

XML-Entscheidungsbäume für Applikationen

Jan Oevermann (46594)

Projektarbeit an der
Hochschule Karlsruhe
WS 2013/14

Inhaltsverzeichnis

Einführung	5
Abstract	5
Begriffe	6
UseCases	7
Entscheidungsbäume	8
Darstellung	8
Anwendungen	8
Herausforderungen	9
Trennung von Logik und Inhalt	9
Freigabeprozess	9
Wiederverwendung	9
Workflow	10
Arbeitspakete	10
Freigaben	10
Werkzeuge	11
Beispiele	12
Anforderungen	14
Schlanker Aufbau	14
Übersetzung, Versionierung, Austausch	14
Anbindung an CMS	14
Grafischer Editor	14
Testumgebung	14
XML-Entscheidungslogiken	15
Allgemein	15
GraphML	15
RDF	16
BPMN	16
Fazit	17
Das Format tr3	18
Spezifikation	18
Schema	19
Umwandlung	20
Übersicht tr3-Formate	20
Import von TGF	21
Import von GraphML	22
Export nach RDF	23
Preprocessing	24
Erfassung	25
Allgemein	25
Grafischer Editor yED	25

Qualitätskontrolle	27
Notwendigkeit	27
Erzeugen einer tr3x-Testumgebung	27
Technische Grundlagen	29
Funktionen der Testumgebung	29
API	32
Alternative Testumgebung	34
Veröffentlichung	37
Automatisierte Workflows	37
Einsatz in Anwendung	38
Fazit	40
Neues Format	40
Passendes Framework	40
Neue Erfahrungen	40
Ausblick	41
Offener Test	41
Tracking	41
Referenzierungsarten	41
Anhang	42
Ordnerstruktur	42
Systemanforderungen	42
Kurzeinführung tr3	43
Quellen	44

Einführung

Abstract

Sowohl im Customer-Service-Bereich als auch in der Technischen Dokumentation ist ein allmählich ansteigender Trend hin zur geführten Fehlersuche bzw. zu geführter Hilfe zu beobachten. Als Grundlage dafür dienen sogenannte Entscheidungsbäume, die auf Basis von nacheinander zu treffenden Einzelentscheidungen zu einer Gesamtdiagnose führen und die damit eine komplexe Fragestellung in mehrere simple Einzelfragen herunterbrechen können. Um diesen Vorteil von Entscheidungsbäumen nutzen zu können, bieten sich interaktive (Web-)Anwendungen an, die sowohl den eigentlichen Baum als auch den damit verknüpften Content¹ verarbeiten können.

Da der Einsatz von Entscheidungsbäumen in der Technischen Dokumentation bisher noch relativ neu ist, konnten sich auf Grund fehlender Erfahrungswerte noch keine Prozesse oder Standards etablieren. Gerade im Bereich der Speicherung von Entscheidungsbäumen in einem austauschbaren Format – das idealerweise auf dem XML-Standard basiert – fehlt es bisher an Lösungen. Auch die Integration in bestehende Redaktionsprozesse (Erstellung, Freigabe, etc.) und Systemumgebungen (Anbindung an CMS, TMS etc.) stellt sich als Herausforderung dar.

Diese Arbeit beschäftigt sich sowohl mit Anforderungen an Prozesse und Formate als auch mit einer konkreten Umsetzung und Spezifikation eines Formates für anwendungsorientierte XML-Entscheidungsbäume.

¹ Unter Content versteht man alle Inhalte textueller und visueller Art, die in einem CMS in Form von Informationsobjekten verwaltet werden (vgl. Drewer/Ziegler 2011:297).

Begriffe

Entscheidungsbäume

Laut Definition sind Entscheidungsbäume geordnete, gerichtete Bäume, die der Darstellung von Entscheidungsregeln dienen. Bäume sind im Allgemeinen als Untermenge von Graphen zu betrachten. Sie haben die Charakteristiken von zusammenhängenden, azyklischen² Graphen. Entscheidungsbäume haben den Vorteil, leicht nachvollziehbar und gut erklärbar zu sein, was sie für die Verarbeitung von komplexen Sachverhalten auszeichnet. In der Informatik werden sie oft als besonders schnelle Datenstruktur eingesetzt, die das Finden hierarchisch klassifizierter Objekte innerhalb einer größeren Menge vereinfacht.

XML

Als verbreiteter und akzeptierter Standard sowohl im Bereich der Technischen Dokumentation als auch als Austausch und Quellformat in Web-Applikationen liegt der Einsatz von XML (Extensible Markup Language) für die Speicherung von Entscheidungsbäumen auf der Hand. Besonders der Umstand, das XML selbst auf einer verschachtelten (Baum-)Struktur beruht ist von Vorteil für eine effiziente Informationsspeicherung. Weitere Einsatzgründe, die für das Format sprechen, ist die gute Integration in bestehende Systemumgebung und die in Redaktionen bereits bekannten Regeln bezüglich Struktur und Wohlgeformtheit.

Applikationen

Als Applikation wird in dieser Arbeit eine webbasierte Endbenutzeranwendung bezeichnet. Grundsätzlich treffen die getroffenen Aussagen aber auf jede Art von Anwendung zu. Ziel ist die direkte Integration eines Baumes im XML-Format in die jeweilige Anwendung (unterstützt durch eine Art Laufzeitumgebung oder einen Interpreter des Formats).

² Diese Aussage trifft auf (Entscheidungs-)Bäume im Allgemeinen zu, jedoch nicht mehr auf das in dieser Arbeit entworfene tr3-Format (da dort die Bildung von Zyklen bzw. Schleifen möglich ist).

UseCases

Grundsätzlich können Entscheidungsbäume in einer Vielzahl von Anwendungsfällen verwendet werden. Als Standardwerkzeug gelten sie unter Anderem im Formulieren von Regelwerken für zukünftige Entscheidungen auf Basis von Erfahrungswissen aber auch in der Klassifizierung von Objekten (woraus sich z.B. taxonomische Metadaten ergeben können). Bekannte Beispiele sind die Fehlerdiagnose in Fahrzeugen oder softwaregestützte Rechtsschreibprüfungen (hier kommen sog. *Tries* oder *Prefix Trees* zum Einsatz).

Der *Use Case*, der in dieser Arbeit betrachtet werden soll, ist ein relativ neuer Anwendungsfall von Entscheidungsbäumen in der Technischen Dokumentation (TD): die geführte Fehlersuche oder auch *Customer Self Service*. Sie soll dem Nutzer die Möglichkeit zur Selbsthilfe bieten und damit den Hersteller-Support entlasten und ggf. die Reklamation sog. *false positives* (also fälschlich als defekt deklarierte Geräte) verringern. Gleichzeitig soll die Nutzerzufriedenheit durch ein schnelles Hilfeverfahren gesteigert werden. Bisherige Ansätze in der TD stützen sich auf Troubleshooting-Tabellen und Schrittanleitungen. Doch gerade bei komplexeren Produkten wie Unterhaltungselektronik oder Fahrzeugen reichen diese Ansätze oft nicht mehr aus.

Zwei konkrete Beispiele aus der Praxis:

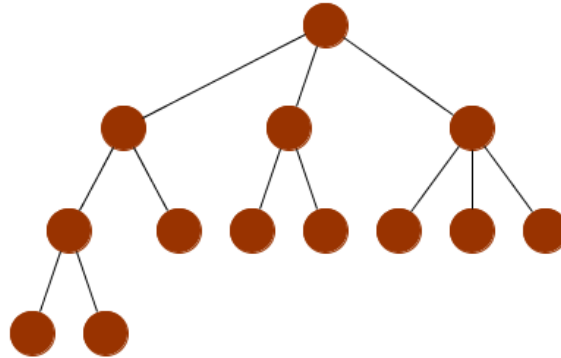
- Der Automobilhersteller Daimler bietet in der Betriebsanleitung für sein Modell smart fortwo das Kapitel *Selbsthilfe* an, in welchem die Nutzer Displaymeldungen anhand einer Tabelle selbst interpretieren können sollen. Diese Darstellungsform ist dann ausreichend und gut gewählt, wenn pro Displaymeldung genau eine Diagnose in Frage kommt. In der Realität ist die tatsächliche Diagnose oder Lösung aber von mehreren Faktoren (Zeitpunkt des Auftretens, Anzeigefrequenz, vorherige Standzeit, etc.) abhängig. Dadurch wird das Stellen der Diagnose und damit das Auffinden des Fehlers sehr komplex. Entscheidungsbäume sind hier die geeigneteren Mittel, z.B. in Form einer Smartphone-App.
- Der Mobilfunkanbieter Vodafone ist als Vertreiber verschiedener Endgeräte (Mobiltelefone, Tablets, Router, etc.) auch für deren Reparatur verantwortlich und möchte die Zahl an Rücksendungen von falsch positiv defekten Geräten verringern. Dies soll durch eine dem Online-Einsendeformular vorgeschaltete Diagoseseite realisiert werden. Durch die steigende Anzahl an Modellen und Funktionen, ist eine Troubleshooting-Tabelle oder eine Schrittliste allerdings nicht geeignet, um die benötigte Übersicht und Eingrenzung zu vermitteln. Abhilfe schafft eine Entscheidungsbaum-basierte Webapplikation.

In beiden Fällen können Entscheidungsbäume Abhilfe und Klarheit schaffen, in dem Sie durch abgrenzende Fragestellung, die Diagnose des Problems einschränken und dem Nutzer eine individuelle Rückmeldung geben.

Entscheidungsbäume

Darstellung

Entscheidungsbäume können in verschiedenen Formen dargestellt werden. In seiner natürlichsten Form wird der Baum in seiner Gesamtheit grafisch abgebildet:



Bei komplexen Bäumen, die viel Information beeinhalten wird diese Darstellungsform allerdings schnell unübersichtlich und verfehlt den Zweck einer einfachen Diagnose.

Eine durch interaktive Medien möglich gewordene Form der Darstellung ist das Ablaufen eines Pfades von der Wurzel bis zu einem Endknoten (Blatt). Dem Nutzer wird hier bei jeder Entscheidung (also einer Verzweigung des Baumes) genau eine Frage gestellt, auf die er eine eingeschränkte Anzahl an Antworten (die Kindknoten des aktuellen Knotens) geben kann. Dadurch wird der individuelle Weg bis zu einem Endknoten (und damit zur Diagnose) gebildet, der sich von Nutzer zu Nutzer unterscheiden kann.

Auch für den Einsatz des Entscheidungsbaums in einer Endnutzeranwendung ist diese Darstellungsform zu empfehlen, da sie die Komplexität des Problems für den Nutzer auf eine Reihe wenig komplexer Fragen herunterbrechen kann ohne die Gesamtkomplexität des Baums preiszugeben.

Anwendungen

In der oben genannten Darstellungsform können Entscheidungsbäume zur geführten Fehlersuche in verschiedene Endanwendungen integriert werden. Die vorliegende Arbeit beschäftigt sich mit der Implementierung in webbasierte Anwendungen, die von einer Einbindung in den Support-Teil einer Website bis hin zur Grundlage einer Smartphone-App reichen können. Diese Anwendungen haben zum Ziel, dem Nutzer mit möglichst wenigen Schritten in seinem Anliegen helfen zu können. Neben der eigentlichen Logik (Frage -> Antwort -> Frage) besteht in der Praxis auch die Anforderung zusätzliche Inhalte (formatierter Text, Bilder, weitere Anwendungen) einblenden zu können. Diese Inhalte sind zum Teil schon als Module in einem CMS vorhanden (z.B. aus der Printversion einer Anleitung). Diese Module sollen gemäß dem Single-Source-Prinzip auch weiterhin verwendet werden und mit einzelnen Knoten des Baums verknüpft werden können.

