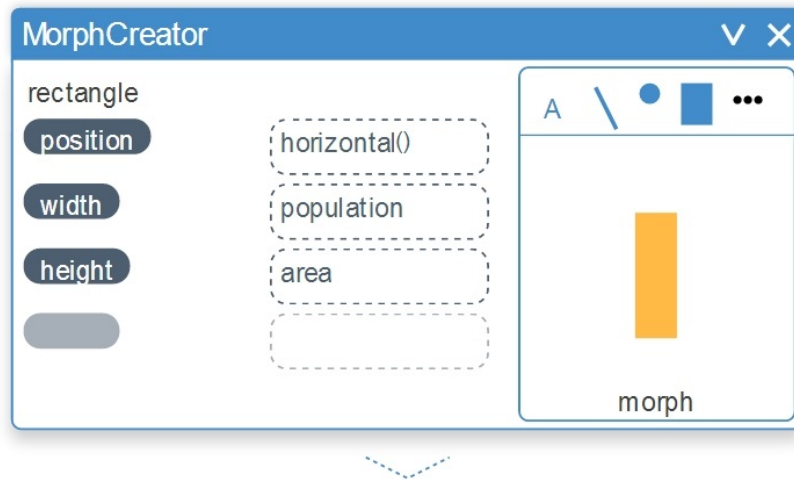


Abbildung 4.11: Positionierung im *MorphCreator*



Abbildung 4.12: Horizontale Anordnung

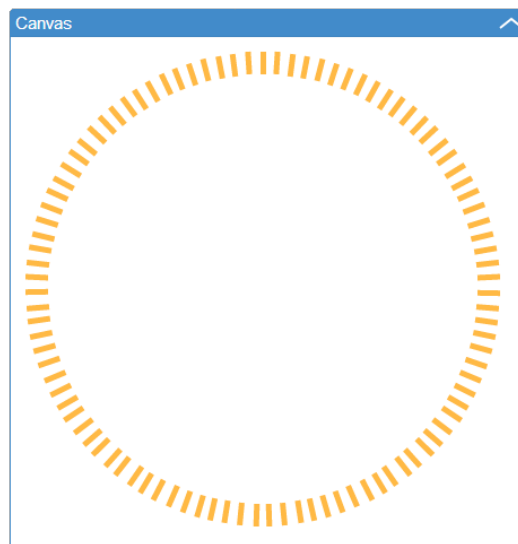


Abbildung 4.13: Anordnung im Kreis

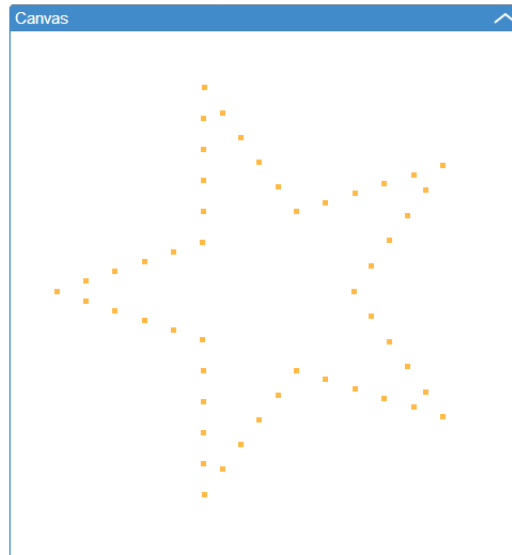


Abbildung 4.14: Anordnung mithilfe eines Pfads

- Horizontale Anordnung
- Vertikale Anordnung
- Anordnung im Kreis
- Anordnung auf einem beliebigen Pfad

Für die obigen Szenarien werden Hilfsfunktionen bereitgestellt, die im *MorphCreator* den Wert für das Positionsattribut berechnen können.

In Abbildung 4.11 ist ein *MorphCreator* zu sehen, der Länder auf gelbe Rechtecke abbildet. Die Position der Rechtecke wird dabei mit der Hilfsfunktion `horizontal` spezifiziert. Breite und Höhe sind von der Einwohnerzahl und dem Energieverbrauch des Landes abhängig.

Abbildung 4.12 zeigt die entstehenden Rechtecke, die horizontal angeordnet sind. Die unterschiedlichen Breiten der Rechtecke werden berücksichtigt, sodass die Abstände zwischen den Mittelpunkten der Rechtecke gleich groß sind. Ähnlich kann die Hilfsfunktion `circular` verwendet werden, um Elemente in einem Kreis anzuordnen und angemessen zu rotieren (siehe Abbildung 4.13).

Die Hilfsfunktion `onPath` hingegen erlaubt die Positionierung entlang eines Pfades, welcher als Parameter angegeben werden kann. In Abbildung 4.14 bildet der genutzte Pfad eine Sternform.

Interpolation innerhalb von Intervallen

Eine weitere Hilfsfunktion erlaubt die Interpolation zwischen Zahlen, Punkten oder Farben. Ein beispielhafter Anwendungsfall dafür wäre die Einfärbung von Ländern anhand der Arbeitslosenrate, die in diesem Land vorzufinden ist. Länder mit einer sehr geringen Arbeitslosenrate sollen grün und Länder mit einer relativ hohen

Arbeitslosenrate sollen rot eingefärbt werden. Unter Nutzung der Hilfsfunktion `range` kann dies wie folgt bewerkstelligt werden:

```
color ← range(Color.green, Color.red)(unemploymentrate)
```

Die Funktion `range` kann auch mehr als zwei Parameter akzeptieren. Dadurch wäre zum Beispiel ein Farbverlauf von Grün über Gelb zu Rot möglich (siehe Abbildung 4.16).



Abbildung 4.15: Verlauf von Rot nach Grün



Abbildung 4.16: Verlauf von Rot über Gelb nach Grün

Abbilden auf Farben

Oft werden Farben genutzt, um die Zugehörigkeit zu einer bestimmten Kategorie darzustellen. So können Länder beispielsweise anhand ihres Kontinents eingefärbt werden. Länder aus Europa könnten zum Beispiel blau und Länder aus Asien grün dargestellt werden. Welche Farben den einzelnen Kontinenten dabei zugeordnet werden, ist oft nicht relevant. Für diesen Anwendungsfall wird die Hilfsfunktion `asColor` bereitgestellt. Diese erhält eine beliebige Zeichenkette als Parameter und bildet sie auf eine Farbe ab. Gleiche Zeichenketten werden dabei auf gleiche Farben abgebildet. Das beschriebene Beispiel findet sich in Abbildung 4.2 auf Seite 59 wieder.

4.3.4 Erstellen von Skalen und Legenden

Nach dem Abbilden der Daten auf visuelle Elemente mittels des *MorphCreators*, ist ein häufiger Anwendungsfall das Hinzufügen von Skalen und Legenden (siehe Abschnitt 4.2.3). Um dem Nutzer möglichst viel Arbeit abzunehmen, soll dieser lediglich spezifizieren, für welche Dimension eine Skala oder Legende erstellt werden soll. Im Beispiel aus Abbildung 4.1 (Seite 59) wurde die Einwohnerzahl der Länder auf die Höhe der Rechtecke abgebildet. Hier müsste der Nutzer nur angeben, dass er für die Einheit *Einwohnerzahl* eine Skala oder Legende benötigt. Dies bewerkstelligt der folgende Code:

```
var canvas = new Canvas(data);  
canvas.addLegendFor("population");
```

Hier wird die Klasse *Canvas* genutzt, welche die visuellen Elemente kapselt und es zusätzlich ermöglicht, die Darstellung geeignet zu parametrisieren. Da bereits im *MorphCreator* angegeben wurde, welche Attribute auf die Einwohnerzahl abgebildet wurden, muss der Nutzer nicht mehr entscheiden, welche Skala oder Legende am besten geeignet ist. Stattdessen wird diese Entscheidung auf Basis der in Abschnitt 4.2.3 vorgestellten Fälle von *Flower* getroffen.

Die erstellten Skalen und Legenden werden automatisch beschriftet und mit den geeigneten Wertebereichen annotiert. Im konkreten Beispiel kann deshalb eine Y-Achse konstruiert werden, die mit Werten von 0 bis 4 000 000 beschriftet ist. Weiterhin ist auch die abgebildete Dimension *population* oberhalb der Achse vermerkt.

4.4 Implementierung

Die Arbeitsweise des *MorphCreators* lässt sich in drei Schritte unterteilen:

1. Für jedes Datenelement wird eine Kopie des Prototyp-Morphs erstellt.
2. Jede Kopie wird anhand der deklarativen Mappings transformiert.
3. Aus den deklarativen Mappings werden Meta-Daten extrahiert, die in den Morphs abgelegt werden. Dies ist für die Erstellung von Skalen und Legenden notwendig (siehe Abschnitt 4.3.4).

Wie diese drei Aufgaben implementiert wurden, wird in den folgenden Bereichen erläutert.

4.4.1 Beschleunigter Kopieralgorithmus

Um dem Nutzer eine schnell reagierende Benutzeroberfläche zu bieten, wurde versucht, Performance-Engpässe zu vermeiden. Ein großes Optimierungspotential stellte dabei das Kopieren von Morphs dar. Wird eine Menge von 1000 Datenelementen auf eine Menge von 1000 Morphs abgebildet, muss der *MorphCreator* 1000 Kopien des Prototyp-Morphs anlegen. Dies kann je nach Komplexität des Morphs und genutztem System bis zu 50 Sekunden dauern. Nach Nielsen [34] führen Aktionen, die länger als 10 Sekunden dauern, dazu, dass der Nutzer die Aufmerksamkeit verliert. Deshalb wurde ein verbesserter Kopieralgorithmus entworfen, der eine deutlich geringe Ausführungszeit hat und selbst bei großen Datenmengen Niensens 10-Sekunden-Grenze oft unterschreitet.

Um den bisherigen Kopieralgorithmus zu verbessern, wird dieser vorgestellt und auf seine Schwachstelle eingegangen. Zunächst wird die Vorgehensweise des Standard-Kopieralgorithmus untersucht. Jene besteht aus den folgenden drei Schritten:

4 Abbildung von Daten auf visuelle Elemente

- 1) *Serialisierung* Der zu kopierende Morph (samt Submorphs) wird zu einer Zeichenkette serialisiert. Dabei werden alle Attribute des Morphs traversiert und die Schlüssel-Wert-Paare als JSON [9] abgespeichert.
- 2) *Deserialisierung* Die in 1) entstandene Zeichenkette wird anschließend genutzt, um aus den enthaltenen Informationen einen neuen Morph zu erstellen. Dies ist die gleiche Vorgehensweise nach der auch beim Laden einer Lively-Welt verfahren wird.
- 3) *Einhalten bestimmter Rahmenbedingungen* Anschließend muss dafür gesorgt werden, dass bestimmte Rahmenbedingungen eingehalten werden. So muss zum Beispiel jeder Morph eine eindeutige ID und einen eigenen Renderkontext besitzen. Hierfür werden Lively-interne Methoden wie *prepareForNewRenderContext* und *findAndSetUniqueName* aufgerufen.

Da ein Großteil der Zeit bei dem Kopieren für die Serialisierung und Deserialisierung verwendet wird, wird versucht auf jene zu verzichten. Stattdessen wird das nativ in JavaScript vorhandene Konzept der *Prototype-Delegation* [41, Kapitel 4.2.1] verwendet. Der neue Kopieralgorithmus teilt sich in die vier folgenden Abschnitte.

- 1) *Prototyp-Referenzierung* Zuerst wird ein neues Objekt k erstellt, dessen Prototyp-Referenz auf den Originalmorph o gesetzt wird. Dadurch wird der lesende Zugriff auf Attribute, die nicht in k vorhanden sind, an o weitergeleitet. Soll hingegen ein Wert *gesetzt* werden, wird dies nicht an den Prototypen weitergeleitet, sodass dieser unverändert bleibt.
- 2) *Kopie der Form* Wenn die Form (shape) nicht separat kopiert werden würde, würden sich alle Kopien das gleiche Form-Objekt teilen. Jede Attributänderung der Form würde auf der Ursprungsform ausgeführt werden. Dies bedeutet, dass alle Kopien stets dieselbe Position, Farbe und alle anderen Attributwerte hätten, die in der Form vorgehalten werden. Um dies zu verhindern, wird von der Form des Morphs eine Kopie angelegt.
- 3) *Einhalten bestimmter Rahmenbedingungen* Hier wird ähnlich wie bei dem Standard-Kopieralgorithmus dafür gesorgt, dass bestimmte Rahmenbedingungen eingehalten werden.
- 4) *Kopie aller Submorphs* Dieser Kopierprozess wird nun rekursiv für alle Submorphs wiederholt.

Der Vorteil dieser Methode ist vor allem, dass auf die zeitaufwendige Serialisierung und Deserialisierung verzichtet wird. Weiterhin kann eine sehr große Anzahl an Kopien erstellt werden. Sollen n Kopien erstellt werden, muss die Kopierfunktion also nicht n -mal aufgerufen werden, sondern nur *einmal*. In dem Aufruf der Kopierfunktion werden in n Schleifendurchläufen n Kopien angelegt. Dadurch werden auch unnötige Performance-Kosten für das Aufrufen von Funktionen gespart.

Evaluierung des Performance-Gewinns

Um den Performance-Gewinn des neuen Kopieralgorithmus zu evaluieren, wurden zwei Test-Szenarien angefertigt. Im ersten Szenario wird ein *simpler* Morph n -mal kopiert und dabei die Zeit gemessen, wobei $n \in \{10, 100, 1000\}$ gilt. Der *simple* Morph ist dabei ein blaues Rechteck. Im zweiten Szenario wird statt des *simple* Morphs ein *komplexer* Morph kopiert. Der *komplexe* Morph ist dabei ein blauer Kreis, der vier Submorphs enthält (drei Rechtecke und eine Linie).

Die Zeitmessung wird mithilfe der `console.time` und `console.timeEnd` Funktionen² durchgeführt, welche von den gängigen, modernen Browsern³ innerhalb des JavaScript-Kontextes bereitgestellt werden.

Die beiden Test-Szenarien wurden auf einem Test-System⁴ jeweils 100-mal durchgeführt und die Ergebnisse gemittelt. Dadurch wurden etwaige Performance-Schwankungen durch externe Einflüsse gemindert.