Technische Grundlage dieser Anwendungen ist der Entscheidungsbaum in seiner XML-Repräsentation und den damit verknüpften Inhalten als HTML-Dateien. In dieser Arbeit wird davon ausgegangen, dass die Verwaltung des Inhalts modulbasiert in einem Content-Management-System (CMS) erfolgt und dieser z.B. über Dateinamen oder IDs mit dem eigentlichen Entscheidungsbaum verknüpft werden kann.

Herausforderungen

Trennung von Logik und Inhalt

Neben der zwingend modularen Denkweise müssen die beteiligten Redakteure auf die saubere Trennung zwischen Logik und Inhalt achten. Diese Trennung ist nötig, um die Wiederverwendung von Inhaltsmodulen zu ermöglichen und unabhängig von der Baumstruktur zu gewährleisten. Eine Strukturänderung des Baumes darf wenig bis keinen Änderungsbedarf am Inhalt verursachen, da sonst Aufwand und Koordinationskomplexität exponentiell ansteigen. Dies stellt neue Anforderungen an den Inhalt, der nun nicht nur inhaltlich abgeschlossen, sondern auch unabhängig von der umgebenen Struktur sein muss.

Auch die Skalierbarkeit des Gesamtprozesses hängt von dieser Anforderung ab. Durch eine strikte Trennung bleibt eine hohe Flexibilität auf Content-Seite erhalten, was dazu führt dass neue Inhalte, Inhaltsvarianten oder Spezifika³ nicht zwingend zu einer Änderung am Baum führen. Damit bleibt der personelle Aufwand beim Administrieren des Baumes gering während die rein textliche Content-Erstellung in Technischen Redaktionen in der Regel problemlos skalierbar ist.

Freigabeprozess

Auch der redaktionelle Freigabeprozess muss neu überdacht werden. Gerade bei sehr großen Bäumen, die mehrere 100 oder gar 1000 Content-Module verknüpfen, muss ein durchdachter Prozess entwickelt werden, der unter anderem die Freigabenstatus zwischen Baum und Content regelt. Ändert sich der Freigabestatus eines Content-Moduls, ändert sich dann auch der Freigabestatus des gesamten Baums? Ändert sich die Struktur des Baums, ändert sich dann der Freigabestatus des Contents?⁴ Die Granularität mit welcher man Status zuweist kann von einzelnen Knoten bis zu Teilbäumen reichen, die den Status kaskadierend vererben. Eine Entscheidung kann hier nur anhand konkreter Anwendungsszenarien getroffen werden.

Wiederverwendung

Eine weitere Herausforderung stellt sich, wenn das streng hierarchische Konzept eines Entscheidungsbaums um die in der TD üblichen Methoden Wiederverwendung (von Teilbäumen) und Referenzierung (im Web-Umfeld: Verlinkung) erweitert werden soll. Sie sind für einen effizienten Einsatz der Inhalte unerlässlich.

³ Im Vodafone-UseCase z.B. die gerätespezifischen Inhalts-Varianten.

⁴ Bei einer klaren Trennung zwischen Inhalt und Logik ist dies nicht der Fall, sollte aber aus QS-Gründen trotzdem überprüft werden

Workflow

Arbeitspakete

Werden sowohl Inhalte als auch Entscheidungsbäume innerhalb der gleichen Redaktion erstellt, gliedern sich die anfallenden Arbeitspakete über den gesamten *Lifecycle* eines Baumes in den meisten Fällen wie folgt:

- Konzeption des Baums
Entwurf, Konzept, grafische Darstellung der geplanten Knoten und Freigabe
- Umsetzung des Baums
Technische Umsetzung oder Übertragen in Zielsystem bzw. -format
- Erstellen des Contents
Redaktionelles Erstellen von textuellen und grafischen Inhalten und Freigabe
- Verknüpfen des Contents
Technische Verknüpfung von Baumknoten zu Inhaltsmodulen
- Testen des fertigen Baums
Qualitätssicherung, Fehlersuche, Analyse des Baums
- Freigabe + Veröffentlichung (Deployment)
*Abnahme des Baums **mit** Content, Überführen in Zielformat*

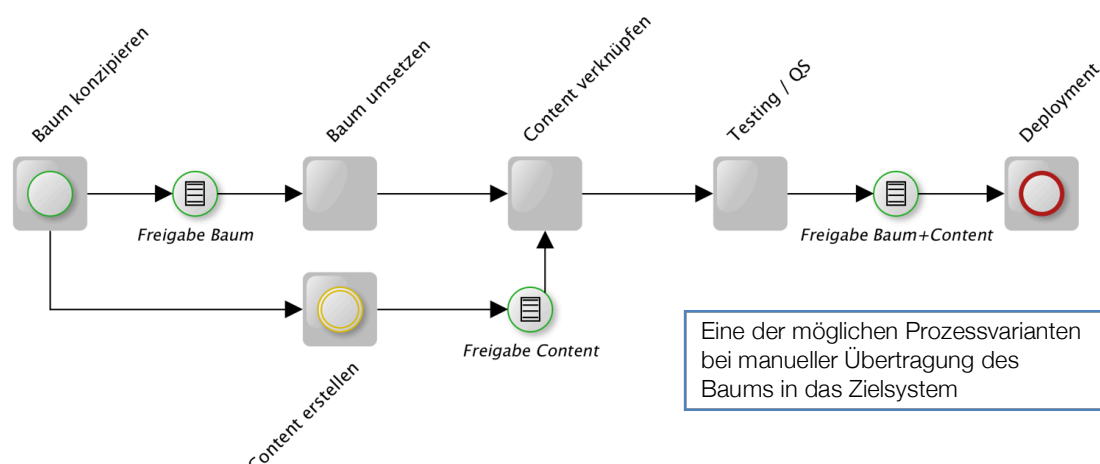
Die Reihenfolge und Verteilung der Arbeitspakete kann bedingt variieren. So fallen im Idealfall Konzeption und Umsetzung des Baumes zusammen oder die Umsetzung wird weitestgehend automatisiert. Ebenso kann die Content-Erstellung versetzt zu den Arbeiten am Baum begonnen werden. Es wird davon ausgegangen, dass in einer Redaktion bereits Prozesse zur Erstellung von Content etabliert sind. Die Verknüpfung des Contents kann entweder kontinuierlich oder für alle Knoten in einem Schritt vorgenommen werden. Ebenfalls kann je nach konzeptionellem Ansatz die Verknüpfung des Contents vor die Umsetzung des Baums in das Zielsystem erfolgen.

Freigaben

Aus der Aufteilung der Arbeitspakete geht hervor, dass zunächst Baum und Content getrennt voneinander erstellt werden und anschließend durch eine Verknüpfung (über IDs, Dateinamen, Includes, etc.) zusammengeführt werden. Aus Gründen der Effizienz ist die Erstellung durch verschiedene Teams oder Personen in den meisten Fällen auch sinnvoll.

Dadurch ergeben sich aber mehrere Freigabe-Ebenen:

- Freigabe des Baums nach der Konzeption, also vor der Umsetzung in ein System
- Freigabe des Contents nach dem Erstellen, also vor dem Ausspielen aus dem CMS
- Freigabe des Baums mit Content nach dem Testing, also vor dem Publizieren



Je nach technischer Umsetzung ist es auch möglich, auf Freigabepunkte zu verzichten, und z.B. nur den mit Content verknüpften Baum zu untersuchen (3. Freigabepunkt). Dies kann allerdings weitreichende Folgen haben: Bei einem strukturellen Fehler im Baum, der einen mit Content verknüpften Knoten betrifft, muss zu diesem Zeitpunkt der Gesamtprozess noch einmal komplett durchlaufen werden.

Werkzeuge

Als Werkzeuge kommen für die Arbeitspakete verschiedene Tools in Frage:

Konzeption des Baums

Im Idealfall wird ein Baum technisch so nah wie möglich an der späteren Implementierung entworfen um Komptabilitätsprobleme bei der Umsetzung zu vermeiden. Oft ergeben sich dadurch auch Möglichkeiten zur Automatisierung. Die Konzeption eines Baumes kann aber auch unabhängig von der späteren Implementierung erfolgen (also unabhängig von der technischen Umsetzung), der Baum muss dann allerdings auch manuell übertragen werden.

Beispiele für mögliche Werkzeuge:

Analog (*Stift/Papier, Flipchart*), Graphen-Editoren (*yED*), Workflow-Designer (*ARIS, Bizagi*), Text oder XML-Editoren (*Notepad++, Sublime Text, XMetaL*), Eigenentwicklungen

Umsetzung des Baums

Um Fehlerquellen bei der Umsetzung des Baumes zu vermeiden, sollte diese so weit wie möglich automatisiert werden - sie kann auch komplett entfallen, wenn bei der Erstellung bereits das Zielformat erzeugt wird. Auch der zeitliche Aufwand wird z.B. durch eine automatische Konvertierung oder Import deutlich verringert.

Beispiele für mögliche Werkzeuge:

Manuell (*händisches Übertragen in Zielsystem*), Transformation (*XSL*), Interpretation (z.B. *durch Skriptsprachen*), Import in ein Zielsystem (*eGain*)

Erstellen des Contents

In dieser Arbeit wird davon ausgegangen, dass die Erstellung von textuellem Content in einem CMS den dafür vorgesehenen Standards und Vorgehensweisen entspricht. Die Erstellung multimedialer Inhalte kann in den dafür vorgesehenen Programmen erfolgen, die Ergebnisse sollten aber im CMS verwaltet werden. Wo das nicht möglich ist (z.B. bei Zusatzinhalten mit Anwendungscharakter) müssen individuelle Lösungen gefunden werden.

Beispiele für mögliche Werkzeuge:

Text: XML-Editor (*XMetaL, PTC Arbortext*), Grafiken: Bildbearbeitungsprogramme (*Photoshop*), Anwendungen: Entwicklungsumgebungen (*Eclipse, NetBeans*)

Verknüpfen des Contents

Die Verknüpfung des Contents mit den entsprechenden Knoten des Baums kann je nach Ansatz im Entwurf-Werkzeug oder erst in der Umsetzung erfolgen. In beiden Fällen kommen schon bekannte Systeme zum Einsatz. Zum Verknüpfen kann unter Umständen ein Export des Inhalts aus dem CMS nötig sein. In manchen Fällen gibt es auch kombinierte Ansätze. So wird die logische Verknüpfung (z.B. über einen gemeinsamen Schlüssel) bereits in der Entwurfsphase vorgenommen, die tatsächliche technische Zusammenführung erfolgt aber erst in der Umsetzung des Formats.

Testen des fertigen Baums

Um die Funktionalität des Baums und die korrekte Verknüpfung zu CMS-Inhalten zu überprüfen, empfehlen sich Tests zur Qualitätssicherung. Diese erfolgen im besten Fall automatisiert oder programmatisch unterstützt und ist stark von der jeweiligen Umsetzung abhängig.