Die gemessenen Werte finden sich in Tabelle 4.5. Die Zeiten für die Kopieralgorithmen sind pro Morph und in Millisekunden angegeben. Die Leistungssteigerung ist als Faktor angegeben. Erkennbar ist, dass der neue Kopieralgorithmus besonders bei einer hohen Anzahl an zu kopierenden Morphs bis zu einem Faktor von 5 schneller ist.

Tabelle 4.5: Vergleich der verschiedenen Kopieralgorithmen

Typ / Anzahl	Vorher	Nachher	Leistungssteigerung
Simple Morphs			
10	3,40	1,12	3,03
100	3,42	0,62	5,50
1000	3,35	0,58	5,75
Komplexe Morphs			
10	53,78	15,53	3,46
100	51,18	10,51	4,87
1000	51,73	9,55	5,42

² <https://developer.mozilla.org/en-US/docs/Web/API/console.time>; besucht am 01. Oktober 2014.

³ Chrome, Firefox, Opera, Safari und Internet Explorer in den aktuellen Versionen, Stand 27.06.2014.

⁴ Die Tests wurden in Chrome 35 unter Windows 8.1 mit einem Intel Core i5-2540M (2.60 GHz) Prozessor durchgeführt.

4.4.2 Anwendung der deklarativen Zuordnungen

Nach dem Kopieren des Prototyp-Morphs werden die vom Nutzer spezifizierten Zuordnungen (*Mappings*) datenabhängig evaluiert und die entsprechenden Attribute im Morph angepasst. Ein Mapping ist dabei nach folgendem Schema aufgebaut:

$$\text{Morphattribut} \leftarrow \text{Ausdruck}$$

Aus Programmierersicht liegen Morphattribut und Ausdruck jeweils als String vor (siehe Textfelder in Abbildung 4.9.3, Seite 67). Anhand des Ausdrucks muss der Wert berechnet werden, der für das Morphattribut gesetzt werden soll. Der Prozess teilt sich demnach in *Evaluierung des Ausdrucks* und in *Setzen des dazugehörigen Attributs* auf.

Evaluierung der Ausdrucks

Zunächst muss der Ausdruck evaluiert werden. Dafür wird die JavaScript-Funktion `eval` genutzt, welche einen beliebigen Code-String als Parameter erhält und diesen ausführen kann. Bei der Evaluierung des Ausdrucks sollte er Zugriff auf folgende Variablen haben:

- auf das Datum, welches auf den Morph abgebildet wird (inklusive all seiner Attribute),
- auf den Morph, auf den das aktuelle Datum abgebildet wird,
- auf den Index des aktuellen Datums,
- auf die Interaktionsvariablen (siehe Abschnitt 5.2.1), und
- auf die Hilfsfunktionen (siehe Abschnitt 4.3.3).

Damit der Zugriff auf diese Variablen möglich ist, wird der aktuelle Sichtbarkeitsbereich um jene Variablen erweitert. Dies geschieht mithilfe des `with`-Statements [41, Kapitel 12.10], welches ein Objekt zur lexikalischen Umgebung des aktuellen Ausführungskontexts hinzufügt. Die Implementierung kann in Quelltext 4.2 nachvollzogen werden. Die dabei konstruierte `valueFn`-Funktion benötigt lediglich ein Datum als Parameter und liefert das Resultat des ursprünglich vorhandenen JavaScript-Ausdrucks.

Während die Interaktionsvariablen über `env.interaction` und die Hilfsfunktionen über `lively.morphic.Charts.Utils` bereitgestellt werden, ist das aktuelle Tupel aus Datum und Morph sowie der dazugehörige Index in `currentMappingContext` zu finden. Die einzelnen Attribute des Datums werden über `datum` bereitgestellt.

Setzen des Morphattributs

Nachdem der gewünschte Wert für das spezifizierte Morphattribut berechnet werden kann, muss anschließend jenes Attribut auf diesen Wert gesetzt werden.

Quelltext 4.2: Erstellung der valueFn-Funktion

```

1 var valueFn = function(datum) {
2     // ...
3     with (env.interaction)
4     with (lively.morphic.Charts.Utils)
5     with (datum)
6     with (currentMappingContext) {
7         return eval("return " + eachMapping.value + "");
8     }
9 };

```

Dafür wird ein JavaScript-Objekt erstellt, welches dazu dient, die geeigneten Setter-Funktionen nachzuschlagen. Ein JavaScript-Objekt besteht aus einer Menge an Schlüssel-Wert-Paaren [41, Kapitel 4.2.1]. Die *Schlüssel* des Objekts repräsentieren manipulierbare Morph-Attribute. Die dazugehörigen *Werte* sind Funktionen, welche ein Attribut eines gegebenen Morphs auf einen gewünschten Wert setzen. In Quelltext 4.3 findet sich ein Auszug des dazugehörigen Codes. Die nachgeschlagene Funktion kann mithilfe des zu setzenden Wertes `value` und des aktuellen Morphs als Parameter evaluiert werden.

Quelltext 4.3: Zuordnung von Attribut-Zeichenkette zur korrespondierenden Setter-Funktion

```

1 getAttributeMap: function() {
2     return {
3         // ...
4         // further attributes
5         // ...
6         x: function(morph, value) {
7             morph.setPosition(lively.pt(value, morph.getPosition().y));
8         },
9         // ...
10    }
11 }

```

Kombination von Evaluierung und Setzen

Anschließend kann für alle Mappings folgender Code ausgeführt werden:

```

var setterFn = getAttributeMap()[eachMapping.attribute];
setterFn(morph, valueFn(datum));

```

Dies wird in der `update`-Methode des *MorphCreators* durchgeführt. Die Ausgabe des *MorphCreators* umfasst dadurch Morphs, in denen die jeweiligen Attribute mit den berechneten Werten versehen sind.

4.4.3 Extraktion von genutzten Attributen

Die Wahl geeigneter Skalen beziehungsweise Legenden wird dem Nutzer, wie in Abschnitt 4.3.4 vorgestellt, weitestgehend abgenommen. Die Konstruktion muss in Abhängigkeit von den gewählten Mappings vollzogen werden. Dafür ist es notwendig zu wissen, welche Attribute von welchen Dateneigenschaften abhängig sind. Um zu erfahren, welche Eigenschaften verwendet wurden, können verschiedene Ansätze verfolgt werden. Jene werden im Folgenden vorgestellt.

Realisierung durch Anlegen von Zugriffsfunktionen

Eine denkbare Herangehensweise stellt das klassische Observer-Pattern [42] dar. Dabei werden Getter-Funktionen [36] für alle Attribute der Datenelemente definiert. Dadurch kann bei jedem Zugriff auf die Datenattribute eine Getter-Funktion aufgerufen werden, welche die genutzten Variablen protokolliert. Allerdings müsste für *jedes* Attribut *aller* Eingangsdaten eine Getter-Funktion angelegt werden. Dies ist nicht nur vom Implementierungsaufwand, sondern auch vom Rechenaufwand nicht sinnvoll. Ähnlich wurde auch bei der Implementierung des *Interactionpanels* verfahren (siehe Abschnitt 5.2.1). Dort stellt die Performance jedoch kein Problem dar, weil die Komplexität linear zu der Anzahl der Interaktionsvariablen und nicht zur Anzahl der Daten ist.

Realisierung durch Parsen

Statt dem Anlegen von Getter-Funktionen, wurde ein anderer Ansatz verfolgt.

Quelltext 4.4: Beispielzuordnung

```
height ← Math.sqrt(population)
```

1. Die Ausdrücke werden mittels des *acorn*-Parser [37] geparkt und anschließend wird der AST (*abstract syntax tree*) traversiert und die genutzten Identifier extrahiert.

Als Beispiel wird der Ausdruck aus Quelltext 4.4 verwendet, welche als String vorliegt. Der dazugehörige Datensatz findet sich in Quelltext 4.1. Die im geparkten Code gefundenen Identifier lauten hier: `Math`, `sqrt` und `population`.

2. Die Identifier, welche eine Eigenschaft (*Property* [41, Kapitel 4.3.26]) eines Objekts sind, werden verworfen.

Im konkreten Beispiel wird `sqrt` verworfen, da es ein Attribut von `Math` ist. `Math` und `population` verbleiben.

3. Anschließend wird ein Probedatum ausgewählt, um zu prüfen, ob die Identifier Attribute des Datums sind.

Das Probedatum aus Quelltext 4.4 enthält nur das Attribut `population`. `Math` hingegen ist kein Attribut des Probedatums.

4. Ist der Test erfolgreich, so wird davon ausgegangen, dass das aktuelle Attribut von dem Identifier abhängig ist.

Die Höhe (*height*) der visuellen Elemente ist demnach abhängig von der Einwohnerzahl (*population*) der Länder.

Diesem Vorgehen liegt die Annahme zugrunde, dass die vorliegenden Daten homogen sind und die Existenz eines Attributs in einem Datum repräsentativ dafür ist, dass das Attribut in allen Daten existiert.

4.5 Zusammenfassung und Ausblick

Für die Abbildung von Daten auf visuelle Elemente sind vor allem visuelle Variablen sowie Skalen und Legenden von zentraler Bedeutung. Im Rahmen von *Flower* erlaubt es der *MorphCreator* komplexe Abbildungen einfach und präzise auszudrücken. Dadurch können Visualisierungen schnell konstruiert werden.

Ermöglicht wird dies zum einen durch die Nutzung von Prototypen, die das grundlegende Aussehen der visuellen Elemente bestimmen. Dadurch wird der Nutzer von der Pflicht entbunden, sinnvolle Standards, beispielsweise für Farbe, Position, Größe, programmatisch zu setzen. Über die *Halos* können Änderungen vorgenommen werden, ohne korrespondierenden Code aufsuchen und verändern zu müssen.

Zum anderen liefert die deklarative Beschreibung datenvarianter Eigenschaften eine prägnante Möglichkeit, Abhängigkeiten zwischen Daten und visuellen Variablen darzustellen. Da der *MorphCreator* auf Änderungen unmittelbar mit den angepassten Morphs reagiert, können potentiell geeignete visuelle Variablen erprobt und hinsichtlich ihrer Adäquatheit evaluiert werden. Zusätzlich werden wiederkehrende Aufgaben durch das Anbieten von Hilfsfunktionen abgenommen beziehungsweise stark erleichtert.

4.5.1 Verwandte Arbeiten

Lyra Die Software *Lyra* [43] ermöglicht das Entwerfen von Visualisierungen in einer interaktiven Umgebung ohne jeglichen Quelltext. Neben der Tatsache, dass kein Quelltext verwendet wird, ist der größte Unterschied zu *Flower*, dass kein Datenflusskonzept verwendet wird. Stattdessen gibt es ein *Canvas*, auf dem stets die Visualisierung zu sehen ist (siehe Abbildung 4.17). Modifikationen können über ein kontextsensitives *Side Panel* durchgeführt werden (siehe rechten Bereich von Abbildung 4.17).

Für die Abbildung von Daten auf visuelle Elemente können – ähnlich wie im *MorphCreator* – Basisformen mithilfe einer Toolbox per Drag-and-Drop erstellt werden. Diese können anschließend direkt auf das *Canvas* gezogen werden. Die Abhängigkeiten von den Daten können ebenfalls per Drag-and-Drop erzeugt werden. Dabei werden Attribute aus einer Datentabelle (linkes *Side Panel*) auf die Eigenschaften im rechten *Side Panel* gezogen.

4 Abbildung von Daten auf visuelle Elemente

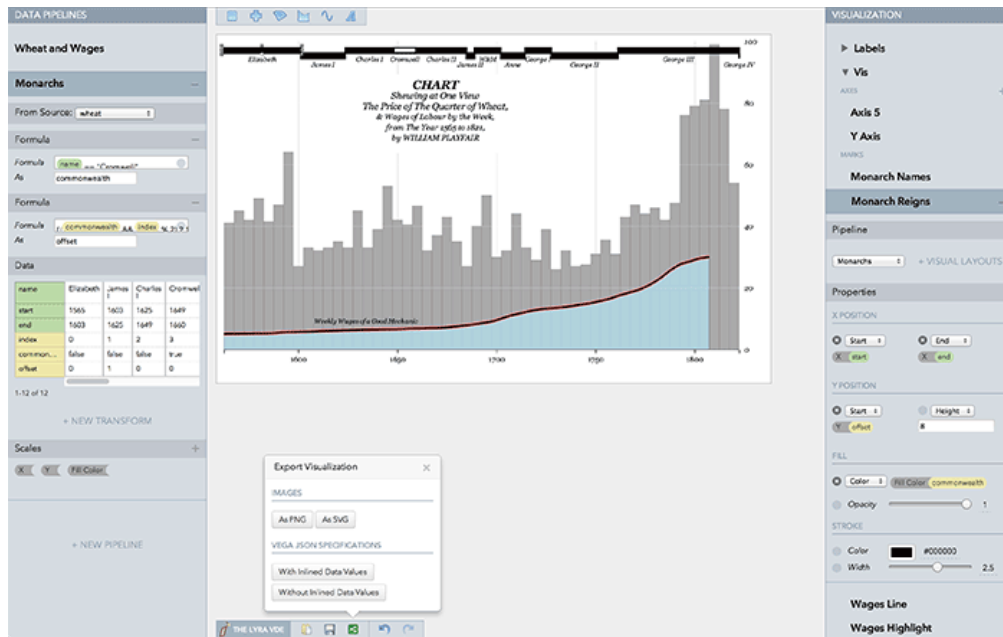


Abbildung 4.17: Lyra: An Interactive Visualization Design Environment.
Quelle: Arvind Satyanarayan und Jeffrey Heer [43]