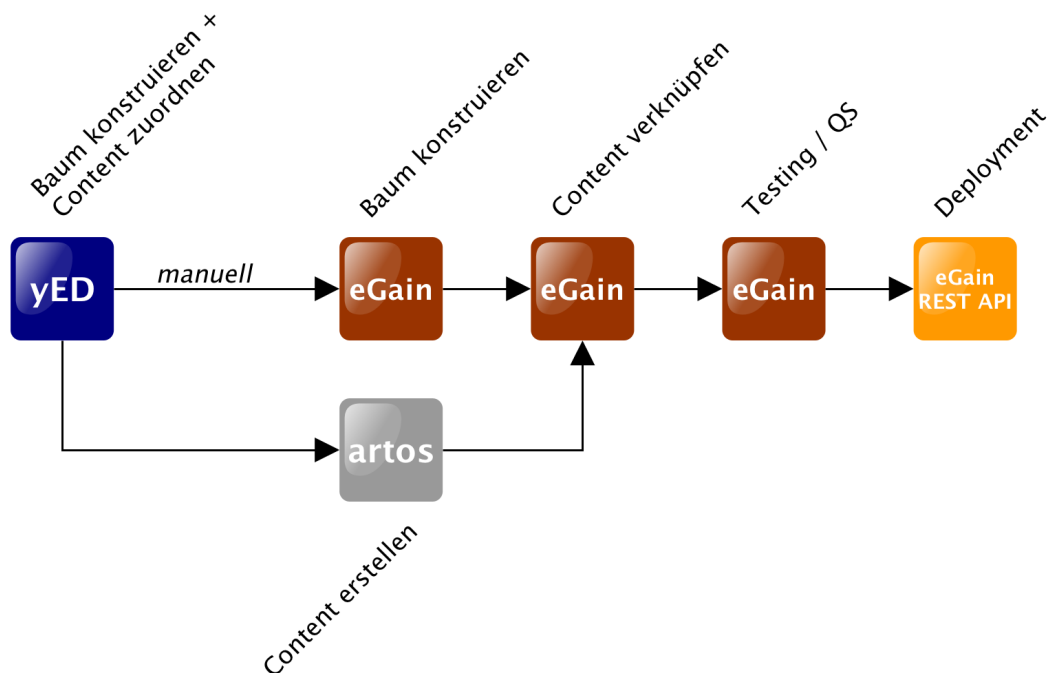
Freigabe + Veröffentlichung (*Deployment*)

Die Freigabe eines Baumes kann je nach Implementierung innerhalb des Baumformats selbst oder überentsprechende Metadaten in z.B. einem CMS umgesetzt werden. Die Veröffentlichung besteht in den meisten Fällen daraus, die entstandene(n) Datei(en) in die Laufzeitumgebung der Applikation zu importieren.

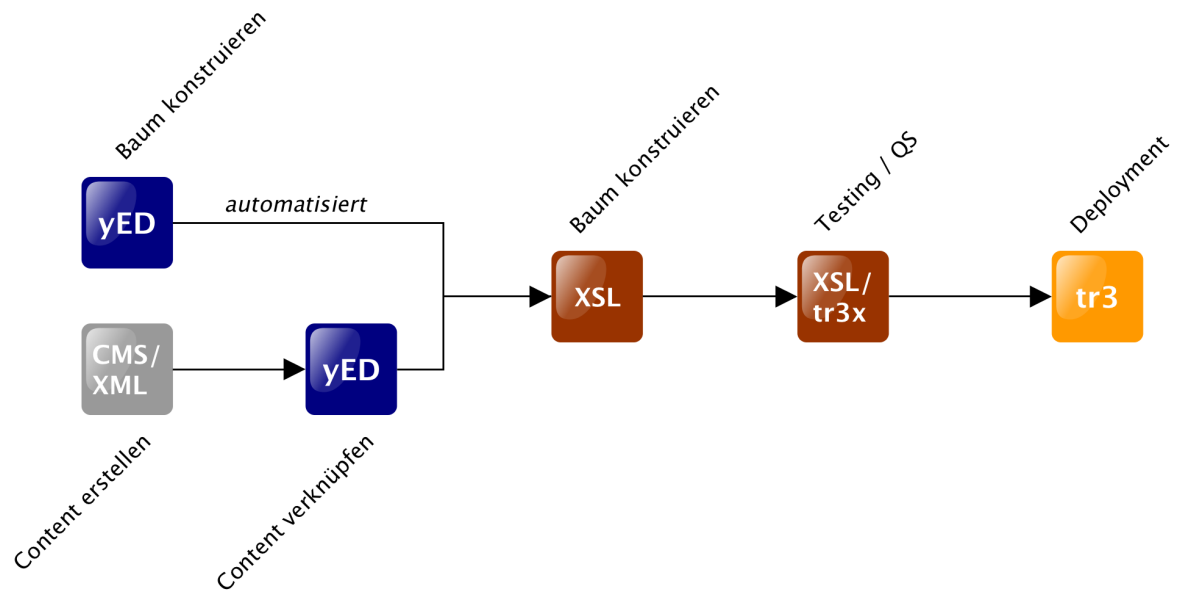
Beispiele

Wie schon in den vorherigen Abschnitten erwähnt wurde, kann die genaue Abfolge und Charakterisierung der Arbeitspakete je nach Umsetzung variieren. Im Folgenden sind die beiden im Abschnitt *UseCases* beschriebenen Anwendungsfälle mit ihren konkreten Implementierungen und Angabe der eingesetzten Werkzeuge abgebildet.

Vodafone Diagnose Service⁵



⁵ Frontend zu erreichen unter: <https://reparaturen.vodafone.de/eservice/>



⁶ Konzeptionelles Hochschulprojekt in Kooperation mit der Daimler AG

Anforderungen

Schlanker Aufbau

Das Format muss so kompakt wie möglich aufgebaut sein, um auch für den Einsatz auf mobilen Geräten geeignet zu sein. Gerade in ländlicheren Gegenden bestehen noch keine Breitband-Mobilfunknetze, so dass bei der Übertragung von Daten jedes Kilobyte an Mehrgröße direkte Auswirkung auf die Ladezeit und damit auf die User Experience haben.

Im Falle eines Entscheidungsbaumes lassen sich Einsparungen an der Dateigröße durch die folgenden Faktoren drastisch reduzieren:

- Strikte Trennung von Logik und Inhalt. Auch wenn das Nachladen⁷ von Zusatzinhalten fehlschlägt muss der Baum in seiner Logik bedienbar bleiben.
- Volle Ausnutzung der XML-Hierarchie. Durch die Ausnutzung der XML-Baumstruktur kann auf eine zusätzliche Beschreibung der Kanten (Knotenbeziehungen) verzichtet werden (wie es bei Formaten für Graphen üblich ist).
- Nur Kernattribute obligatorisch machen. Alle optionalen Attribute bei Null-Werten streichen und sinnvolle Default-Werte spezifizieren (So wird ein ID-Attribut nur bei Verlinkung zwischen Hierarchien benötigt)

Übersetzung, Versionierung, Austausch

Um den hohen Qualitätsstandards in der Technischen Dokumentation zu entsprechen, muss das Format reibungslos durch standardisierte Prozesse von einem Dienstleister zu übersetzen sein. Der Austausch zwischen verschiedenen beteiligten Parteien soll durch einen klar definierten Aufbau keine Komptabilitätsprobleme bereiten. Des Weiteren muss eine Versionierung durch ein CMS oder eine externe Versionsverwaltung (git, svn, etc) möglich sein. Alle oben genannten Probleme können durch den Einsatz eines definierten XML-Schemas behoben werden.

Anbindung an CMS

Um bereits bestehenden Content aus einem CMS als (Zusatz-)Inhalte für den Baum nutzbar zu machen, muss es die Möglichkeit geben, diese einfach einzubinden. Dies kann über eine z.B. über einen eigenen Publikationsweg des CMS sein. Im Speicherformat des Entscheidungsbaums muss es möglich sein, diese publizierten Inhalte zu integrieren (z.B. über eine Dateinamen-Referenzierung)

Grafischer Editor

Der Redakteur, der für das Entwerfen und Konstruieren des Baums verantwortlich ist, soll mit Hilfe eines grafischen Editors unterstützt werden. Dabei soll sein Arbeitsprozess so weit wie möglich vereinfacht werden. Angelehnt an sog. WYSIWYG-Editoren, kann der Redakteur schon vor dem Einsatz in einer Anwendung den schematischen Verlauf des Baumes erkennen und direkt manipulieren.

Testumgebung

Es muss möglich sein, einen Entscheidungsbaum vor seiner Veröffentlichung bzw. der Integration in die Endanwendung auf Fehler untersuchen zu können. Dies ist gerade bei sehr großen Bäumen notwendig. Hier sind die typischen Anforderungen an eine Testumgebung ein Fehler- und Analyse-Reporting, eine baumübergreifende Suche, eine Vorschau der Inhalte und eine umfassende interaktive Visualisierung.

⁷ z.B. beim vom tr3-Format gewählten Ansatz Inhalte via AJAX im Moment des Aufrufs nachzuladen

XML-Entscheidungslogiken

Allgemein

Im Customer-Service-Bereich haben sich bereits einige Anbieter im Bereich der „Guided Help“ platziert und bietet proprietäre Softwarelösungen an, die eine Laufzeitumgebungen für Entscheidungsbäume in webbasierten Kundenportalen anbieten (vgl. eGain SelfService⁸).

Für den Bereich der Technische Dokumentation (TD) sind diese Lösungen allerdings ungeeignet, da sie sich schlecht in kritische Prozesse wie Übersetzung und Freigabe integrieren lassen, keine grafischen Oberflächen bieten und keine ausreichenden Testumgebungen zur Verfügung stellen. Des Weiteren zeigen diese Produkte im Vergleich zu CMS methodische Schwächen im Variantenmanagement und der parametrisierten Inhaltsfilterung.

Da sich im Bereich der TD bereits XML als de-facto-Standard für die systemunabhängige Speicherung von Inhalten durchgesetzt hat, sollen nun ein XML-basiertes Format für Entscheidungsbäume für den redaktionellen Einsatz verwendet werden.

GraphML

Als ein offenes, auf XML basierendes Format für die Repräsentation von Graphen aller Art hat sich GraphML etabliert. Als Nachfolger der Graph Modeling Language (GML), die bereits seit 1995 bestand hatte, fand das Format schnell Verwendung in Graphen-Editoren aller Art. Der verbreitete Editor „yED“ verwendet GraphML als natives Format.

GraphML zeichnet sich durch gute Verständlichkeit bezüglich Struktur und Lesbarkeit, sowie Erweiterungsmöglichkeiten im Bereich der anwendungsspezifischen Datenfelder aus.

Das Format speichert den Graphen als eine Liste von Knoten („nodes“) und Kanten („edges“) mit ihren jeweiligen Eigenschaften („data“). Kanten haben Quell- und Zielattribute, die wiederum auf Knoten verweisen:

```
<graph id="Reise" edgedefault="undirected">
  <node id="KA"/>
  <node id="B"/>
  <edge id="Flug_657" source="KA" target="B"/>
</graph>
```

Beim Einsatz von sog. WYSIWYG-Editoren werden den Grundelementen weitere Elemente anderer Namensräume untergeordnet, die in der Regel Informationen zu Position, Gestaltung und Beschriftung der Elemente enthalten:

```
<node id="n3">
  <data key="d4"><![CDATA[Lorem Ipsum]]></data>
  <data key="d5">
    <y:ShapeNode>
      <y:Geometry height="30.0" width="30.0" x="700" y="346"/>
      <y:Fill color="#FF9900" transparent="false"/>
      <y:NodeLabel>Carpe Diem!</y:NodeLabel>
      <y:Shape type="rectangle"/>
    </y:ShapeNode>
  </data>
</node>
```

⁸ http://www.egain.com/products/web_self-service/

Durch den auf visueller Repräsentation basierenden Ansatz werden gerade bei Baumstrukturen logische Hierarchien mit der Speicherung umgewandelt.⁹ Auch der durch Editoren verursachte sog. Overhead sorgt für einen Zuwachs der Dateigröße um das Mehrfache der in der Endanwendung benötigten Informationen. Durch diese Eigenschaften eignet sich GraphML als Speicherformat in Konzeptions- und Erstellungsphase eines Entscheidungsbaumes, jedoch nicht für den direkten Einsatz in Applikationen.

RDF

Das Resource Description Framework (RDF) wurde vom W3C ursprünglich als Standard zur Beschreibung von Metadaten konzipiert. Das Konzept kann Sachverhalte („Aussagen“) über beliebige Dinge („Ressourcen“) festhalten. Dabei werden die Informationen zu diesen Aussagen in Tripeln (3er-Informationssätze) gespeichert. Diese Tripel bestehen aus Subjekt (das Beschriebene), Prädikat (die Beziehung oder die Art der Aussage) und Objekt (das Ausgesagte). Um eine universelle Eindeutigkeit zu gewährleisten werden Ressourcen in der Regel mit URLs repräsentiert. Durch das Einbinden standardisierter Namensräume, kann auf umfangreiches vordefiniertes Vokabular zum Bilden von Aussagen zurückgegriffen werden.

Die Herangehensweise von RDF ist grundsätzlich unabhängig von einem Format oder einer textuellen Repräsentation findet jedoch häufig als eine XML-Sprache ihre Anwendung:

```
<rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  <rdf:Description rdf:about="http://www.hs-karlsruhe.de/kmm-m">
    <dc:description>Website des Studiengangs KMM Master</dc:description>
  </rdf:Description>
</rdf:RDF>
```

Die Vorteile von RDF sind das universelle Konzept und die Unabhängigkeit einer speziellen Implementierung. Allerdings bringt gerade die Komplexität des XML-Formats für die Anwendung in Applikationen Probleme mit sich. So muss jede mögliche direkte Verbindung, die in einem Baum besteht (z.B. auch Referenzierungen) in einem eigenen Tripel festgehalten werden. Dadurch entsteht schon für kleine Bäume eine hohe Dateigröße. Auch das Fehlen eines intuitiven grafischen Editors zur Erstellung des Formats schließen es für die Verwendung in einer Redaktion aus.

BPMN

Die *Business Process Model and Notation (BPMN)* ist eine aus dem Bereich der Prozessmodellierung stammende (grafische) Spezifikationssprache. Das zu Grunde liegende Modell stellt bestimmte Symbole zur Notation von Geschäftsprozessen zur Verfügung. Anfang 2011 wurde mit der Einführung BPMN 2.0 ein XML-basiertes Format zur Speicherung spezifiziert. Mit diesem Format lassen mit Hilfe von sog. *exclusive gateways*¹⁰ Entscheidungsknoten innerhalb eines Prozessgraphs speichern und somit auch Entscheidungsbäume simulieren. Ein innerer Knoten würde in der XML-Notation als *Gateway* mit einem Eingang (*incoming*) und ein oder mehreren Ausgängen (*outgoing*) wie folgt dargestellt:

```
<semantic:exclusiveGateway gatewayDirection="Unspecified" name="Frage" id="_117">
  <semantic:incoming>_141</semantic:incoming>
  <semantic:outgoing>_194</semantic:outgoing>
  <semantic:outgoing>_247</semantic:outgoing>
</semantic:exclusiveGateway>
```

⁹ Die Baumstruktur wird als gerichteter Graph abgebildet

¹⁰ <http://www.omg.org/spec/BPMN/20100601/10-06-02.pdf>

Die in den Elementen des Gateways stehenden Zeichenketten sind wiederum IDs auf die mit dem Gateway verknüpften Prozesselemente. Durch diese Struktur sind für die Abbildung von Entscheidungsbäumen sehr viele Knoten notwendig (mindestens zwei abzubildende Knoten pro logischem Knoten).

BPMN zeichnet sich durch die Vielzahl an Editoren und Implementierungen sowie durch seinen Funktionsumfang aus. Allerdings bringt das Format sehr viel *Overhead* und Komplexität mit sich, was es für den effizienten Einsatz in einer Redaktion ausscheiden lässt.

Fazit

Alle betrachteten Formate sind in ihrer nativen Form graphenorientiert, d.h. zu wenig restriktiv, was die Struktur des Aufbaus betrifft. Da ein Baum nur eine spezielle Form eines Graphen ist, lassen sich Entscheidungsbäume problemlos in diesen Formaten abbilden, allerdings wird der natürliche hierarchische Aufbau von XML-Dokumenten nicht ausgenutzt.

Die untersuchten Formate GraphML, RDF und BPMN bieten können die Anforderungen aus Erstellungs- und Anwendungsseite nicht durchgängig erfüllen. Sowohl RDF als auch BPMN sind zwar standardisierte und etablierte Formate aber für die Abbildung reiner Entscheidungsbäume zu komplex. GraphML stellt sich auf Erstellungsseite auf Grund der guten Implementierung in Editoren (speziell in *yED*) und seiner guten Verständlichkeit als geeignet dar, ist allerdings für den Einsatz in einer Anwendung auf Grund seiner flachen, graphenorientierten Struktur und der großen erzeugten Datenmenge eher ungeeignet.

Auf Grundlage dieser Betrachtungen wurde die Lösung gewählt, GraphML als Erstellungsformat zu verwenden, welches für den produktiven Einsatz in ein neu erschaffenes Format, *tr3*, transformiert wird. RDF ist als universell austauschbares Format eine Exportmöglichkeit.

Das Format tr3

Spezifikation

XML-Namespace: <http://www.hs-karlsruhe.de/kmm-m/tr3>¹¹

Das entwickelte Format *tr3* speichert als zentrales Format die reine Logik eines Out-Trees¹² in seiner natürlichen hierarchischen Repräsentation. Im Gegensatz zu graphenorientierten Formaten existieren nur Knoten (Kanten existieren implizit), die sich in drei strukturelle und drei funktionale Typen gliedern:

Strukturell (wie bei jedem Baum):

- Wurzel (XML-Element: *tr3*) `<tr3/>`
- Ast (XML-Element: *branch*) `<branch/>`
- Blatt (XML-Element: *leaf*) `<leaf/>`

Funktional:

- Normal (Standard-Strukturknoten) `<tr3/> <branch/> <leaf/>`
- Kopie (XML-Element *ref* mit Attribut *copy*) `<ref copy="id" />`
- Link (XML-Element *ref* mit Attribut *link*) `<ref link="id" />`

Jeder Knoten besitzt ein mindestens ein *out*-Attribut, das den Ausgabewert bei Erreichen des Knotens angibt. Dies ist in der Regel eine Frage oder Anweisung, bei Blatt-Knoten eine Diagnose. Alle *branch*- und *leaf*-Knoten besitzen eine Eingangsbedingung, die im *in*-Attribut erfasst wird. Dies stellt in der Regel die Antwort dar, die der Benutzer geben muss um zu dem Knoten zu gelangen, der das Attribut mit dem gewählten Wert besitzt:

```
<branch out="Kaffee oder Tee?" ...>
  <leaf in="Kaffee" out="Du bekommst einen Kaffee" />
  <leaf in="Tee" out="Du bekommst einen Tee" />
</branch>
```

Durch den Einsatz von Eingangsbedingungen ergeben sich diverse Vorteile:

- Keine separate Speicherung von Kanten (Verknüpfungen von Knoten), dadurch geringere Dateigröße
- Bei Wiederverwendung oder Referenzierung eines Knotens kann eine andere Eingangsbedingung gewählt werden (siehe Beispiel *Referenzierung*)
- Durch den Wegfall separat definierter Kanten muss nicht mehr jeder Knoten eine ID besitzen und kann einfach über seine hierarchische Position und seine Eingangsbedingung selektiert werden

Sollen Knoten oder Teilbäume wiederverwendet oder verlinkt werden müssen sie ein *id*-Attribut besitzen, das frei wählbar ist, solange es innerhalb des Baumes eindeutig ist. Die eigentliche Referenzierung erfolgt über das *ref*-Element, das an die Stelle tritt, an die der referenzierte Knoten oder Teilbaum treten soll. Dabei kann eine andere Eingangsbedingung gewählt werden:

¹¹ 3 funktionale und strukturelle Knotentypen / tr = Technische Redaktion / tr3 ist ausgesprochen: tre(e)

¹² <http://de.wikipedia.org/wiki/Out-Tree> - Mit Ausnahme von zyklischen Referenzierungen

```
<branch out="Kaffee oder Tee?">
  <leaf in="Kaffee" out="Du bekommst ein Heißgetränk" id="heiß" />
  <ref in="Tee" link="heiß" />
</branch>
```

Im gezeigten Beispiel ist es egal, ob der Benutzer „Kaffee“ oder „Tee“ wählt. Es wird immer die Ausgabe des Knotens mit der id „heiß“ ausgegeben.¹³ Auch Teilbäume können referenziert (in diesem Fall kopiert) werden:

```
<branch out="Kaffee oder Tee?">
  <branch in="Kaffee" out="Mit Zucker?" id="zucker">
    <leaf in="Ja" out="Du bekommst ein Heißgetränk mit Zucker" />
    <leaf in="Nein" out="Du bekommst ein Heißgetränk ohne Zucker" />
  </branch>
  <ref in="Tee" copy="zucker" />
</branch>
```

Soll externer Content (in etwas aus einem CMS), der zusätzlich zum out-Attribut ausgegeben werden, so kann über das *content*-Attribut eine HTML-Datei mit dem entsprechenden Knoten verknüpft werden.

```
<leaf in="Hilfe" out="Folgen Sie der Anweisung" content="anweisung.html" />
```

Des Weiteren können Knoten beliebige Klassen zugewiesen werden, die sie in ihrer Art klassifizieren. Diese Klassen haben keine Auswirkung auf die Verarbeitung des Baumes, können aber für die Ausgabe von Bedeutung sein. So ist es z.B. sinnvoll, Diagnosen zu bewerten oder Spezialcontent auszuzeichnen.

```
<leaf in="Feuer" out="Es besteht Explosionsgefahr" class="danger" />
```

Um den gesamten Baum zu benennen existiert zusätzlich das *title*-Attribut ausschließlich am Element *tr3*. Dieses Element besitzt kein *in*-Attribut.

Weitere Attribute werden durch Transformationen automatisch vergeben: das *origin*-Attribut als Herkunftsnachweis beim Auflösen von kopierten Teilbäumen oder zusätzliche *id*- und *namespace*-Attribute. So wird das oben gezeigte Beispiel nach seiner Transformation zu:

```
<tr3 xmlns="http://www.hs-karlsruhe.de/kmm-m/tr3" id="dle1" out="Kaffee oder Tee?">
  <branch id="zucker" in="Kaffee" out="Mit Zucker?">
    <leaf id="dle5" in="Ja" out="Du bekommst ein Heißgetränk mit Zucker"/>
    <leaf id="dle7" in="Nein" out="Du bekommst ein Heißgetränk ohne Zucker"/>
  </branch>
  <branch id="dle10" in="Tee" origin="zucker" out="Mit Zucker?">
    <leaf id="dle5" in="Ja" out="Du bekommst ein Heißgetränk mit Zucker"/>
    <leaf id="dle7" in="Nein" out="Du bekommst ein Heißgetränk ohne Zucker"/>
  </branch>
</tr3>
```

Schema

Zur automatischen Validierung von tr3-Dateien wurde eine Dokumenttyp-Definition (DTD) und ein XML-Schema (XSD) erstellt. Der Formal Public Identifier (FPI) lautet:

```
-//tr3//tr3 Base Format 1.0//DE
```