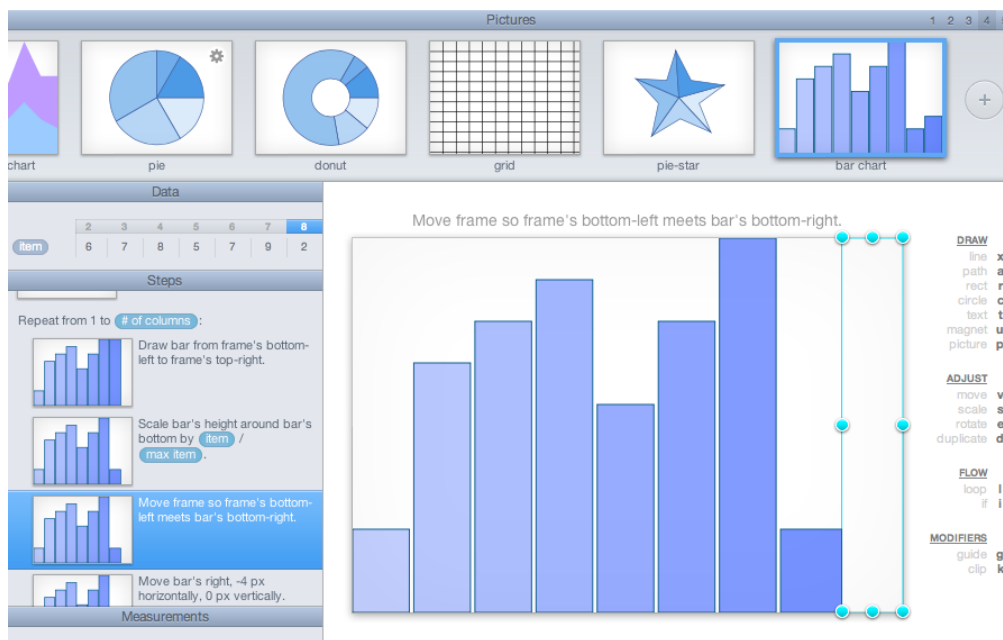


Abbildung 4.18: Drawing Dynamic Visualizations von Bret Victor.
Quelle: Bret Victor [44]

Drawing Dynamic Visualizations Ein Ansatz, der ebenfalls ohne Quelltext auskommt, wurde von Bret Victor in seinem Vortrag *Drawing Dynamic Visualizations* [44] präsentiert. Die Benutzeroberfläche kann in Abbildung 4.18 gefunden werden. Formen werden ähnlich wie in einem Zeichenprogramm erstellt. Es gibt verschiedene Zeichenwerkzeuge zur Auswahl, welche zum Beispiel Linien, Pfade, Rechtecke und Kreise erzeugen können. Jede Aktion des Nutzers wird dabei in einem Protokoll aufgelistet. Die einzelnen Aktionen des Protokolls können im Nachhinein anhand der Daten parametrisiert werden. Ermöglicht wird dies durch Drag-and-Drop von den Datenattributen auf die jeweiligen Parameter im Protokoll. Durch das Einfügen von Schleifen können die manuell durchgeführten Schritte für alle Datenelemente wiederholt werden, womit das Erstellen kompletter Diagramme möglich ist.

4.5.2 Zukünftige Arbeiten

Für die zukünftige Arbeit sind verschiedene Ideen denkbar, die sich vor allem auf die Vereinfachung bestimmter Arbeitsabläufe konzentrieren.

Automatische Skalierung Oft müssen konkrete Werte skaliert werden, sodass auf dem Ausgabemedium eine angemessene Größe erreicht wird. So war beispielsweise in Abbildung 4.9 eine Skalierung anhand des Faktors 500 000 zu sehen. Denkbar wäre, dass die Position und/oder die Größe gewünschter Elemente automatisch skaliert werden. So könnten alle Elemente auf dem *Canvas* Platz finden, ohne dass der Nutzer genaue Skalierungsfaktoren manuell bestimmen muss. Zusätzlich könnte man hier nicht nur lineare, sondern auch exponentielle oder andere Skalierungsarten anbieten.

Linien und Flächen Eine weitere Idee ist das einfache Erstellen von Linien oder Flächen. Bisher kann mithilfe des *MorphCreators* eine Menge an Daten auf eine Menge an Linien oder Flächen abgebildet werden. Allerdings wäre es auch denkbar, eine Menge an Daten auf eine Menge an Punkten abzubilden, die zu einer Linie oder eine Fläche verbunden werden. Dies könnte sich besonders bei Linien- und Flächendiagrammen als hilfreich erweisen.

Form als visuelle Variable Bisher wird die Form der visuellen Elemente durch den Prototyp-Morph bestimmt. Eine datenabhängige Wahl der Form ist nicht möglich. Denkbar wäre, dass man mehrere Prototyp-Morphs im *MorphCreator* erzeugt und jene mit Namen versieht. Anschließend könnte man über ein Attribut *shape* eine Expression anlegen, die in Abhängigkeit der Daten entscheidet, welcher Prototyp gewählt wird. Ein Beispiel dafür kann in Abbildung 4.19 betrachtet werden. Dort werden Datenelemente mit einem durch zwei teilbaren Index auf Kreise abgebildet. Die restlichen Datenelemente werden auf Rechtecke abgebildet. Es handelt sich hierbei allerdings noch um ein Mock-up.

4 Abbildung von Daten auf visuelle Elemente

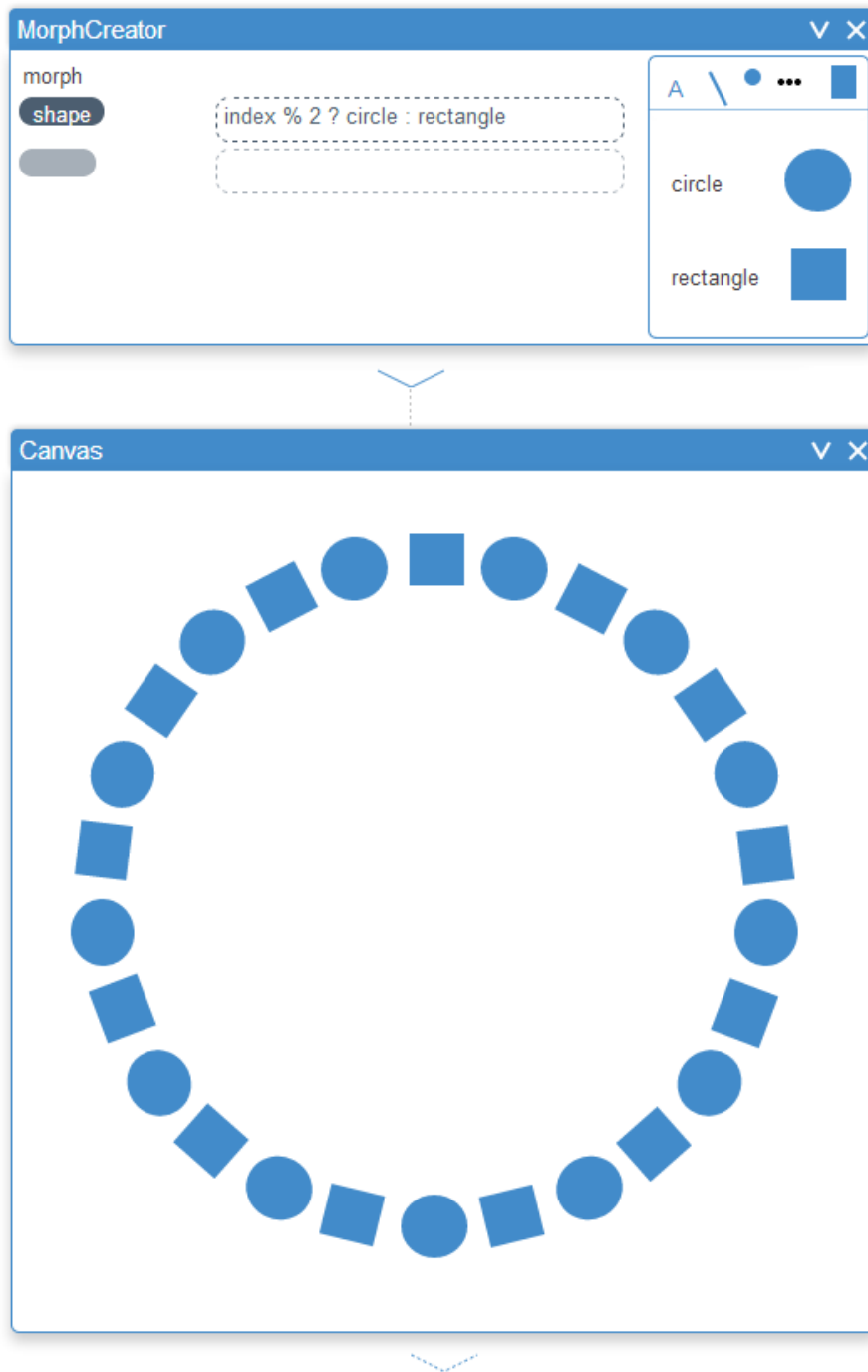


Abbildung 4.19: Datenabhängige Wahl der Form als visuelle Variable

5 Interaktionskonzepte

5.1 Einleitung und Motivation

Bei einer komplexen Visualisierung zählt Interaktion in vielen Fällen zu einem wesentlichen Bestandteil. Durch das eigenständige Interagieren mit der Visualisierung kann der Nutzer die Daten selbst entdecken. Interaktion erleichtert zudem das Verstehen von komplexen Zusammenhängen.

Nachdem die Daten aufbereitet wurden und auf visuelle Elemente abgebildet wurden, wird im letzten Schritt der Visualisierungserstellung Interaktion hinzugefügt. Dieses Kapitel beschreibt, wie der Nutzer bei diesem Schritt unterstützt wird und wie sich Interaktionskonzepte in das Projekt *Flower* integrieren lassen.

Im ersten Abschnitt wird die Interaktivität bei Diagrammen definiert sowie auf die Unterschiede zwischen statischen und dynamischen Visualisierungen eingegangen. Anschließend werden einige Interaktionsmöglichkeiten an drei interaktiven Beispielgrafiken vorgestellt. Im letzten Unterabschnitt wird eine Unterteilung der Interaktionskonzepte eingeführt, die im Rahmen dieses Projektes entstand.

5.1.1 Interaktivität bei Visualisierungen

Unter Interaktion versteht man im Allgemeinen die Wechselwirkungen zwischen mindestens zwei verschiedenen Individuen oder Akteuren [31]. Im Kontext von Visualisierungen erlauben interaktive Diagramme dem Nutzer die Veränderung dieser Diagramme durch seine Eingaben. Generell wird zwischen *statischen* und *interaktiven* Visualisierungen unterschieden [32].

- *Statische* Visualisierungen eignen sich vor allem im Druckbereich und um Rohdaten oder das fertige Ergebnis darzustellen. Bei *statischen* Visualisierungen ist jedoch keine Veränderung des Diagramms möglich.
- Wenn die Interessen der Nutzergruppe unklar sind, so sind *interaktive* Diagramme zu bevorzugen [33]. Dies begründet sich darin, dass der Nutzer die Möglichkeit hat, die Visualisierung zu untersuchen und seine eigenen Schlüsse zu ziehen. Weiterhin kann der Anwender die Daten selbst erforschen und für ihn relevante Informationen genauer analysieren.

5.1.2 Untersuchung beispielhafter Visualisierungen

Der Twitter-Monitor zur Bundestagswahl

Eine Beispielformalisierung ist der *Twitter-Monitor* [22] für die Bundestagswahl 2013. Diese Visualisierung, welche in Abbildung 5.1 zu sehen ist, zeigt, wie viele Tweets

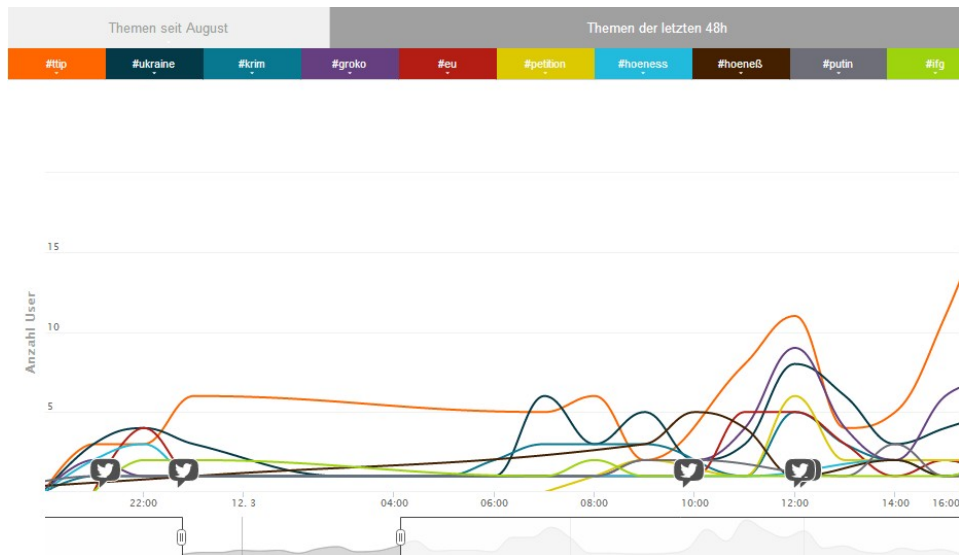


Abbildung 5.2: Nach Verschieben der Zeitspanne nach links

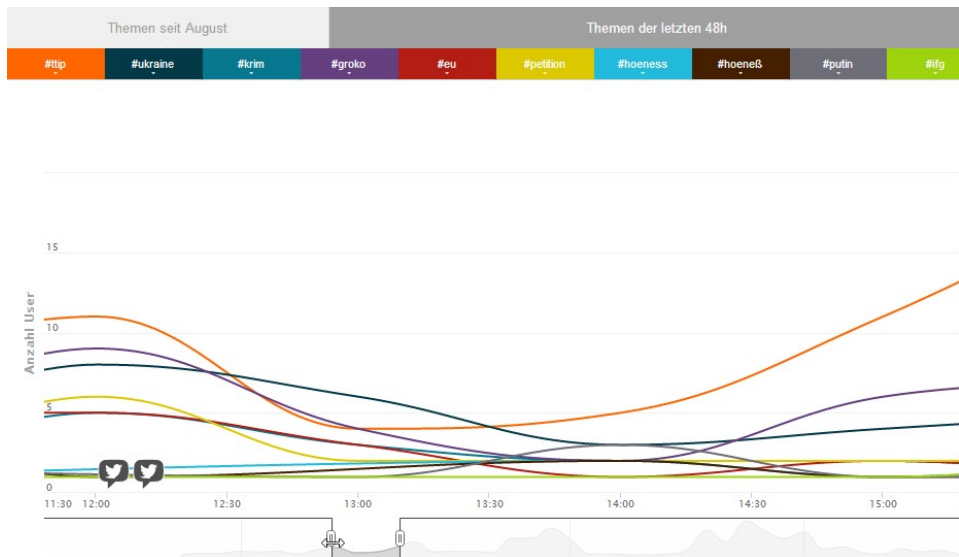


Abbildung 5.3: Nach Verkleinerung der Zeitspanne



Abbildung 5.4: Nach Klick auf das Hashtag Ukraine

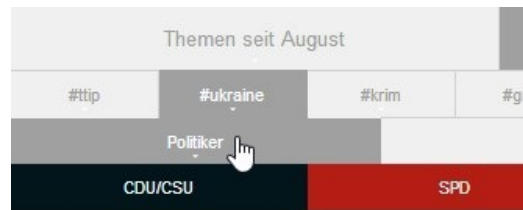


Abbildung 5.5: Nach Klick auf das Hashtag Politiker

2. Der *Twitter-Monitor* verwendet neben dem Zeitfenster eine weitere Interaktionsmöglichkeit. Durch einen Klick auf einen der oberen Hashtags ist es möglich, dass die Daten auf Tweets mit einem bestimmten Hashtag eingeschränkt werden. Wie die Visualisierung nach dem Klick auf Ukraine aussieht, zeigt Abbildung 5.4. Es werden wiederum mehrere Graphen angezeigt, jedoch steht diesmal jeder Graph für Tweets von einer bestimmten Personengruppe (zum Beispiel Politiker, Medien, Verbände). Diese Interaktion kann man nun wiederholen und auf die Gruppe Politiker klicken, dadurch entsteht Abbildung 5.5. Durch das Klicken auf die einzelnen Hashtags werden die angezeigten Daten gefiltert und der Nutzer kann sie untersuchen.

Gapminder World

Als zweites Beispiel dient ein Diagramm von Hans Rosling namens *Gapminder World* [20] (siehe Abbildung 5.6). In dieser Visualisierung steht jeder Kreis für ein Land beziehungsweise einen Kontinent. Auf der x-Achse wird das Einkommen pro Person abgebildet und auf der y-Achse die Lebenserwartung. Die Größe der Kreise stellt die Einwohnerzahl dar. Jede Farbe der abgebildeten Objekte repräsentiert eine geografische Region, wie sie rechts auf der Weltkarte dargestellt ist. Die *Gapminder World*-Visualisierung ermöglicht folgende *Interaktionsarten*:

1. Es kann über einen der Kreise gehovert werden. Wenn über Deutschland gehovert wird (wie in Abbildung 5.6 dargestellt), wird der Kreis hervorgehoben und es erscheint der Ländername („Germany“). Zusätzlich werden an den Achsen die exakten Werte zu Deutschland angegeben. Auch in den anderen Legenden zum Beispiel bei der geografischen Region und der Größenabbildung werden die zu dem Datenelement passenden Werte hervorgehoben.
2. Im unteren Teil des Diagramms sind eine Play-Schaltfläche und ein Zeitschieberegler abgebildet. Diese beiden Eingabelemente befähigen den Nutzer durch die Daten zu navigieren oder die zeitliche Entwicklung automatisch ablaufen zu lassen.
3. Auch das Austauschen der visualisierten Daten ist durchführbar. Dazu benutzt man die Dropdown-Menüs, welche sich neben den Skalen befinden. Beispielsweise kann auf die x-Achse statt des Einkommens auch die Anzahl der Kinder pro Frau abgebildet werden.

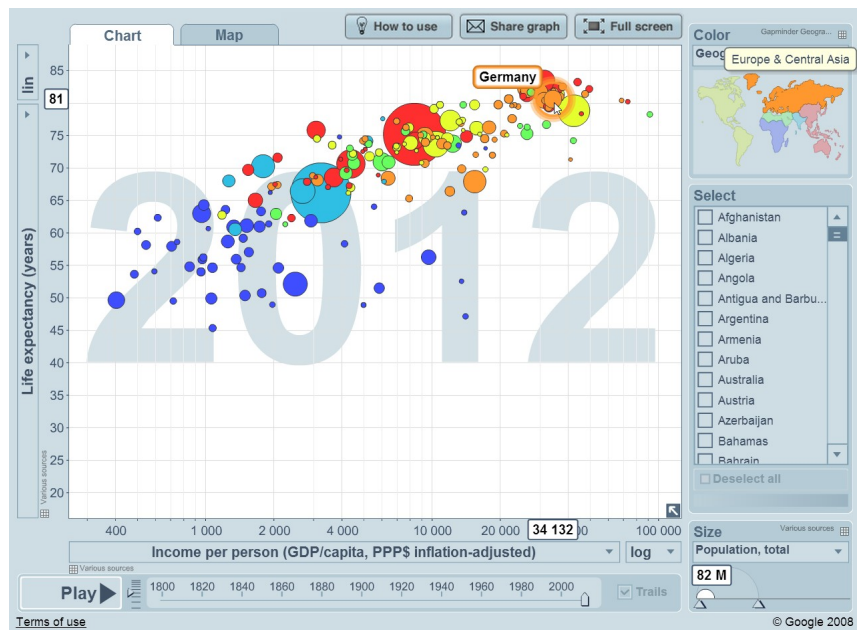


Abbildung 5.6: *Gapminder World* beim Hovern über Deutschland
 Quelle: Hans Rosling [20]

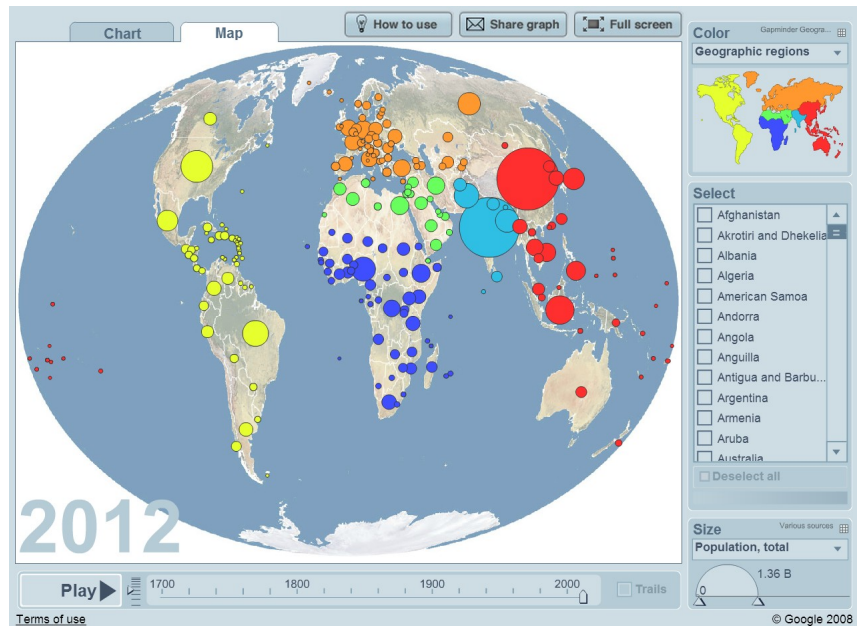


Abbildung 5.7: Kartenansicht bei der *Gapminder World*; Quelle: Hans Rosling [20]

4. Weiterhin lässt sich die Darstellungsmöglichkeit der Skalen mit Hilfe eines Dropdown-Menüs zwischen linear und logarithmisch wechseln.
5. Auch die Datenquelle, die auf den Kreisradius abgebildet wird, lässt sich austauschen. Hier ist es ebenfalls möglich, den Skalierungsfaktor zu ändern, das heißt alle Kreise entsprechend zu vergrößern oder zu verkleinern.
6. Eine weitere Interaktion ist, mehrere Länder mit Hilfe von den Checkboxes auf der rechten Seite auszuwählen. Die selektierten Länder werden hervorgehoben und mit dem Ländernamen beschriftet. Die restlichen werden etwas transparent dargestellt. Wenn man Länder ausgewählt hat, wird ein Schieberegler aktiviert. Durch diesen kann dann die Transparenz der ausgewählten Länder bestimmt werden.
7. Mittels eines Klicks auf den „Map“-Tab, wird eine alternative Darstellung angezeigt. Die Kreise werden dann auf einer Weltkarte dargestellt, wie Abbildung 5.7 zeigt. Dadurch, dass nun die x- und y-Dimensionen für die Kartendarstellung benötigt werden, gehen diese für die dargestellten Informationen verloren. Es können dadurch weniger Datenquellen gleichzeitig dargestellt werden.

„Wer wählt was“-Visualisierung

Ein weiteres Beispiel, ebenfalls zur Bundestagswahl 2013, ist die Visualisierung „Wer wählt was“ [23] von der deutschen Bundeszentrale für politische Bildung. Wie in Abbildung 5.8 zu sehen ist, kann durch verschiedene Kriterien (unter anderem die Wahlentscheidung oder das Geschlecht) herausgefunden werden, welchen Gruppen die Wähler der einzelnen Parteien angehören.

Zu den *Interaktionsarten* in diesem Diagramm gehören die untenstehenden.

1. Eine Interaktionsmöglichkeit stellt das Drag-and-Drop dar. Sobald man die einzelnen Kriterien über den inneren oder äußeren Kreis zieht, wird das Diagramm aktualisiert.
2. Auch bei diesem Beispiel gibt es die Möglichkeit mehr Informationen durch Hovern zu erhalten. Der dann erscheinende Tooltip (siehe Abbildung 5.9) zeigt die exakten Prozentwerte an, in diesem Beispiel für die männlichen SPD-Wähler sowie ein weiteres Diagramm mit Daten von vorherigen Wahlen. Alle anderen Kreisabschnitte und Beschriftungen werden ausgegraut.
3. Die dritte Art zu interagieren, selektiert die Umfragedaten, welche visualisiert werden. Dazu kann man mit Hilfe von zwei Schaltflächen (auch *RadioButtons* genannt) entscheiden, ob die Ergebnisse aller Befragten oder nur die der Befragten mit Parteipräferenz berücksichtigt werden sollen (siehe Abbildung 5.10).

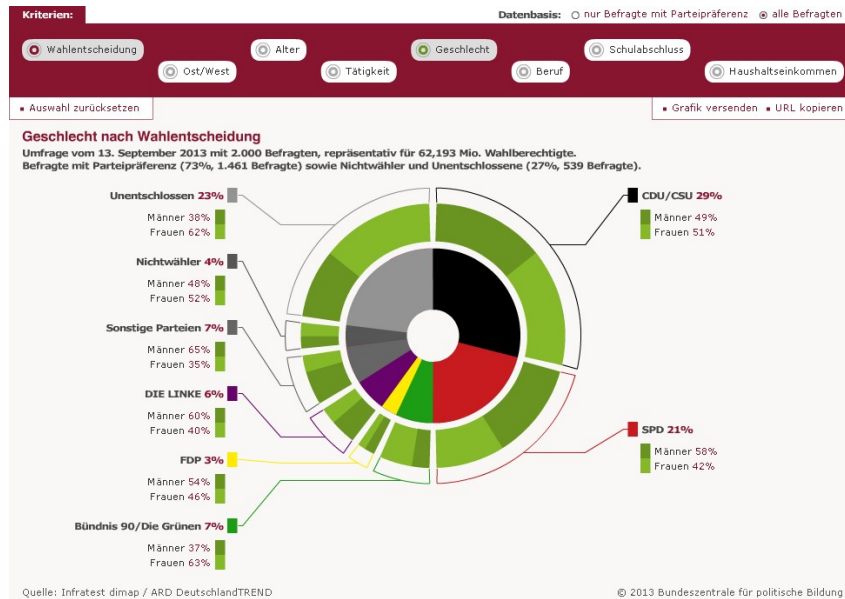


Abbildung 5.8: „Wer wählt was“-Visualisierung im Überblick. Quelle: Infratest / ARD DeutschlandTREND und Bundeszentrale für politische Bildung [23]

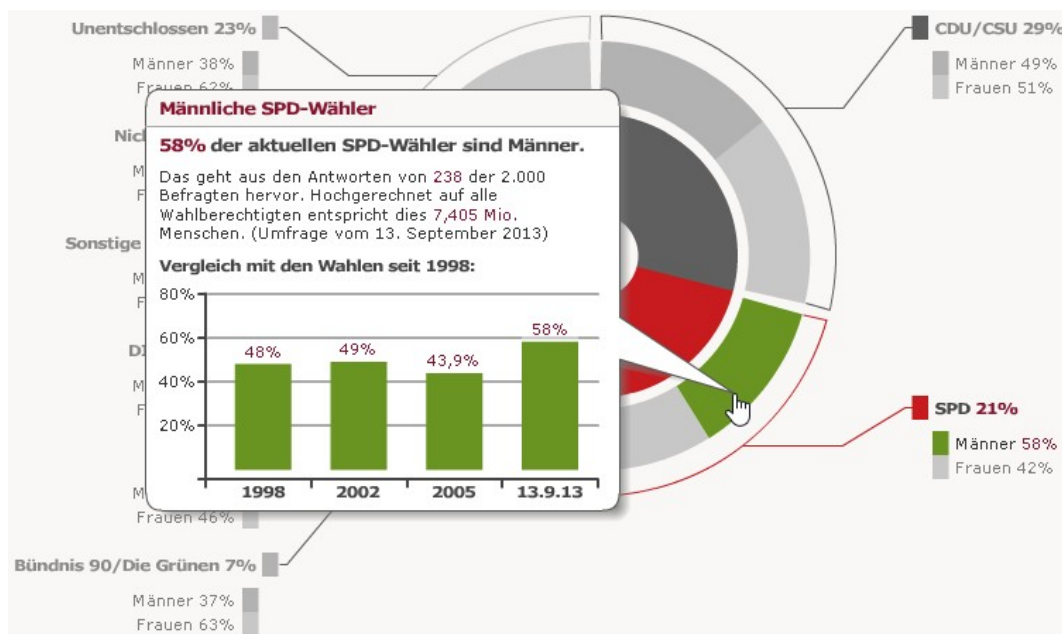


Abbildung 5.9: Tooltip bei der „Wer wählt was“-Visualisierung. Quelle: Infratest / ARD DeutschlandTREND und Bundeszentrale für politische Bildung [23]

Datenbasis: nur Befragte mit Parteipräferenz alle Befragten

Abbildung 5.10: RadioButtons bei der „Wer wählt was“ Visualisierung. Quelle: Infratest / ARD DeutschlandTREND und Bundeszentrale für politische Bildung [23]

5.1.3 Unterteilung von Interaktionskonzepten

Im Rahmen dieser Arbeit werden Interaktionskonzepte in (1) *datenabhängige* und (2) *visuelle* Interaktionskonzepte unterteilt.