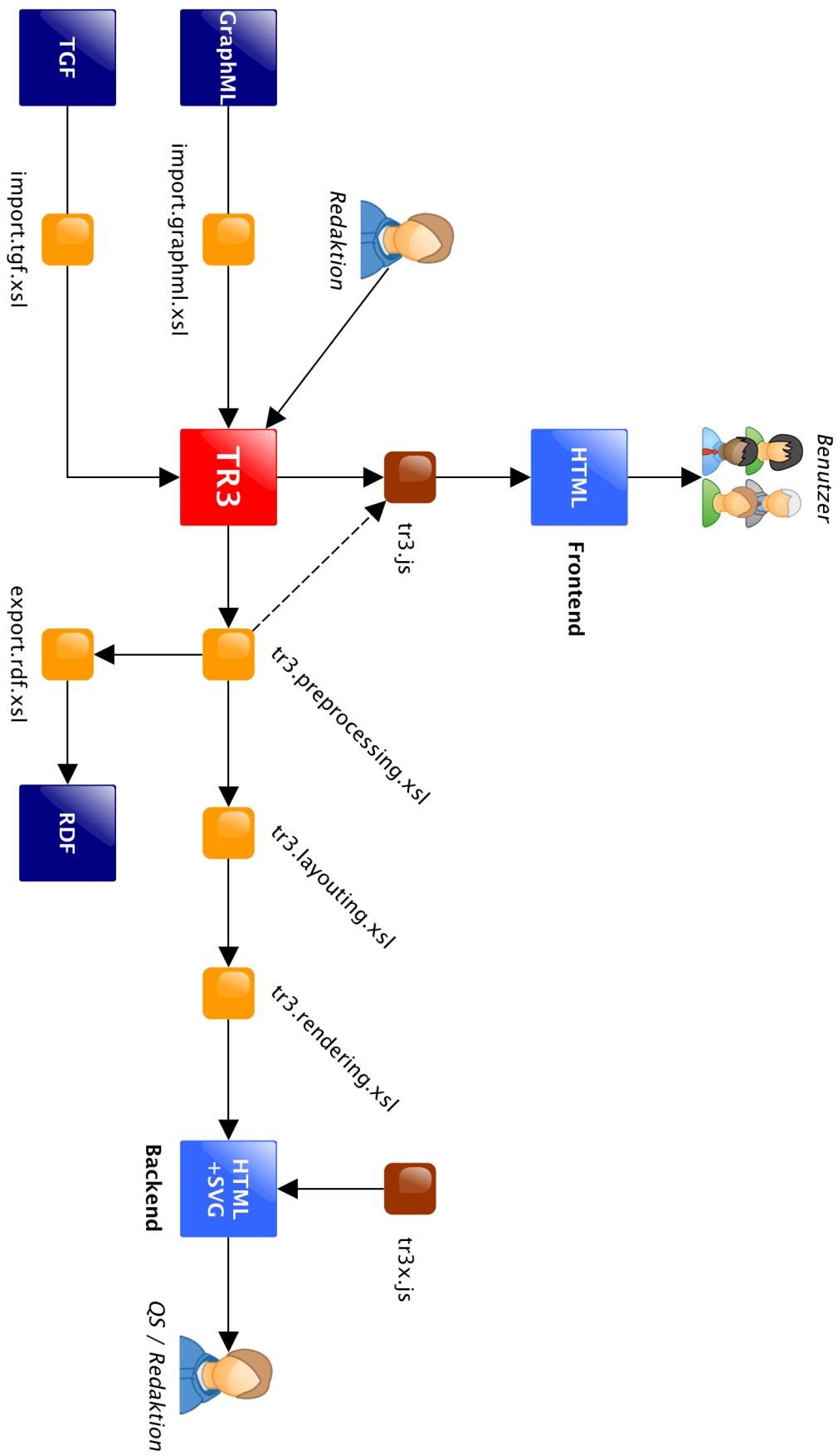
Die Einbindung der DTD in ein XML kann dementsprechend so erfolgen:

```
<!DOCTYPE tr3 PUBLIC "-//tr3//tr3 Base Format 1.0//DE" "tr3.dtd">
```

¹³ in diesem Beispiel ist es für die Ausgabe auch irrelevant, ob der Knoten kopiert oder referenziert wird.

Umwandlung

Übersicht tr3-Formate



Import von TGF

Das Trivial Graph Format ist eine sehr einfache textbasierte Form, gerichtete Graphen zu speichern.¹⁴ Ähnlich der GraphML-Struktur wird eine Liste aller Knoten und Kanten mit Beschriftungen gespeichert. Zwischen beiden Auflistungen steht das Trennzeichen #:

```
1 KA
2 B
#
1 2 Flug
```

Um TGF in tr3 umzuwandeln muss zunächst der Text als String eingelesen werden, der zunächst anhand des Trennzeichens „#“ in zwei Teile zerlegt wird. Anschließend werden einzelne Knoten und Kanten anhand der Trennzeichen {Leer} und {Umbruch} extrahiert und in eine flache XML-Struktur (eine Art Pseudo-GraphML) überführt:

```
<xsl:variable name="tgf.structure">
  <structure>
    <!-- start with nodes: split string at line breaks -->
    <xsl:for-each select="tokenize($tgf.nodes,'\n')">
      <!-- nodes are defined by two parts separated by a space char
      -> 'id'-'[space]'- 'name' -->
      <!-- id is before first space -->
      <xsl:variable name="node.id" select="substring-before(current(), ' ')" />
      <!-- name is everything after first space -->
      <xsl:variable name="node.out" select="substring-after(current(), ' ')" />

      <!-- test for empty values -->
      <xsl:if test="$node.id != '' and $node.out != ''">
        <node id="{ $node.id }" out="{ $node.out }" />
      </xsl:if>
    </xsl:for-each>

    <!-- continue with links: split string at line breaks -->
    <xsl:for-each select="tokenize($tgf.links,'\n')">
      <!-- links are defined by three parts separated by space chars
      -> 'start'-'[space]'- 'end'-'[space]'- 'label' -->
      <!-- get last two parts after first space char -->
      <xsl:variable name="temp.tail" select="substring-after(current(), ' ')" />
      <!-- get first part before first space char -->
      <xsl:variable name="link.start" select="substring-before(current(), ' ')" />
      <!-- split second part in last two parts -->
      <xsl:variable name="link.end" select="if (contains($temp.tail, ' '))
        then substring-before($temp.tail, ' ') else $temp.tail" />
      <xsl:variable name="link.label" select="substring-after($temp.tail, ' ')" />

      <!-- construct link -->
      <xsl:if test="$link.start != '' and $link.end != ''">
        <link start="{ $link.start }" end="{ $link.end }" label="{ $link.label }" />
      </xsl:if>
    </xsl:for-each>
  </structure>
</xsl:variable>
```

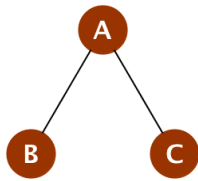
Die anschließende Verarbeitung entspricht der GraphML-Konvertierung. Zu beachten ist, dass bei TGF per Spezifikation im ASCII-Zeichensatz gespeichert wird, und es damit zu Problemen mit deutschen Sonderzeichen kommen kann.

Durch die Einschränkungen, die das TGF-Format mit sich bringt, ist zu beachten, dass keine Referenzierungen, Content-Verknüpfungen oder Klassenzuweisungen möglich sind, was es für den realen Einsatz als nicht brauchbar einstufen lässt.

¹⁴ <http://reference.wolfram.com/mathematica/ref/format/TGF.html>

Import von GraphML

Die Grundstruktur von GraphML-Dateien wurde bereits im Kapitel *XML-basierte Entscheidungslogiken* behandelt. Dieser graphenorientierte Aufbau muss bei der Konvertierung in eine hierarchische Struktur überführt werden. Dabei werden die Eigenschaften von gerichteten, azyklischen Graphen ausgenutzt:



```
<graph>
  <node id="A" />
  <node id="B" />
  <node id="C" />
  <edge source="A" target="B" />
  <edge source="A" target="C" />
</graph>
```

Laut Definition eines Out-Trees ist jeder Knoten durch genau einen gerichteten Pfad von der Wurzel aus erreichbar. Das heißt der Elternknoten eines Knotens lässt sich ermitteln, indem man den Ursprung der eingehenden Verbindung betrachtet.¹⁵

In XSLT lässt sich die *id* des Elternknoten mit folgendem XPATH-Ausdruck bestimmen:

```
<xsl:variable name="node.parent" select="//g:edge[@target=$node.id]/@source" />
```

Weißt man jedem Knoten in der noch flachen Struktur diese Information als *parent*-Attribut zu, kann mit einem rekursiven Template eine Baumstruktur erzeugt werden:

```
<xsl:template name="buildTree">
  <!-- named template parameter -->
  <xsl:param name="node" />
  <!-- for each node or ref element build node and call self-->
  <xsl:for-each select="$graph.layout//g:node[@parent=$node/@id]">

    <!-- ...
    generate element
    ... -->

    <xsl:call-template name="buildTree">
      <xsl:with-param name="node" select="." />
    </xsl:call-template>
  </xsl:for-each>
</xsl:template>
```

Das Template arbeitet wie folgt: Für jeden Knoten deren Elternelement das aktuelle Element ist (also für jeden Kindknoten) generiere das entsprechende Element und rufe das Template mit dem jeweiligen Kindknoten als Parameter wieder auf.

Während der Umwandlung in das tr3-Format werden je nach Position im Baum die jeweiligen Elemente erzeugt und mit Attributen versehen. Dabei werden *out*- sowie *in*-Attribute aus den Beschriftungen der Knoten bzw. Kanten erzeugt. Alle weiteren Attribute (z.B. *content*, *class*, etc.) werden aus sog. *Benutzerdefinierten Eigenschaften* (wie sie z.B. auch *yED* verwendet) verwaltet:

```
<node id="n0">
  <data key="d6"><![CDATA[content/example.html]]></data>
  <data key="d7"><![CDATA[Beispiel]]></data>
</node>
```

¹⁵ Laut Definition kann ein Knoten in einem Out-Tree nur ein Elternelement, und somit nur einen eingehende Verbindung besitzen


Ausgelesen werden können diese Datenfelder über den vom Editor oder Erfasser vergebenen *datakey*, der als *key*-Attribut jedem data-Element zugewiesen ist. Die Zuordnung dieser Schlüssel zu den jeweiligen Eigenschaften-Namen erfolgt zu Beginn jeder Datei in einer Art Verzeichnis:

```
<key attr.name="content" attr.type="string" for="node" id="d6"/>
<key attr.name="description" attr.type="string" for="node" id="d7"/>
```

Die in den Informationen aus den Data-Elementen werden je nach Schlüssel aufgelöst als Attribute den Elementen des tr3-Formats zugeordnet. Alle im GraphML enthaltenen Informationen zur visuellen Repräsentation des Graphen werden bei der Konvertierung verworfen.

Das zu Beginn des Abschnitts gezeigte Beispiel würde nach der Konvertierung nach tr3 so aussehen¹⁶:

```
<graph>
  <node id="A" />
  <node id="B" />
  <node id="C" />
  <edge source="A" target="B" />
  <edge source="A" target="C" />
</graph>
```



```
<tr3 id="A">
  <leaf id="B" />
  <leaf id="C" />
</tr3>
```

Ein typischer Knoten eines ausgearbeiteten tr3-Baums hat etwa folgende Erscheinung:

```
<branch id="display" in="Ja" class="app switch" content="content/panel.html"
out="Welches Display-Segment?">...</branch>
```

auswahl und Knoteneigenschaften:

Export nach RDF

Um die Vorteile von RDF (Universell, Implementierungsunabhängig) ausnutzen zu können ohne es als primäres Format nutzen zu müssen, besteht die Möglichkeit für einen Datenaustausch mit anderen Systemen das tr3-Format als RDF-Datei zu exportieren.

Voraussetzung für die Anwendung des Export-Stylesheets ist das Preprocessing der tr3-Datei mit aktiviertem `generate.ids`-Parameter (genauerer siehe Abschnitt *Preprocessing von tr3-Dateien*).