1. Zu den *datenabhängigen* Interaktionen gehören alle Interaktionen, mit denen die Daten verändert werden. Durch diese Datenveränderung wandelt sich wiederum die Visualisierung um. Diese Interaktionen, welche die Daten selektieren, können mit folgenden Eingabewidgets und Methoden vorgenommen werden:
 - Drag-and-Drop
 - RadioButton
 - CheckBox
 - Schaltfläche (auch *Buttons* genannt)
 - Zeitfenster
 - Play-Schieberegler
 - Dropdown-Menü
2. Bei den *visuellen* Interaktionen wird das Diagramm im optischen Aussehen verändert. Dies kann beispielsweise durch Klick- oder Hover-Events ausgelöst werden. Zu den visuellen Interaktionskonzepten zählen:
 - Tooltip
 - Hervorheben
 - Skalierung ändern
 - alternative Darstellung

Bei vielen Visualisierungen [21] sind nicht *nur* datenabhängige Interaktionskonzepte oder *nur* visuelle Interaktionskonzepte integriert, sondern meistens sind beide Konzepte vereint in einer Grafik zu finden.

5.2 Konzeptvorschläge

Um die im vorherigen Kapitel extrahierten Interaktionskonzepte zu unterstützen, wurden in dem Projekt *Flower* die folgenden Ansätze integriert:

1. Zum einen wird eine Methode erläutert, um mit Hilfe von Eingabeelementen den Datenfluss zu parametrisieren und dadurch die Visualisierung ebenfalls zu beeinflussen. Für dieses datenabhängige Konzept wurde ein neuer Komponententyp namens *InteractionPanel* entwickelt.
2. Zum anderen wird der *MorphCreator* erweitert, um visuelle Interaktionen zu erstellen.

Bei der Umsetzung der Interaktionskonzepte wurde darauf geachtet, redundanten Quelltext zu vermeiden und die Interaktion in wenigen Arbeitsschritten (zum Beispiel durch Beschreibungen) zu erstellen.

5.2.1 Datenabhängige Interaktionen

Einwohnerzahldiagramm

Im Folgenden wird ein Balkendiagramm beschrieben, in dem mit einem Schieberegler die Jahreszahl selektiert werden kann. Diese Visualisierung ist auf der rechten Seite der Abbildung 5.11 zu sehen. Um den Fokus auf die Integration des datenabhängigen Interaktionskonzepts zu legen, ist bewusst ein einfaches Beispiel gewählt worden. Bei diesem Diagramm werden die Einwohnerzahlen von zehn verschiedenen Ländern für die Jahre von 1960 bis 2013 dargestellt. Jeder Balken steht für eines der Länder. Die Höhe der Balken gibt die Einwohnerzahl in diesem Land an. Im oberen rechten Bereich der Abbildung ist das *InteractionPanel* zu sehen, welches den Schieberegler enthält. Neben dem Regler sind noch weitere Felder für die Variablenbezeichnung sowie den aktuellen Wert des Reglers vorhanden. Mit Hilfe des Schiebereglers können die einzelnen Jahreszahlen selektiert werden, sodass der Verlauf der Daten dargestellt wird.

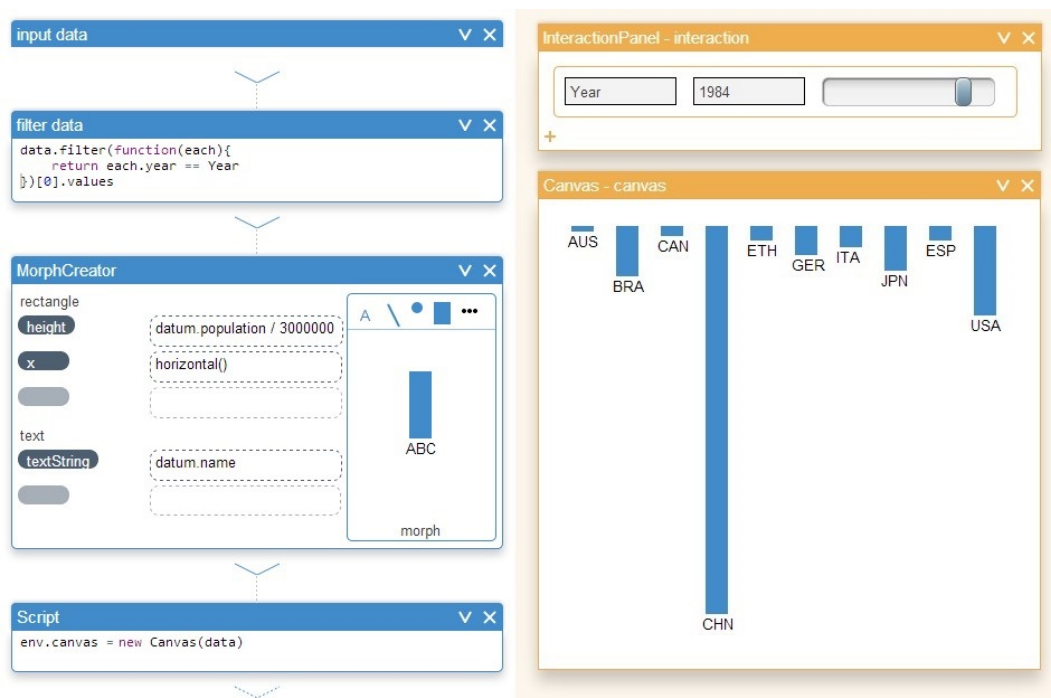


Abbildung 5.11: Einwohnerzahldiagramm in *Flower*

```
[
  {
    "year": 1960,
    "values": [
      {
        "name": "AUS",
        "population": 10286184
      },
      {
        "name": "CAN",
        "population": 17909009
      },
      {
        "name": "GER",
        "population": 72814899
      }
    ]
  },
  ...
]
```

Abbildung 5.12: JSON-Rohdaten

Datenfluss

Auf der linken Seite der Abbildung 5.11 ist das Programm zu sehen, welches dafür benötigt wird. In der ersten, eingeklappten *Script*-Komponente namens „input data“ sind die Rohdaten enthalten, welche für das Balkendiagramm benötigt werden. Die Daten haben die Struktur aus Abbildung 5.12. Sie liegen im JSON-Format vor. Im zweiten *Script* („filter data“) werden die Daten nach einem Jahr gefiltert. Dabei wird zunächst eine Beispieljahreszahl genutzt, später wird diese durch eine noch zu erstellende Interaktionsvariable `Year` ersetzt. Diese Variable beinhaltet später den Wert des Reglers. Im nächsten Schritt werden im *MorphCreator* die Zuordnungen vorgenommen. Für genauere Beschreibungen des *MorphCreator* sei auf 4.3 dieser Arbeit verwiesen. Anschließend werden in der untersten Komponente die Daten auf das *Canvas* geschrieben.

Sowie der Datenfluss für dieses Beispiel komplett erstellt wurde, können die Interaktionselemente – im Beispiel der Schieberegler – hinzugefügt werden. Dazu klickt man auf die Plus-Schaltfläche im *InteractionPanel* und wählt den „Slider“ aus, wie auf Abbildung 5.13a zu sehen ist. Nun hält man einen Schieberegler an der Maus (siehe Abbildung 5.13b). Sobald dieser auf das *InteractionPanel* fallen gelassen wird, entstehen weitere Felder für den Variablennamen und den aktuellen Wert des Reglers. Dies zeigt Abbildung 5.13c. Der Variablenname kann in `Year` umbenannt werden. Es kann nun die Beispieljahreszahl im Datensatz durch die Interaktionsvariable `Year` ersetzt werden. Des Weiteren müssen noch Einstellungen für den Schieberegler getroffen werden. Der standardmäßige Wertebereich reicht von 0 bis 1. Das Verändern des Wertebereichs sowie der Schrittweite werden über den *Morphic-Inspektor* und anschließendes Evaluieren von Quelltext vorgenommen. Dieser Prozessschritt ist verbesserungswürdig. Im Kapitel 5.4.3 „Ausblick“ werden andere Möglichkeiten erläutert, um Eingabelemente benutzerfreundlicher zu gestalten.

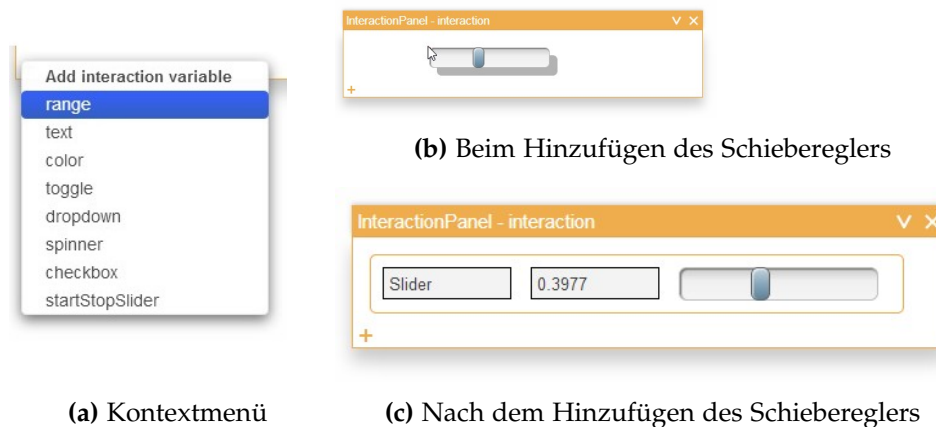


Abbildung 5.13: Erstellen des Schiebereglers und der Interaktionsvariable Year

InteractionPanel

Um datenabhängige Interaktionen mit Hilfe unseres Tools zu erstellen, verwendet man das *InteractionPanel*. Dieses Panel ermöglicht es den Datenfluss und somit auch die Visualisierung zu beeinflussen. Durch die Neuevaluierung wird das Programm variabel.

Das *InteractionPanel* ist ein Komponententyp, der nur im Dashboard benutzt wird. Es besteht aus einer *DroppingArea*, welcher beliebige Morphs hinzugefügt werden können. Es gibt drei Möglichkeiten um Interaktionsvariablen hinzuzufügen:

1. Über einen Button können, wie bereits im vorherigen Beispiel beschrieben, Standardelemente gewählt werden. Diese sind bereits vorkonfiguriert und ermöglichen eine schnelle Nutzung.
2. Über den *PartsBin* [5] können von anderen Lively-Nutzern erstellte Eingabeelemente verwendet werden. Diese können in der *DroppingArea* abgelegt werden. Der Nutzer wird anschließend nach dem Attribut des Eingabeelements gefragt, welches den Wert der Variable liefert.
3. Als dritte Alternative können Nutzer selbst Morphs kombinieren und daraus Eingabewerkzeuge erstellen. Die Verwendung ist dabei analog zu Punkt 2.

Das *InteractionPanel* erfüllt durch die unterschiedlichen vorhandenen Eingabeelemente diverse Verwendungsmöglichkeiten. Im Rahmen unseres Projektes wurden aus der HTML5 Spezifikation [24] folgende – für die Visualisierungserstellung nützliche – Eingabeelemente ausgewählt (siehe Abbildung 5.14):

- **Schieberegler, Spinner** Wenn die Daten für eine Visualisierung für mehrere Zahlen (-bereiche) vorliegen, kann man durch einen Schieberegler oder einen Spinner einen Bereich auswählen und der weitere Datenfluss arbeitet nur mit diesem Teil der Daten. Beide Eingabetypen unterscheiden sich im Aussehen. Des Weiteren sind Regler für große, annähernd kontinuierliche Wertebereiche

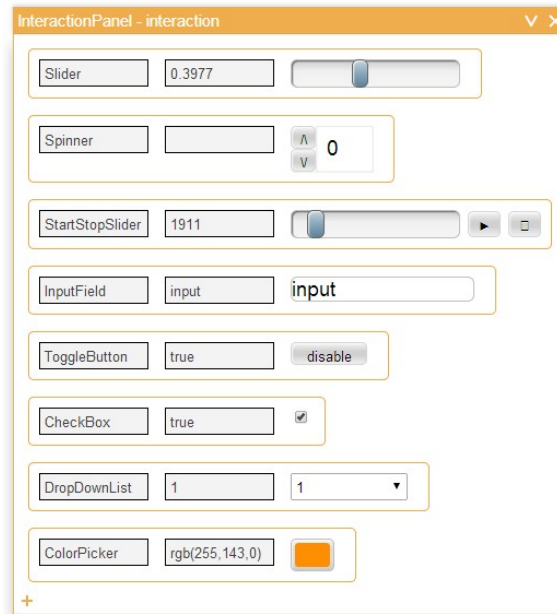


Abbildung 5.14: Eingabelemente des *InteractionPanel*s

geeignet, weil man schnell von einer Grenze zur anderen navigieren kann. Ein Spinner eignet sich hingegen eher für kleinere, meist diskrete Wertebereiche, da es erforderlich ist für jeden Wert einmal zu klicken, um zum nächsten Wert zu gelangen. Ein Schieberegler wurde beispielsweise bei der *Gapminder World* beim Ändern der Transparenz der nicht selektierten Länder verwendet.

- **Start-Stop-Slider** Dieser Slider ist eine Erweiterung des Schiebereglers.
- **Textfeld** Das Textfeld eignet sich, um benutzerdefinierte Filterexpressions zu integrieren.
- **ToggleButton, CheckBox, RadioButton** Diese Eingabelemente liefern die Werte `true` und `false`. **ToggleButton** und **CheckBox** unterscheiden sich nur im Aussehen. Beim **RadioButton** gibt es zusätzlich die Bedingung, dass immer nur ein **RadioButton** gleichzeitig ausgewählt sein kann. Ein Beispiel für die Verwendung von **ToggleButtons** sind die Hashtags beim *Twitter-Monitor* Beispiel aus Kapitel 5.1.2.
- **Dropdown-Liste** Wenn Rohdaten für mehrere Zeichenketten (wie Ländernamen) existieren, könnte das Programm abhängig von der Dropdown-Liste sein. Zum Beispiel wurde diese Interaktionsmöglichkeit in der *Gapminder World* für die Auswahl der Daten genutzt (siehe Punkt 3, Seite 86).
- **Farbauswahlknopf** Mit Hilfe der Farbauswahl kann man die Farbe von den visuellen Elementen im Nachhinein leicht anpassen. Das Finden von passenden Farbtönen wird dadurch vereinfacht. Dies ist gerade im Kontext der `range`-Funktion (siehe Kapitel 4.3.3) nützlich.

Allgemein bietet das *InteractionPanel* die Möglichkeit Variablen zu definieren, die jeweils mit einem zugehörigen Eingabeelement verknüpft sind. Diese können überall im Datenfluss verwendet werden.

5.2.2 Visuelle Interaktionen

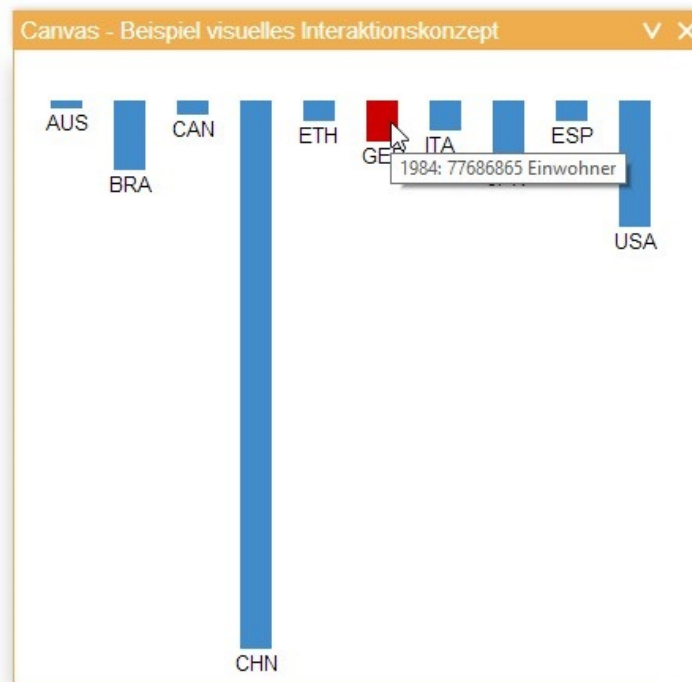


Abbildung 5.15: Beispiel für ein visuelles Interaktionskonzept: Tooltip und Hervorheben

Beispiel: Tooltip und Hervorheben

Neben der Integration von datenabhängigen Interaktionen, wie in Kapitel 5.2.1 beschrieben, wurde im Projekt *Flower* mit dem Erstellen von Interaktionen im *MorphCreator* ein Konzept für visuelle Interaktionen umgesetzt. Um einen Überblick über das Erstellen von diesen visuellen Interaktionsmöglichkeiten zu erhalten, wird das Beispiel aus dem vorherigen Kapitel fortgesetzt. Wenn sich die Maus über einem Balken befindet (im Folgenden *hovern* genannt), soll ein Tooltip angezeigt werden sowie die einzelnen Balken hervorgehoben werden. Der Tooltip enthält die Jahreszahl sowie die Einwohnerzahl von diesem Land zum gewählten Zeitpunkt. Hovert man über den Deutschland-Balken, wie bei Abbildung 5.15 zu sehen, so entsteht ein Tooltip mit „2013: 81804228 Einwohner“. Beim Hovern soll nicht nur die Kurzinformation angezeigt werden, sondern auch das aktuell betrachtete Land

hervorgehoben werden. In der Abbildung wird das aktuelle Land durch Rotfärben des Balkens betont.

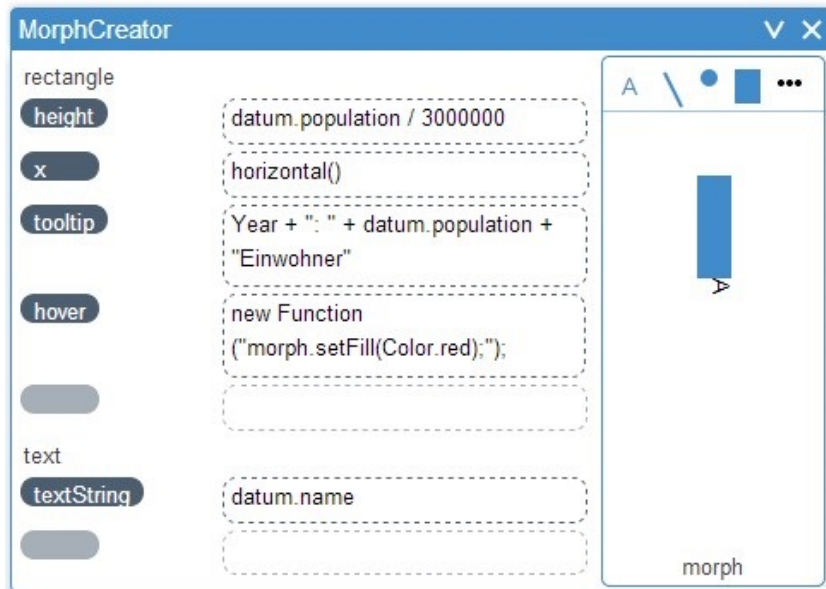


Abbildung 5.16: MorphCreator (Beispiel Datenfluss für Tooltip und Hervorheben)

Datenfluss

Um den Datenfluss für das oben beschriebene Diagramm mit Tooltip und Hervorhebung zu erreichen, werden weitere Zuordnungen im *MorphCreator* (wie Abbildung 5.16 zeigt) hinzugefügt. Es gibt das Attribut `tooltip`, das auf der linken Seite im *MorphCreator* steht. Auf der dazugehörigen rechten Seite befindet sich die Zeichenkette, welche beim Hovern über den Balken in der Visualisierung im Tooltip angezeigt wird. Weiterhin gibt es die Funktion `hover`, welche als Mapping-Attribut auf der linken Seite der Deklaration des *MorphCreators* steht. Auf der rechten Seite ist eine eigene Funktion implementiert, welche die Füllfarbe des Morphs auf rot ändert. Stattdessen können auch andere Methoden aufgerufen werden, um das Hervorheben anders zu implementieren. Beispielsweise kann der Rand mit dem Aufruf `morph.setBorderSize(3)` vergrößert werden. Wie genau das visuelle Interaktionskonzept implementiert ist, wird in Kapitel 5.3.2 erläutert.

Erweiterung des MorphCreators

Wie in dem Kapitel 4 beschrieben, werden in der *MorphCreator*-Komponente Zuordnungen zwischen Eigenschaften eines Morphs und deren Werten gesetzt. Um visuelle Interaktionskonzepte mittels unserer Software in eine Visualisierung zu integrieren, wird dieser *MorphCreator* mit Events erweitert. Events (wie `onClick`, `onMouseOver` oder `onMouseMove`) eines Morphs sind spezielle Funktionen des Morphs.

Der *MorphCreator* ermöglicht für diese Trigger ebenfalls eine Zuordnung. Welche Aktionen für das jeweilige Event ausgeführt werden, kann durch beliebigen JavaScript-Code auf der rechten Seite der Zuordnungen beim *MorphCreator* festgelegt werden. Es besteht ebenfalls die Möglichkeit, neben den vorgefertigten Triggern eigene zu schreiben. Zu den vorgefertigten Triggern gehören unter anderem `click`, `contextmenu`, `hover` sowie `tooltip`.

5.3 Implementierung

In diesem Abschnitt wird auf einzelne Implementierungsdetails für datenabhängige und visuelle Interaktionen eingegangen. Dies sind nur Quelltextausschnitte. Der gesamte Code von *Flower* ist im Github-Repository [2] zu finden.

5.3.1 Datenabhängige Interaktionen

Bei der Implementierung des *InteractionPanels* werden folgende Details vorgestellt:

- Aufbau des *InteractionPanels*
- Erstellen von Variablen
- Verwenden der Interaktionsvariablen
- Entfernen von Variablen

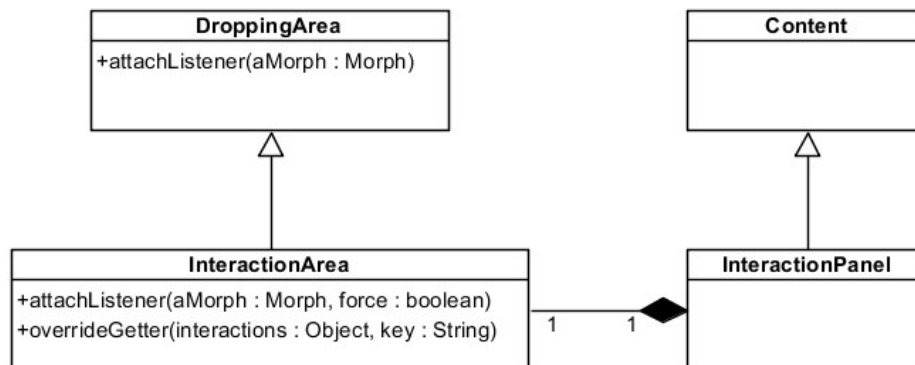


Abbildung 5.17: Klassendiagramm des *InteractionPanels* mit einigen relevanten Methoden

Aufbau des InteractionPanels

Das *InteractionPanel* ist ein weiterer Komponententyp, der nur im Dashboard eingesetzt wird. Es erbt von der *Content*-Klasse wie beispielsweise auch das *Script*, der *MorphCreator* oder das *Canvas*.

Das *InteractionPanel* erstellt im `initialize` eine *InteractionArea*. Sie ist ein Bereich, wo beliebige *Morphs* fallen gelassen werden können. Auf diese Art und Weise können Interaktionsvariablen angelegt werden. Wie dies genau implementiert ist, wird im nächsten Abschnitt auf Seite 99 erklärt.

Die *InteractionArea* erbt von der *DroppingArea*, die auch an anderer Stelle (innerhalb der *PrototypeArea* des *MorphCreators*) verwendet wird. Dabei überschreibt *InteractionArea* die Methoden `attachListeners` und `getContextMenuComponents`. In `attachListeners` werden, wie oben beschrieben, einige Methoden des neu erstellten Eingabeelements überschrieben. Die Funktion `getContextMenuComponents` liefert alle Menüeinträge, die erscheinen, wenn auf die Plusschaltfläche gedrückt wird. In den folgenden Abschnitten werden die wichtigen Methoden vorgestellt und es wird auf Implementierungsdetails eingegangen.

Quelltext 5.1: Auszug aus `attachListeners`: Erstellung von Verbindungen für die Interaktionsvariable in `createContainer`

```

1  attachListener: function(aMorph, force) {
2    // ...
3    var attribute = this.getConnectionAttribute();
4    aMorph.remove = function() { /* ... */ }
5    aMorph.onDropOn = function() { /* ... */ }
6    aMorph.createContainer = function() {
7      // ...
8      connect(nameField, "textString", this, "setName");
9      connect(this, attribute, valueField, "setTextString");
10   }
11   aMorph.createInteractionVariable = function() {
12     // ...
13     dashboard.env.interaction[name] = attribute;
14     this.interactionConnections = [
15       connect(this, attribute,
16         dashboard.env.interaction, name),
17       connect(this, attribute,
18         this, "updateObservers")
19     ];
20     interactionArea.overrideGetter(dashboard.env.interaction, aMorph.
21       getName());
22   }
23 }

```

Erstellen von Variablen

Das Anlegen der Interaktionsvariablen ist in der Methode `attachListener` implementiert. Im Codebeispiel 5.1 werden ein paar Auszüge aus dieser Methode dargestellt. Die Variable `aMorph` ist der `Morph`, der mit Hilfe des Kontextmenüs des *InteractionPanels* erstellt wurde. Diesem werden unter anderem die Funktionen `remove`,¹ `onDropOn`, `createContainer` sowie `createInteractionVariable` hinzugefügt. Bei der `onDropOn`-Methode wird sichergestellt, dass sich bei selbsterstellten `Morphs` ein Eingabefeld öffnet. Dieses fragt den Nutzer, wie man von diesem Eingabeelement den Wert für die Variable bekommt. Beispielsweise sollte bei einem Textfeld das Attribut `setTextString` und bei einem Dropdown-Menü `selection` eingefügt werden. Weiterhin werden in den Methoden `createContainer` und `createInteractionVariable` mehrere Verbindungen mittels `lively.connect`² angelegt. Diese verknüpfen das Eingabeelement mit:

- einem Feld, welches den aktuellen Wert anzeigt (vgl. Quelltext 5.1, Zeile 8);
- einem Feld, welches den aktuellen Variablenamen anzeigt (vgl. Quelltext 5.1, Zeile 9) – dieser Name der Interaktionsvariablen kann im Programm verwendet werden;
- der internen Speicherung der Interaktionsvariablen (vgl. Quelltext 5.1, Zeile 13) – diese Verbindungen werden zusätzlich in einem Array gespeichert, um sie beim Entfernen der Variable löschen zu können.

Quelltext 5.2: `overrideGetter`-Funktion

```

1  overrideGetter: function(interactions, key) {
2      interactions["_" + key] = interactions[key];
3      interactions.__defineGetter__(key, function() {
4          window.activeComponent.interactionVariables.pushIfNotIncluded(key
5              );
6          return interactions["_" + key];
7      });
8      interactions.__defineSetter__(key, function(value) {
9          interactions["_" + key] = value;
10     })

```

¹Siehe Abschnitt 5.3.1.

²Siehe Abschnitt *Data bindings* in [25].

Verwenden der Interaktionsvariablen

Um das Verwenden der Interaktionsvariablen zu ermöglichen, wurde das klassische Observer-Pattern [26] implementiert. Es werden somit die Zugriffsfunktionen mittels `__defineGetter__` beziehungsweise `__defineSetter__` (vgl. Quelltext 5.2) überschrieben. Wenn innerhalb einer Komponente eine Interaktionsvariable verwendet wird, wird sie bevor der Wert der Variable zurückgegeben wird, in den verwendeten Interaktionsvariablen dieser Komponente gespeichert. Der Wert muss nur beim erstmaligen Verwenden der Interaktionsvariablen gespeichert werden.