Dabei wird die hierarchische Baumstruktur in eine flache Graphstruktur zerlegt. Dabei werden zusätzlich zu den Knoten für jede Kombination¹⁷ aus in-Attribut und Ziel eine Kante (Verbindung) erzeugt. Dies ist nötig um eine globale Eindeutigkeit von Kanten innerhalb des erzeugten Graphen zu gewährleisten:

```
<tr3 out="Kaffee oder Tee?" xmlns="http://www.hs-karlsruhe.de/kmm-m/tr3">
  <leaf in="Kaffe" out="Du bekommst einen Kaffee" />
  <leaf in="Tee" out="Du bekommst einen Tee" />
</tr3>
```

wird zu (leicht gekürzt zur besseren Lesbarkeit):

¹⁶ Bei diesem Beispiel wird auf die sonst notwendigen zusätzlichen Attribute (in, out, etc.) zu Gunsten der Lesbarkeit verzichtet.

¹⁷ Durch Referenzierungen kann ein Knoten verschiedene in-Attribute besitzen.

```

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:tr3="http://www.hs-karlsruhe.de/kmm-m/tr3">
  <rdf:Description rdf:about="http://www.hs-karlsruhe.de/kmm-m/tr3/node#dle1">
    <tr3:out>Kaffee oder Tee?</tr3:out>
    <tr3:answers>
      <rdf:alt>
        <rdf:li rdf:resource="[...]/kmm-m/tr3/connection#Kaffe-dle3"/>
        <rdf:li rdf:resource="[...]/kmm-m/tr3/connection#Tee-dle5"/>
      </rdf:alt>
    </tr3:answers>
  </rdf:Description>
  <rdf:Description about="[...]/kmm-m/tr3/connection#Kaffe-dle3">
    <tr3:in>Kaffe</tr3:in>
    <tr3:target rdf:resource="[...]/kmm-m/tr3/node#dle3"/>
  </rdf:Description>
  <rdf:Description about="[...]/kmm-m/tr3/connection#Tee-dle5">
    <tr3:in>Tee</tr3:in>
    <tr3:target rdf:resource="[...]/kmm-m/tr3/node#dle5"/>
  </rdf:Description>
  <rdf:Description rdf:about="[...]/kmm-m/tr3/node#dle3">
    <tr3:result>Du bekommst einen Kaffee</tr3:result>
  </rdf:Description>
  <rdf:Description rdf:about="[...]/kmm-m/tr3/node#dle5">
    <tr3:result>Du bekommst einen Tee</tr3:result>
  </rdf:Description>
</rdf:RDF>

```

Werden Referenzierungen verwendet, werden diese während der Transformation aufgelöst und eine direkte Kante zum Zielknoten erzeugt. Separate `ref`-Knoten werden deshalb bei der Ausgabe als RDF nicht benötigt.

Preprocessing

Alle `tr3`-Entscheidungsbäume können durch ein sog. Preprocessing (dt.: Vorverarbeitung) vor dem Einsatz in einer Anwendung oder einer Testumgebung verarbeitet werden. Dabei werden mehrere wichtige Schritte durchlaufen:

- Analyse des Baums auf Fehler (leere Attribute, tote Links, rekursive Kopien, etc.)
- Erzeugen von *id*-Attributen für alle Knoten, die keine *id* besitzen
- Auflösen von Wiederverwendungen (Teilbäume kopieren)
- Korrigieren von Fehlern (rekursive Kopien in Referenzen umwandeln)

Werden Fehler entdeckt oder Korrekturen vorgenommen wird dies in der Konsole des XSL-Transformators zurückgegeben, z.B.:

```

tr3 notice: changed copy attribute to link attribute
tr3 error: @out empty on branch
tr3 error: no id ,falsy_id'

```

Ebenfalls werden je nach Fehlerart spezielle Fehlerknoten erzeugt, um die Wohlgeformtheit des Baumes zu gewährleisten und in einer Quelltextsuche oder einer Testumgebung schnell zur Fehlerstellengelangen zu können. Wo das nicht möglich ist (z.B. bei Attributen) muss auf `tr3x` (s.u.) zurückgegriffen werden. Die Umwandlung von rekursiven Kopien, die automatische Vergabe von ids und die Erzeugung von Teilbaum-Kopien können mit folgenden Parametern gesteuert werden:

<code>avoid.recursion</code>	1: Umwandlung falscher Attribute, 0: Erzeugung von Fehlerknoten
<code>enable.reuse</code>	1: Teilbäume werden kopiert, 0: Nur Referenzen werden erzeugt
<code>generate.ids</code>	1: Alle Knoten haben ids, 0: Nur ids wo explizit vergeben

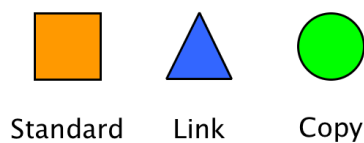
Erfassung

Allgemein

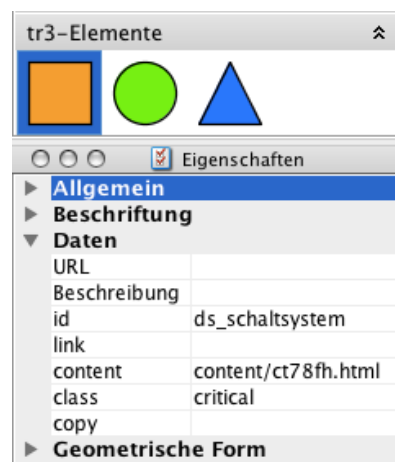
Das tr3-Format kann bei kompakten Bäumen direkt als XML erfasst werden oder in einem grafischen Editor erstellt werden. Dabei bietet sich der Graphen-Editor *yED* an, dessen natives Speicherformat *GraphML* in das tr3-Format transformiert werden kann.

Grafischer Editor yED

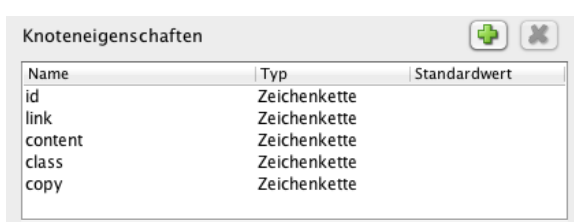
Um bei der Erstellung in yED zwischen verschiedenen Knotentypen zu unterscheiden, werden drei unterschiedliche geometrische Repräsentationen für die drei funktionalen Knotentypen verwendet. Standardknoten (tr3, branch, leaf) werden als Quadrate, Links als Dreiecke und Beginn von Wiederverwendungen als Kreise dargestellt. Die verschiedenen Knotentypen können vom Ersteller aus einer Seitenleiste heraus direkt mit dem Baum verknüpft werden.



Oben: Die drei möglichen Baum-Elemente, die bei der Erstellung in yED verwendet werden können: Standard, Link oder Copy.
Rechts: Ausschnitt der Seitenleiste mit Knotentypen und Benutzerdefinierten Attributen.



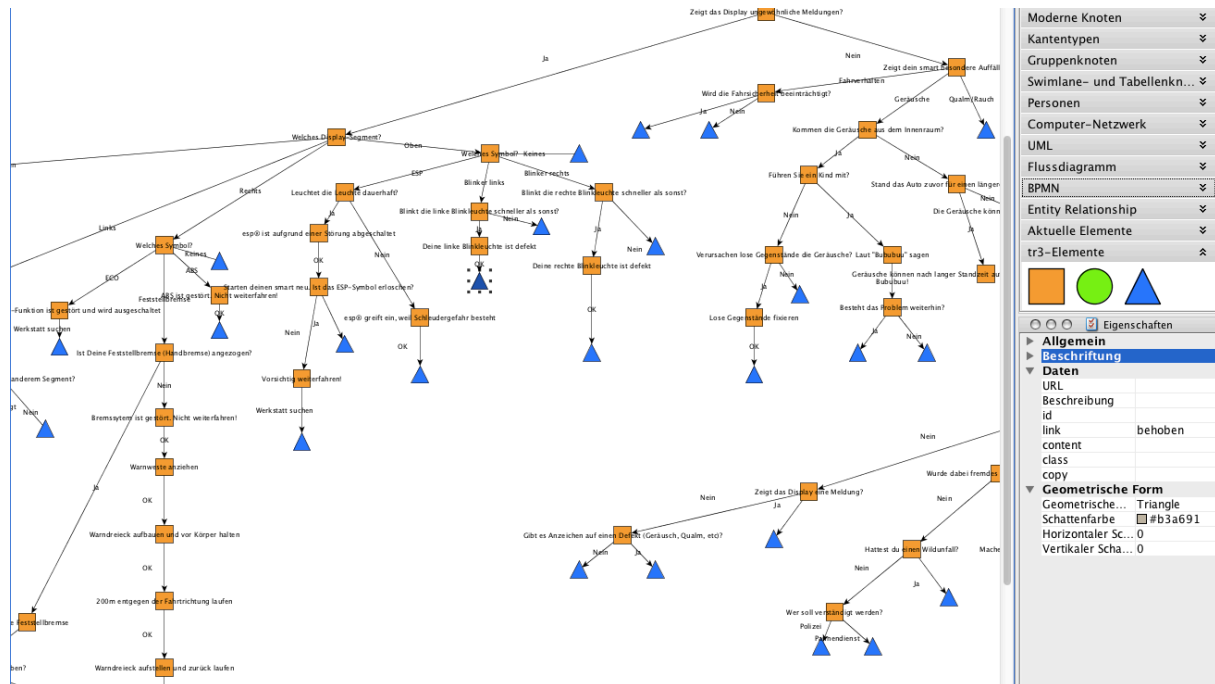
Für die Auszeichnung der Knoten mit Attributen (Metadaten) werden Benutzerdefinierte Eigenschaften definiert. Nach einmaliger Konfiguration können diese Eigenschaften nun jedem Knoten zugewiesen werden und werden beim Speichern in den *data*-Elementen des GraphML-Formats gespeichert:



Oben links: Bearbeitungsansicht eines referenzierten Knotens.
Oben rechts: Bearbeitungsansicht eines referenzierenden Knotens.
Unten links: Baumweite Definition Benutzerdefinierter Knoteneigenschaften.

Grafische Oberfläche

Die grafische Oberfläche des Graphen-Editors *yED* mit einem geladenen Baum:



Qualitätskontrolle

Notwendigkeit

Mit steigender Komplexität von Bäumen steigt auch deren Fehlerhäufigkeit. Diese können von leeren Attributen bis hin zu nicht vorhandenen Zielen bei Verlinkung oder rekursiver Wiederverwendung von Teilbäumen reichen. Um solche Fehler zu vermeiden und einen Entscheidungsbaum ‚untersuchbar‘ zu machen, ist es notwendig ein Werkzeug zur Qualitätssicherung zu haben, das den Autor darin unterstützt, Fehler zu finden, die Struktur zu analysieren und verschiedene Szenarien vor einer Veröffentlichung zu testen.

Aus diesen Gründen lässt sich aus jeder tr3-Datei automatisiert eine Testumgebung erzeugen: den „tr3 explorer“, kurz: *tr3x*.