Sobald sich eine Interaktionsvariable ändert, werden alle Komponenten informiert, die diese Variable verwenden. Dadurch wird der Datenfluss neu evaluiert. In der folgenden Aufzählung werden andere mögliche Lösungsansätze vorgestellt:

- Eine Möglichkeit – wenn mehrere Komponenten die gleiche Variable benutzen – ist, nur die allererste Komponente im Datenfluss zu benachrichtigen. Dadurch werden aber überflüssige Schritte, die potentiell rechenintensiv sind, auch neu durchgeführt.
- Die Alternative wäre, nur die Komponente zu informieren, die in der Ausführungsreihenfolge vorn liegt. Allerdings gibt es bei dieser Variante viele Ausnahmefälle zu beachten. Bei mehreren Ausführungssträngen können Komponenten in verschiedenen Strängen die gleiche Interaktionsvariable verwenden. In diesem Fall sollten alle Komponenten benachrichtigt werden. Oder es wird die erste Komponente benachrichtigt, die den Datenfluss in beide Stränge unterteilt. Da die Struktur des Programmes nicht gespeichert wird, kann nur mit erheblichem Aufwand herausgefunden werden, ob zwei Komponenten in verschiedenen Strängen die Interaktionsvariable verwenden oder welche die erste gemeinsame Komponente ist.

Um den Zugriff auf die Interaktionsvariablen innerhalb der Komponenten zu vereinfachen, wird der Sichtbarkeitsbereich mit Hilfe des `with`-Statements³ (*with env.interactions*) um diese erweitert.

Entfernen von Variablen

Um beim Entfernen der Interaktionsvariablen neues Verhalten hinzuzufügen, wird die `remove`-Methode der Eingabemorphs erweitert. Dafür muss sie zunächst zwischengespeichert und danach innerhalb der neuen Funktion evaluiert werden. Dieser Umweg wird benötigt, weil die vorherige Methode das Löschen des Morphs übernimmt. Aus diesem Grund wird sie zwischengespeichert und nach den zusätzlichen Aktionen ausgeführt. Dieses Erweitern geschieht innerhalb der Funktion `attachListeners`, wie Quelltext 5.3 zeigt. Falls der Eingabemorph aus dem *InteractionPanel* entfernt wird, wird die Methode `removeVariable` aufgerufen. Außerdem werden alle Verbindungen getrennt, die zuvor mit `lively.connect` angelegt wurden. Diese Funktion löscht die in *env.interactions* gespeicherten Verweise der zu entfernenden Variable. Am Ende wird das Eingabeelement entfernt.

³<http://www.ecma-international.org/ecma-262/5.1/#sec-12.10>; besucht am 01. Oktober 2014.

Quelltext 5.3: Überschreiben der remove-Funktion in attachListeners

```

1 // attach remove -> remove interaction variable
2 var oldRemove = aMorph.remove;
3
4 aMorph.remove = function() {
5     if (this.owner instanceof lively.morphic.Charts.InteractionArea) {
6         this.owner.removeVariable(this.getName());
7         if (this.interactionConnections)
8             this.interactionConnections.each(function(ea) {
9                 ea.disconnect();
10            });
11    }
12    oldRemove.apply(aMorph, arguments);
13 }

```

5.3.2 Visuelle Interaktionen

Um visuelle Interaktionen zu implementieren, werden Events benötigt. Deshalb wird in diesem Kapitel zunächst vorgestellt, wie Events in Lively implementiert sind. Danach wird auf einige Anpassungen bei den Event-Deklarationen eingegangen, die im *MorphCreator* vorgenommen wurden.

Events in Lively

Bei den Eingabetypen wird in Lively [25] zwischen Maus- und Tastatureingaben unterschieden. Die aktuelle Lively-Version⁴ bietet für Events eine Abstraktionsebene vom Document Object Model [27] (DOM), indem Events pro Morph behandelt werden. Pro Morph werden dadurch von einem Eventhandler mehrere Funktionen (wie zum Beispiel `onClick`) implementiert. Diese Funktionen werden automatisch getriggert und bekommen ein Event-Objekt als Parameter übergeben.

Event-Deklarationen

Die Event-Deklarationen (*Mappings*) werden wie alle anderen Deklarationen im *MorphCreator* implementiert, wie in Abschnitt 4.4.2 auf Seite 76 zu sehen. Um eine große Nutzergruppe anzusprechen, wurden im Projekt *Flower* neben den Lively-Eventbezeichnungen auch die Funktionsnamen der Events von D3.js⁵ [28] integriert, einer weitverbreiteten Visualisierungsbibliothek in JavaScript [32]. Die Tabelle 5.1 zeigt die Unterschiede der Eventnamen zwischen D3.js und Lively. Bei einigen Events in Lively existieren zusätzlich zu den normalen Events noch Eingangsevents, welche beim ersten Aufruf des normalen Events getriggert werden.

⁴In Modulen *lively.morphic.Events* außer *Textevents* (im Modul *lively.morphic.Text*). Siehe <https://github.com/LivelyKernel/LivelyKernel>; besucht am 23. Juni 2014.

⁵Data Driven Development.

Tabelle 5.1: Vergleich der Bezeichnung von D3.js und Lively

D3.js	Lively intern ¹
click	onClick
dblclick	onDoubleClick
contextmenu	onMouseWheel
—	onMouseWheelEntry
mousemove	onMouseMove
—	onMouseMoveEntry
mouseout	onMouseOut
mouseover	onMouseOver
mouseup	onMouseUp
—	onMouseUpEntry
mousedown	onMouseDown
—	onMouseDownEntry

¹ core/lively/morphic/Events.js in Morph (Extensions)

Quelltext 5.4 zeigt im oberen Teil wie die zusätzlichen Funktionsbezeichnungen implementiert wurden. Die `getAttributeMap`-Funktion liefert Funktionen, die im *MorphCreator* auf der linken Seite der Zuordnungen verwendet werden können.

Neben den bisher beschriebenen Events, stellt *Flower* dem Anwender das `hover`-Event zur Verfügung. Dieses ist eine Komposition des `mouseover`- und `mouseout`-Events. Die Implementierung geschieht innerhalb der `getAttributeMap`-Funktion (siehe Quelltext 5.4, unterer Teil). Zunächst müssen häufig verwendete Eigenschaften gespeichert werden. Anschließend werden die `onMouseOver` sowie die `onMouseOut`-Methoden überschrieben. Beim `onMouseOut` werden die zu Beginn gespeicherten Eigenschaften wieder gesetzt. Damit soll der Zustand wiederhergestellt werden, in welchem sich der Morph vor dem `onMouseOver`-Event befand.

5.4 Ausblick und Diskussion

Dieses Kapitel beschreibt Konzepte, die noch nicht im Projekt *Flower* integriert sind, die für die Nutzerfreundlichkeit jedoch sehr wertvoll wären. Es wird zunächst die Problemstellung der noch nicht implementierten Aspekte erläutert. Anschließend werden die möglichen Lösungen vorgestellt und diskutiert. Falls es zu dieser Problematik schon existierende Lösungen gibt, werden diese ebenfalls erwähnt.

5.4.1 Canvas als Werkzeug zur Visualisierungserstellung

Das *Canvas* ist bisher dafür zuständig, visuelle Elemente anzuzeigen. Es wäre denkbar, dass das *Canvas* zusätzlich Aufgaben, wie das Anpassen der Positionierung der Morphs oder Skalen oder das Verändern von anderen visuellen Variablen (wie beispielsweise Größe, Farbe oder Transparenz) übernehmen könnte. Eine Möglichkeit

Quelltext 5.4: dbclick und hover innerhalb der getAttributeMap-Funktion

```

1 getAttributeMap: function() {
2   return { // ...
3     dbclick: function(value, morph) { morph.onDoubleClick = value; },
4     hover: function(valueFn, morph, datum) {
5       // save standard properties
6       var stdColor, stdPosition, stdBorderWidth;
7       setTimeout(function(){
8         stdColor = morph.getFill();
9         stdPosition = morph.getPosition();
10        stdBorderWidth = morph.getBorderWidth();
11      }, 10);
12      morph.onMouseOver = valueFn(datum);
13      morph.onMouseOut = function(){
14        morph.setFill(stdColor);
15        morph.setPosition(stdPosition);
16        morph.setBorderWidth(stdBorderWidth);
17      }
18    },
19  };
20 }

```

wäre es, wenn diese Abwandlung direkt im *Canvas* durchgeführt werden könnte. Die Modifikationen im *Canvas* sind gleichzusetzen mit einem Teil Quelltext im bisherigen Datenfluss beziehungsweise Änderungen am Prototypen im *MorphCreator*. Dieser sollte im Programm automatisch ergänzt werden, wenn der Nutzer im *Canvas* etwas bearbeitet. Dieser Prozess ermöglicht es, die Visualisierung nach dem Erstellen noch zu bearbeiten und einige differenziertere Einstellungen zu treffen.

Lösungsansätze

Ein Ansatz, das Erstellen der Visualisierung im *Canvas* durchzuführen, ist von Bret Victor entwickelt worden. In seinem Tool entsteht die Visualisierung fast ausschließlich durch Interaktionen direkt auf dem *Canvas*. Dieses Programm stellte er in dem Talk „Drawing Dynamic Visualizations“ [29, 30] im Februar 2013 vor. Bret Victor erklärt, dass mit seinem Tool das direkte Manipulieren der visuellen Elemente im Vordergrund steht. Dieses Manipulieren ist weiterhin datengetrieben und kann durch Parameter gesteuert werden.

Die Oberfläche der Visualisierungssoftware (siehe Abbildung 5.18) gliedert sich in drei Hauptbereiche:

1. oben eine Übersicht von allen Visualisierungen,
2. darunter links die Daten, Ablauf und die Aktionen,
3. rechts daneben das Canvas mit einer Liste von Tastaturkürzeln.

5 Interaktionskonzepte

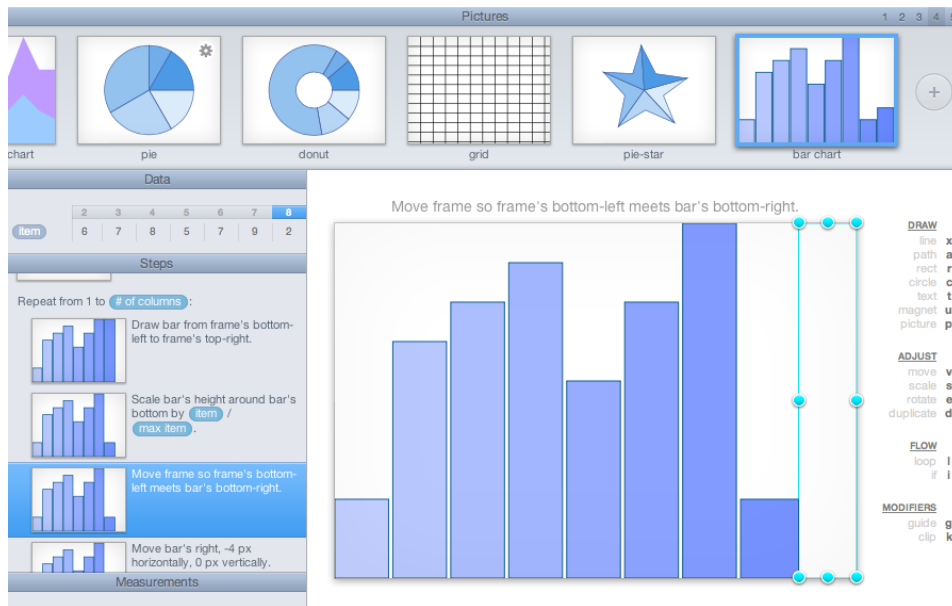


Abbildung 5.18: *Drawing Dynamic Visualizations* – Software von Bret Victor. Quelle: Bret Victor, Drawing Dynamic Visualization Video [30]

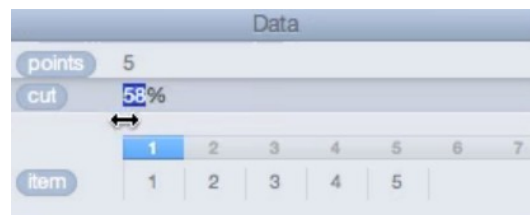


Abbildung 5.19: Schieberegler über einem numerischen Wert zum Verändern der Daten. Quelle: Bret Victor, Drawing Dynamic Visualization Video [30]



Abbildung 5.20: Untermenü zum Manipulieren der Attribute von Grundformen. Quelle: Bret Victor, Drawing Dynamic Visualization Video [30]

Rotate line around line's start by 0.5 /

Abbildung 5.21: Ein Arbeitsschritt in Bret Vectors Tool. Quelle: Bret Victor, Drawing Dynamic Visualization Video [30]

Die Tastenkürzel erlauben es, einfache Grundformen (wie Kreis, Rechteck, Linie) schnell zu erstellen. Dazu wird der entsprechende Buchstabe gedrückt und dann die Größe auf dem Canvas festgelegt. Weitere Veränderungen sind ebenfalls durch Tastenkürzel und anschließendes Verschieben von dadurch auftauchenden Kontrollpunkten möglich. „Snapping Points“ helfen beim exakten Positionieren. Um dem Nutzer die Möglichkeit zu geben, schnell und einfach einen numerischen Wert zu verändern, kann dieser die Maus über die Zahl bewegen. Nun erscheint anstatt des Mauszeigers ein Slider, wie in Abbildung 5.19 zu sehen. Bewegt man diesen, so ändert sich auch die Zahl und alle weiteren Eigenschaften, die mit der Nummer zusammenhängen.

Um Eigenschaften von den gezeichneten Grundformen zu korrigieren, kann mit einem Doppelklick ein Kontextmenü aufgerufen werden. Dieses Menü ist in Abbildung 5.20 sichtbar. Dort erscheinen Standardattribute wie die Füll- und die Rahmenfarbe sowie die Rahmenbreite.

Jeder Schritt wird im Ablauf auf der linken Seite des Prototyps festgehalten. Der aktuelle Arbeitsschritt steht auch oberhalb des *Canvas*, wie beispielsweise in Abbildung 5.21 zu sehen. Um eine Aktion zu parametrisieren, zieht man eine Variable per Drag-and-Drop in den Satz über dem *Canvas*. In der Abbildung wurde die Variable *points* hineingezogen. Der aktuelle Task lässt sich auch durch Eintippen von Zahlen verändern.

Ein weiteres Mittel, um die Diagrammerstellung zu beschleunigen, ist es, wiederkehrende Aufgaben in der Ablaufseite zu markieren. Diese können dann durch Tippen von „1“ automatisch wiederholt werden. Auch diese Schleife lässt sich parametrisieren. Dazu nutzt man, wie beim Parametrisieren des aktuellen Arbeitsschrittes, Drag-and-Drop.

5.4.2 Zoom

Ein Bestandteil vieler Visualisierungen (siehe Beispielvisualisierungen [21]) ist das Verwenden von Zoomen. Dies bedeutet, dass es unterschiedliche Detailstufen in einem Diagramm gibt. Somit kann in einer rausgezoomten Ansicht von zu spezifischen Details abstrahiert werden, sodass dem Nutzer ein grober Überblick gegeben werden kann.

Lösungsansätze

Mit Hilfe unserer Software lassen sich Zoom-abhängige Visualisierungen nur umständlich umsetzen. Pro Sichtausschnitt müsste ein eigener Datenflussstrang erstellt werden. Dieser benötigt alle Daten und erstellt dann mittels des *MorphCreators* visuelle Elemente für die aktuelle Zoomstufe. Für ein kontinuierliches Zoomverhalten existiert nur die Unterstützung des Browsers für den Nutzer. Allerdings ist damit kein semantischer Zoom möglich.

5.4.3 Einstellung von Eingabeelementen

Beim Verwenden von vorgefertigten Eingabeelementen für das *InteractionPanel* (siehe Abschnitt 5.2.1) existieren einige Schwierigkeiten. Diese sind auch schon beim Beispiel zu erkennen, in dem über den Wertebereich mit Hilfe eines Schiebereglers iteriert wurde (siehe Abschnitt 5.2.1). Bei diesem mussten der Wertebereich und die Schrittgröße des Schiebereglers über das manuelle Ausführen von Quelltext eingestellt werden. Für verschiedene Eingabeelemente sind dementsprechend noch Einstellungen zu treffen. Zu diesen Parametern gehören beim Schieberegler unter anderem Minimum, Maximum und Schrittgröße.

Lösungsansätze

Eine Möglichkeit wäre, diese Parameter beim *InteractionPanel* pro Eingabeelement hinzuzufügen. Das heißt, es würden pro Element neben dem Namen und dem Wert der Variablen, sowie dem eigentlichen Eingabewidget noch Einstellungsparameter vorhanden sein. Ein Problem dabei wäre, dass dann durch die vielen Felder das *InteractionPanel* schnell unübersichtlich wird. Außerdem werden die Parameter häufig nur beim Erstellen einmal angepasst.

Eine weitere Alternative wäre ein Popup-Fenster, welches nach dem Hinzufügen des Morphs in das *InteractionPanel* erscheint. Dort könnten dann die Einstellungs-elemente festgelegt werden. Veränderungen im Nachhinein könnten durch ein Kontextmenü, welches auf den Widgets aufgerufen werden kann, ermöglicht werden.

5.5 Zusammenfassung

Interaktivität hilft Visualisierungen interessanter zu gestalten. Der Nutzer kann in interaktiven Diagrammen die Daten untersuchen und Zusammenhänge verstehen. *Flower* ermöglicht es dem Programmierer sowohl visuelle als auch datenabhängige Interaktionen schnell in seine Visualisierungen zu integrieren, ohne ihm Freiheiten zu nehmen.

Durch das *InteractionPanel* können in wenigen Schritten und ohne Quelltext zu schreiben, Interaktionsvariablen erstellt werden. Diese können im Datenfluss wie eine normale Variable ohne Weiteres verwendet werden. Ändert sich der Wert der Variablen, wird das Programm automatisch neu evaluiert. Dadurch, dass Standardeingabeelemente zur Verfügung stehen sowie selbst erstellte Morphs genutzt werden können, lassen sich Interaktionswidgets leicht in die Visualisierung integrieren.

Die Erweiterung des *MorphCreators* erleichtert die Nutzung der schon aus anderen Visualisierungsbibliotheken bekannten Events. Weiterhin werden für häufig verwendete Interaktionen (wie beispielsweise Hovern oder Tooltips) Methoden zur Verfügung gestellt.

Literaturverzeichnis

- [1] *Github Punchcard Diagramm*. URL: <https://help.github.com/articles/using-graphs#punchcard> (besucht am 2014-06-05).
- [2] *bp2013h2 Branch im LivelyKernel Repository auf github.com*. URL: <https://github.com/LivelyKernel/LivelyKernel/tree/38846c5786c8910f2b89620780724cc27895f272> (besucht am 2014-06-05).
- [3] *Dataflow programming*. URL: <http://c2.com/cgi/wiki?DataflowProgramming> (besucht am 2014-06-23).
- [4] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari und T. Mikkonen. „The Lively Kernel: A Self-Supporting System on a Web Page“. In: *Proceedings of the Workshop on Self-Sustaining Systems (S3) 2008, Potsdam, Germany*. Springer, Mai 2008, Seiten 31–50.
- [5] J. Lincke, R. Krahn, D. Ingalls, M. Röder und R. Hirschfeld. „The Lively PartsBin: A Cloud-based Repository for Collaborative Development of Active Web Content“. In: *Proceedings of Collaboration Systems and Technology Track at the Hawaii International Conference on System Sciences (HICSS) 2012, Hawaii, USA*. IEEE Computer, Jan. 2012.
- [6] D. Fichter. *Library Mashups, Chapter 1, What is a mashup?* 2001. URL: <http://books.infoday.com/books/Engard/Engard-Sample-Chapter.pdf> (besucht am 2014-06-20).
- [7] J. Lincke, R. Krahn, D. Ingalls und R. Hirschfeld. „Lively Fabrik - A Web-based End-user Programming Environment“. In: *Proceedings of the Conference on Creating, Connecting and Collaborating through Computing (C5), Tokyo, Japan*. IEEE, Jan. 2009, Seiten 11–19.
- [8] M. S. Puckette. „Pure Data: Another Integrated Computer Music Environment“. In: *Proceedings, Second Intercollege Computer Music Concerts, Tachikawa*. Mai 1997, Seiten 37–41.
- [9] *ECMA-404 The JSON Data Interchange Standard*. URL: <http://www.json.org/> (besucht am 2014-06-01).
- [10] *MSDN xls documentation*. URL: [http://msdn.microsoft.com/en-us/library/gg615597\(v=office.14\).aspx#UnderstandMS_XLS_Overview](http://msdn.microsoft.com/en-us/library/gg615597(v=office.14).aspx#UnderstandMS_XLS_Overview) (besucht am 2014-05-31).
- [11] *Mr. Data Converter*. URL: <https://github.com/shancarter/Mr-Data-Converter> (besucht am 2014-05-28).
- [12] *The R Project for Statistcal Computing*. URL: <http://www.r-project.org/index.html> (besucht am 2014-06-20).

- [13] RFC4180 - Common Format and MIME Type for Comma-Separated Values (CSV) Files. Okt. 2005. URL: <https://tools.ietf.org/html/rfc4180> (besucht am 2014-05-26).
- [14] Extensible Markup Language (XML) 1.0 (Fifth Edition). Nov. 2008. URL: <http://www.w3.org/TR/REC-xml/#sec-logical-struct> (besucht am 2014-05-31).
- [15] W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. Apr. 2012. URL: <http://www.w3.org/TR/xmlschema11-1/> (besucht am 2014-05-31).
- [16] W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. Apr. 2012. URL: <http://www.w3.org/TR/xmlschema11-2/> (besucht am 2014-05-31).
- [17] P. von der Lippe. *Deskriptive Statistik: Formeln, Aufgaben, Klausurtraining*. München: Oldenbourg, 2006. URL: <http://hdl.handle.net/10419/41601> (besucht am 2014-06-03).
- [18] Wikibooks. *Mathematik: Statistik: Einführung – Wikibooks, Die freie Bibliothek*. 2011. URL: http://de.wikibooks.org/w/index.php?title=Mathematik:_Statistik:_Einf%C3%BChrung&oldid=599996 (besucht am 2014-07-09).
- [19] J. W. Tukey. *Exploratory data analysis*. Reading, Mass.: Pearson, 1977. ISBN: 9780201076165.
- [20] *Gapminder World*. URL: <http://www.gapminder.org/> (besucht am 2014-06-02).
- [21] *Visualizations*. URL: <https://web.archive.org/web/20140619014451/http://visualizing.org/galleries> (besucht am 2014-06-29).
- [22] *Der Twitter-Monitor zur Wahl*. URL: <http://www.zeit.de/politik/deutschland/2013-09/twitter-monitor> (besucht am 2014-06-02).
- [23] *Wer wählt was?* URL: <https://web.archive.org/web/20140328231643/http://www.bpb.de/fsd/werwaehltwas/> (besucht am 2014-06-19).
- [24] *HTML input types*. URL: http://www.w3schools.com/tags/att_input_type.asp (besucht am 2014-06-03).
- [25] *Lively introduction*. URL: <http://lively-web.org/users/robertkrahn/lively-cheat-sheet.html> (besucht am 2014-06-19).
- [26] *Observer Pattern*. URL: <http://msdn.microsoft.com/en-us/library/ee817669.aspx> (besucht am 2014-06-04).
- [27] *Document Object Mode*. URL: <http://www.w3.org/DOM/> (besucht am 2014-06-19).
- [28] *D3.js - Data Driven Documents*. URL: <http://d3js.org/> (besucht am 2014-06-19).
- [29] B. Victor. *Drawing Dynamic Visualizations Notes*. URL: <http://worrydream.com/#!/DrawingDynamicVisualizationsTalkAddendum> (besucht am 2014-06-11).
- [30] B. Victor. *Drawing Dynamic Visualizations Talk*. URL: <http://vimeo.com/66085662> (besucht am 2014-06-11).
- [31] G. M. Buurman. *Total Interaction: Theory and practice of a new paradigm for the design disciplines*. Herausgegeben von W. de Gruyter. Birkhäuser, 2005.
- [32] S. Murray. *Interactive Data Visualization for the Web. An Introduction to Designing with D3*. O'Reilly Media, 2013.

- [33] E. Zudilova-Seinstra, T. Adriaansen und R. van Liere. *Trends in Interactive Visualization: State-of-the-Art Survey*. Advanced Information and Knowledge Processing. Springer, 2008. ISBN: 9781848002685.
- [34] J. Nielsen. *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 0125184050.
- [35] *Gapminder Data*. URL: <http://www.gapminder.org/data/> (besucht am 2014-06-24).
- [36] *Working with objects*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#Defining_getters_and_setters (besucht am 2014-06-26).
- [37] *acorn.js*. URL: <http://marijnhaverbeke.nl/acorn/> (besucht am 2014-06-26).
- [38] C. Culy. *Textual Data; Visual Variables*. 2011. URL: http://www.sfs.uni-tuebingen.de/~cculy/courses/W2011/vis/LInfoVis_02_visVariables.ppt (besucht am 2014-06-18).
- [39] J. Bertin. *Semiology of Graphics*. Paris: Esri Press, 1967.
- [40] J. Maloney. „An introduction to morphic: The squeak user interface framework“. In: M. J. Guzdial und K. M. Rose. *Squeak: Open Personal Computing and Multimedia*. Prentice Hall, 2001. ISBN: 9780130280916.
- [41] ECMA. *ECMA-262: ECMAScript Language Specification*. Third. Geneva, Switzerland: ECMA (European Association for Standardizing Information und Communication Systems), Dez. 1999. URL: <http://www.ecma-international.org/ecma-262/5.1/> (besucht am 2014-06-24).
- [42] E. Gamma, R. Helm, R. Johnson und J. Vlissides. *Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley, 1994.
- [43] A. Satyanarayan und J. Heer. „Lyra: An Interactive Visualization Design Environment“. In: *Computer Graphics Forum*. Band 33. 3. Wiley Online Library. 2014, Seiten 351–360.
- [44] B. Victor. „Drawing Dynamic Visualizations“. Stanford HCI seminar. 2013. URL: <http://vimeo.com/66085662> (besucht am 2014-06-20).

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
100	978-3-86956-345-9	Proceedings of the 9th Ph.D. retreat of the HPI Research School on service-oriented systems engineering	Christoph Meinel, Hasso Plattner, Jürgen Döllner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch, Tobias Friedrich (Hrsg.)
99	978-3-86956-339-8	Efficient and scalable graph view maintenance for deductive graph databases based on generalized discrimination networks	Thomas Beyhl, Holger Giese
98	978-3-86956-333-6	Inductive invariant checking with partial negative application conditions	Johannes Dyck, Holger Giese
97	978-3-86956-334-3	Parts without a whole?: The current state of Design Thinking practice in organizations	Jan Schmiedgen, Holger Rhinow, Eva Köppen, Christoph Meinel
96	978-3-86956-324-4	Modeling collaborations in self-adaptive systems of systems : terms, characteristics, requirements and scenarios	Sebastian Wätzoldt, Holger Giese
95	978-3-86956-324-4	Proceedings of the 8th Ph.D. retreat of the HPI research school on service-oriented systems engineering	Christoph Meinel, Hasso Plattner, Jürgen Döllner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch
94	978-3-86956-319-0	Proceedings of the Second HPI Cloud Symposium "Operating the Cloud" 2014	Sascha Bosse, Esam Mohamed, Frank Feinbube, Hendrik Müller (Hrsg.)
93	978-3-86956-318-3	ecoControl: Entwurf und Implementierung einer Software zur Optimierung heterogener Energiesysteme in Mehrfamilienhäusern	Eva-Maria Herbst, Fabian Maschler, Fabio Niephaus, Max Reimann, Julia Steier, Tim Felgentreff, Jens Lincke, Marcel Taeumel, Carsten Witt, Robert Hirschfeld
92	978-3-86956-317-6	Development of AUTOSAR standard documents at Carmeq GmbH	Regina Hebig, Holger Giese, Kimon Batoulis, Philipp Langer, Armin Zamani Farahani, Gary Yao, Mychajlo Wolowyk
91	978-3-86956-303-9	Weak conformance between process models and synchronized object life cycles	Andreas Meyer, Mathias Weske
90	978-3-86956-296-4	Embedded Operating System Projects	Uwe Hentschel, Daniel Richter, Andreas Polze

ISBN 978-3-86956-346-6
ISSN 1613-5652