Erzeugen einer tr3x-Testumgebung

Eine Testumgebung ist eine Webanwendung, die aus einer tr3-Datei erzeugt wird und aus generierten Teilen (die sich je nach Ausgangsdateien unterscheiden) und statischen Teilen (über alle Umgebungen hinweg gleich) bestehen. Zentraler Bestandteil der Anwendung ist eine bei der Transformation generierte SVG (Scalable Vector Graphic), die durch eine JavaScript-Anwendung analysiert wird. Die generierten Anteile werden durch eine mehrstufige Transformation erzeugt:

- | | |
|------------------|------------------------------------|
| 1. Preprocessing | <code>tr3.preprocessing.xml</code> |
| 2. Layouting | <code>tr3.layouting.xml</code> |
| 3. Rendering | <code>tr3.rendering.xml</code> |

Preprocessing

Der grundlegende Ablauf des Preprocessing wurde bereits im vorhergehenden Kapitel behandelt. Für die Vorverarbeitung beim Erzeugen einer tr3x-Testumgebung sind einige Besonderheiten zu beachten:

- `generate.ids` muss aktiviert (1) sein
- Statistiken zur Pfadlänge und Wiederverwendungsquote sind nur dann akkurat, wenn Teilbäume immer kopiert werden (nicht nur referenziert) und der Parameter `avoid.recursion` aktiviert (1) ist

Die resultierende Datei ist Grundlage für das Layouting.

Layouting

Beim Layouting werden den einzelnen Knoten wichtige Eigenschaften für die spätere Darstellung als Attribute zugewiesen:

- Level: Die hierarchische Ebene auf der sich der Knoten im Baum befindet
- Width: Die maximale Kindknoten-Anzahl der Nachfahren eines Knoten (Breite)
- Class: In diesem Attribut werden mehrere Eigenschaften zusammengefasst:
 - Die eigentliche Knotenklasse (`tr3-class`-Attribut)
 - *copy* - wenn es sich um einen kopierten Knoten handelt
 - *link* - wenn es sich um einen Link handelt
 - Die ids aller Vorfahren, jeweils mit dem Präfix *d-of-{id}*
 - Die id des Elternelements, jeweils mit dem Präfix *c-of-{id}*

Des Weiteren werden leere *content*-Attribute entfernt und *ref*-Elemente in *leaf*-Elemente mit dem *out*-Attribut ‚[link]‘ + Ziel-*id* versehen.

Die resultierende Datei ist Grundlage für das Rendering.

Rendering

Die Generierung der eigentlichen Anwendung findet in der letzten Transformation statt. Dabei wird eine HTML-Datei aus vorgefertigten Bausteinen (*head.xml* und *panel.xml*) und einer bei der Transformation generierten SVG zusammengesetzt.

Zunächst werden für jeden Knoten des Baums drei SVG-Gruppen (*g*) erzeugt:

- Ein Rechteck (*rect*) mit Beschriftung (*text*) und Beschreibung (*desc*)
- Alle ausgehenden Verbindungen (*line*) mit Beschriftungen (*text*)
- Bei Links: Der ausgehende Link (*line*)

Die Positionierung von Elementen wird dabei auf Basis der beim Layouting übergebenen Werte nach folgenden Algorithmen berechnet:

y-Position:

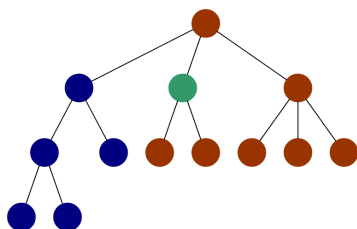
```
<xsl:variable name="y" select="@level * 4"/>
```

Das Level des aktuellen Knoten mit 4 multipliziert ergibt die y-Position¹⁸ des Objekts.

x-Position:

```
<xsl:variable name="x" select="
  (sum(
    preceding::*[
      @level = current()/@level
      or
      (
        count(*) = 0
        and
        @level <= current()/@level
      )
    ]
  )/@width)
  + (@width div 2)
  )
  * 5
"/>
```

Die fünffache Summe der Breite aller Knoten auf der *preceding*-Achse, die auf dem gleichen Level oder kinderlos und auf einem geringeren Level als der aktuelle Knoten sind plus die Hälfte der Breite des aktuellen Knotens ergibt die x-Position des Objekts.



Die *preceding*-Achse bildet in XSL/XPath alle vorhergehenden Knoten ab, die sich auf dem gleichen hierarchischen Level oder tiefer als der aktuelle Knoten befinden. Im dargestellten Beispiel ist der aktuelle Knoten grün dargestellt. Die auf der *preceding*-Achse liegenden Knoten blau markiert.

¹⁸ Eine Koordinaten-Einheit entspricht in SVG einer regelmäßigen Einteilung relativ zur Gesamtzeichenfläche (siehe: <http://www.w3.org/TR/SVG11/intro.html#TermUserUnits>)

Allen generierten Objekten werden die beim Layouting aggregierten Klassen zugewiesen sowie zusätzliche Identifier, die für die interne Verarbeitung der Testumgebung notwendig sind (ids, SVG-Klassifizierung, etc.).

Das out-Attribut des tr3-Formats wird als Beschriftung eines Baumknotens verwendet. Aufgrund der fehlenden Textfluss-Funktionen von SVG wird es allerdings gekürzt wenn seine Länge 23 Zeichen (entspricht ca. der Breite eines gerenderten Knotens) überschreitet. Das content-Attribut (also der Pfad zur verknüpften Inhaltsdatei) wird in das desc-Element geschrieben, das der Gruppe eine Knotens untergeordnet ist.

Eine Übersicht der erzeugten Elemente mit tr3-Entsprechung in Xpath-Notation für jeden Knoten des Layout-Baums:

g	g	g
rect (branch leaf) desc (@content) text (@out)	line (*) text (*/@in)	line (ref)

Technische Grundlagen

Die tr3x-Testumgebung basiert in ihren statischen Anteilen auf HTML + SVG und wird durch die Skriptsprache JavaScript dynamisch. Als Funktionsbibliothek für die Selektion und Manipulation von DOM-Elementen kommt D3¹⁹ zum Einsatz.

Die Anwendung basiert auf einem serverseitigen Ansatz, bei dem pro Baum eine eigene Testumgebung generiert wird, bei der der jeweilige Baum ‚fest‘ integriert ist und clientseitig untersucht werden kann. Dem entgegen steht ein clientseitiger Ansatz, der im Kapitel Alternative Testumgebung kurz vorgestellt wird.

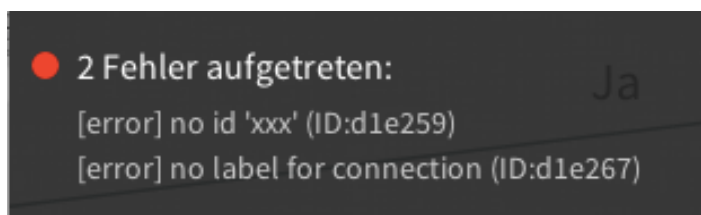
Funktionen der Testumgebung

Fehler-Reporting

Beim Preprocessing erzeugte Fehlerknoten werden aggregiert und durch die Prüfung auf leere in-Attribute²⁰ ergänzt.

```
// select all generated error nodes by class and push them to array
d3.selectAll('text.tr3-error:not(.copy)').each(function() {
  _text = d3.select(this).text();
  _id = d3.select(this.parentNode).attr('id');
  _errors.push({'text': _text, 'id': _id});
});
```

Für jeden gefundenen Fehler wird ein verlinkter Eintrag im Fehlerreport generiert:



¹⁹ D3: Data Driven Documents (d3js) von Mike Bostock - <http://d3js.org>

²⁰ Diese Prüfung wird auch beim Preprocessing durchlaufen. Da es sich aber um eine Attribut-Prüfung handelt, werden keine Fehlerknoten erzeugt, die aggregiert werden können.

Analyse

Beim Start der Anwendung werden bestimmte Kennzahlen zum jeweiligen Baum gesammelt und ausgegeben. Diese sind:

- Minimale, maximale und durchschnittliche Pfadlänge
- Gesamtanzahl Knoten, Blattknoten, kopierte Knoten
- Wiederverwendungsanteil (nur wenn kopierte Knoten verwendet werden)

Die Pfadlängen-Analyse gibt nur dann korrekte Ergebnisse zurück wenn Wiederverwendungen durch Kopien und nicht durch Verweise erzeugt werden. Der Grund dafür ist, das beim Abzählen der Pfadpunkte Links nicht aufgelöst werden können, da sonst die Gefahr einer rekursiven Schleife besteht. Ein Link wird daher als Endpunkt interpretiert und die Zählung stoppt an dieser Stelle.

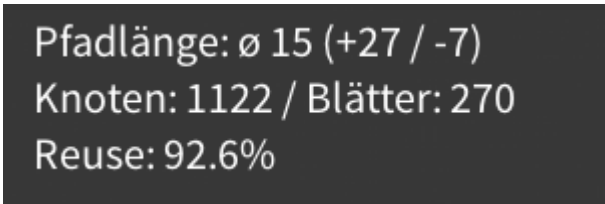
Bei der Pfadlängen-Analyse wird für jeden Blattknoten die Anzahl seiner Vorfahren bestimmt:

```
for (var p = 0; p < _leafs.length; p++){  
  _paths.push(_leafs[p].descs.length)  
}
```

Um den Wiederverwendungsanteil zu bestimmen, wird das Verhältnis von ‚normalen‘ zu kopierten Knoten berechnet und auf eine Nachkommastelle gerundet:

```
tr3x.stats.reuse = Math.round(tr3x.stats.copied / tr3x.stats.total * 1000) / 10;
```

Die errechneten Kennzahlen werden unterhalb des Fehlerreports in der Anwendung angezeigt:

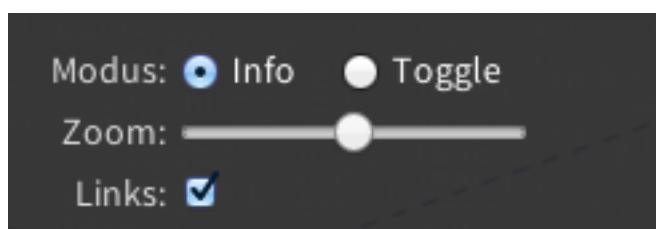
A dark grey rectangular box containing white text. The text is arranged in three lines: 'Pfadlänge: ø 15 (+27 / -7)', 'Knoten: 1122 / Blätter: 270', and 'Reuse: 92.6%'.

Pfadlänge: ø 15 (+27 / -7)
Knoten: 1122 / Blätter: 270
Reuse: 92.6%

Navigation

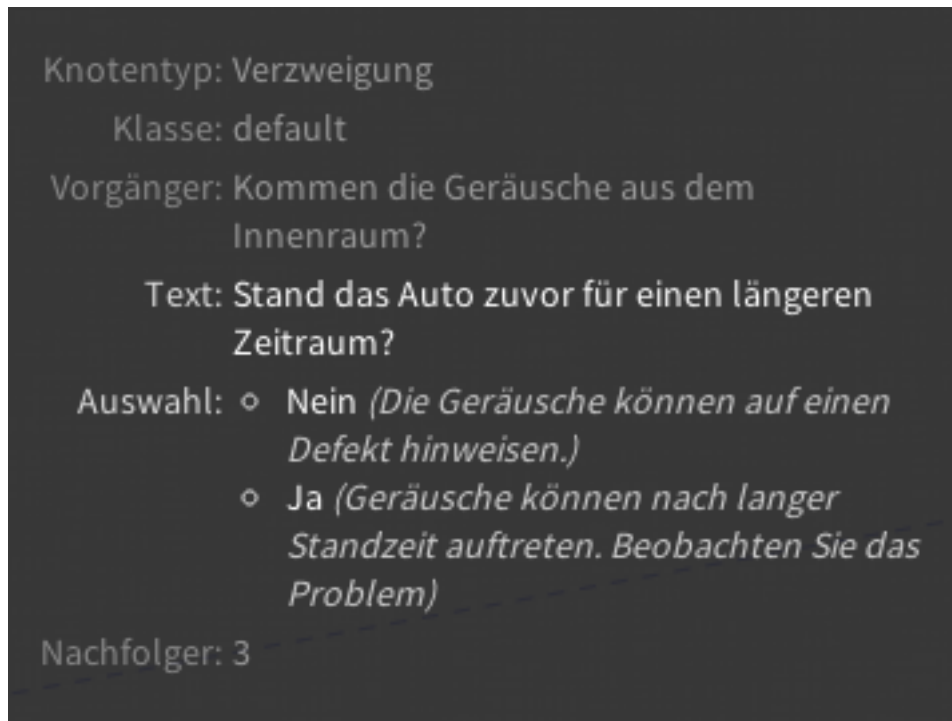
Es gibt verschiedene Arten, durch den Baum zu navigieren:

- Durch einfaches scrollen kann der Baum in jede Richtung untersucht werden
- Im *Info*-Modus kann durch das Anklicken der verwandten Knoten im Informations-Panel an die entsprechenden Stellen gesprungen werden
- Im *Toggle*-Modus können Teilbäume auf- und zugeklappt werden um eine bessere Übersicht zu bekommen
- Mit Hilfe des Zoom-Reglers kann die Skalierung des generierten SVGs beeinflusst werden
- Bei Bedarf können alle Links ausgeblendet werden



Knoteninformationen

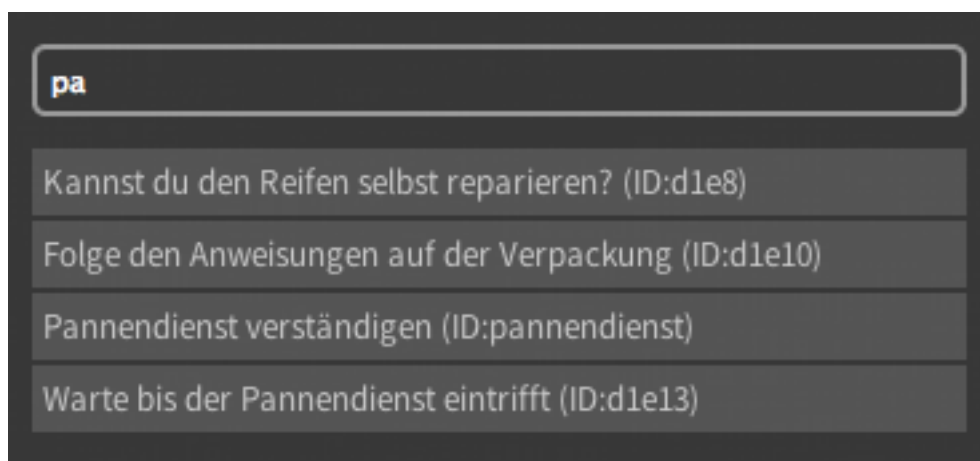
Im Modus *Info* können durch Klick auf einen Knoten Informationen über ihn gesammelt werden. Die angezeigten Informationen unterscheiden sich nach Knotentyp (siehe API-Eintrag zu `tr3x.writeNodeInformation`).



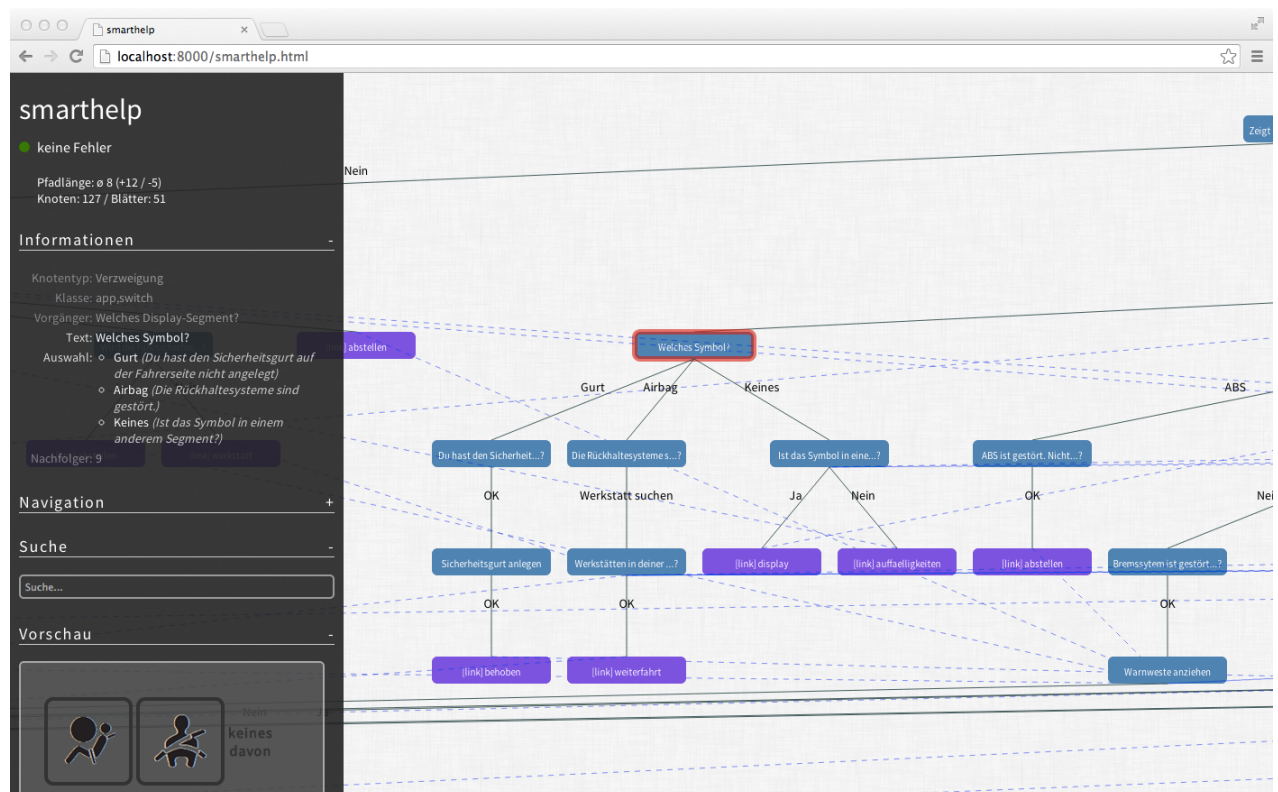
Suche

Alle out-Attribute von Standardknoten können über den Baum hinweg mit einer *search-as-you-type*-Suche durchsucht werden, dabei wird für jeden Tastenanschlag ein Index durchsucht und die Ergebnisse als verlinkte Liste im entsprechenden Panel angezeigt:

```
// iterate over index and push matches to results (except links and errors)
for (var j = 0; j < tr3x.nodes.length; j++) {
  if ( ~tr3x.nodes[j].text.toLowerCase().indexOf(term.toLowerCase()) &&
    !~tr3x.nodes[j].text.indexOf('[link]') &&
    !~tr3x.nodes[j].text.indexOf('[error]') ){
    _results.push(tr3x.nodes[j]);
  }
}
```



Gesamtübersicht



Das Anwendungspanel wird als Overlay über dem eigentlichen Baum am linken Rand angezeigt und ist leicht durchlässig, um auch darunter liegende Teile des Baums sehen zu können. Das Gesamtpanel ist wiederum unterteilt in mehrere Kategorien, die sog. *Sub-Panels*. Diese können je nach Bedarf auf- oder zugeklappt werden. Im unteren Bereich wird verknüpfter Zusatzcontent in ein grau hinterlegtes *Preview-Panel* nachgeladen.

API

Übersicht

Die Anwendung stellt eine objektbasierte JavaScript-Schnittstelle zur Verfügung, die von anderen Scripten oder über die Browser-Konsole durch das *tr3x*-Objekt angesteuert werden kann.

Eigenschaften

Eigenschaft	liefert	Beschreibung
<code>tr3x.dict</code>	<i>Objekt</i>	Enthält pro Sprache (z.B. <code>de</code>) ein Unterobjekt mit der Zuordnung von sprachspezifischen Labels zu intern verwendeten sprachneutralen Strings
<code>tr3x.errors</code>	<i>Array</i>	Enthält pro Fehler, der beim angezeigten Baum aufgetreten ist, ein Objekt mit Fehlertext (<code>text</code>) und ID des dem Fehler zugeordneten Knoten (<code>id</code>).
<code>tr3x.leafs</code>	<i>Array</i>	Enthält pro Endknoten (Blatt) des Baumes ein Objekt mit Knotentext bzw. <i>out</i> -Attribut (<code>text</code>), Knoten-ID (<code>id</code>) und eine Liste aller Vorfahren (<code>descs</code>)
<code>tr3x.l</code>	<i>Objekt</i>	Enthält die Zuordnung von Labels der aktuellen Sprache zu intern verwendeten sprachneutralen Strings

<code>tr3x.nodes</code>	<i>Array</i>	Enthält pro Knoten des Baumes ein Objekt mit Knotentext bzw. <i>out</i> -Attribut (<code>text</code>) und Knoten-ID (<code>id</code>)
<code>tr3x.root</code>	<i>String</i>	Liefert die ID des Root-Knotens
<code>tr3x.scale</code>	<i>String</i>	Liefert den Skalierungsfaktor des generierten SVGs
<code>tr3x.stats</code>	<i>Objekt</i>	Enthält die Eigenschaften durchschnittliche (<code>avg</code>), maximale (<code>max</code>) und minimale (<code>min</code>) Pfadlänge, Anzahl aller (<code>total</code>), kopierter (<code>copied</code>) und Blatt- <code>(leafs)</code> Knoten, sowie den Wiederverwendungsanteil (<code>reuse</code>) in Prozent
<code>tr3x.title</code>	<i>String</i>	Liefert den Titel des Baums (title-Attribut des <code>tr3</code> -Knotens)
<code>tr3x.zoom</code>	<i>String</i>	Liefert die aktuelle Zoomstufe der Anwendung (Reglerstellung)

Methoden

Methode	Parameter	Beschreibung
<code>tr3x.bindEvents</code>	-	Verknüpft <code>events</code> von Bedienelementen mit den ihnen zugeordneten Funktionen
<code>tr3x.buildIndex</code>	-	Erzeugt einen Index über alle Knoten (<code>tr3x.nodes</code>)
<code>tr3x.buildUI</code>	-	Erzeugt den Titel des Baums und erzeugt Panel-Buttons (minimieren/maximieren)
<code>tr3x.generateErrorReport</code>	-	Sammelt und findet Fehlerknoten im Baum und generiert einen Fehlerreport mit Verlinkungen
<code>tr3x.generateNodeInformation</code>	<i>node</i>	Sammelt und erstellt Knoteninformationen für den übergebenen Knoten
<code>tr3x.generateTreeStats</code>	-	Sammelt und generiert Kennzahlen des Baumes
<code>tr3x.getNodeAdditional</code>	<i>node</i>	Wenn der übergebene Knoten vom Typ <i>link</i> ist, wird eine Link zum Zielknoten zurückgegeben.
<code>tr3x.getNodeChoices</code>	<i>id</i>	Für die übergebene Knoten-ID werden für alle Auswahlmöglichkeiten als Links zu den entsprechenden Zielen zurückgegeben (<i>link</i> -Knoten werden aufgelöst)
<code>tr3x.getNodeClass</code>	<i>node</i>	Für den übergebenen Knoten werden alle <code>tr3</code> -Klassen zurückgegeben
<code>tr3x.getNodeDescendantsCount</code>	<i>id</i>	Für die übergebene Knoten-ID wird die Zahl der Nachfolger zurückgegeben
<code>tr3x.getNodeParent</code>	<i>node</i>	Für den übergebenen Knoten wird das Elternelement zurückgegeben
<code>tr3x.getNodeType</code>	<i>node</i>	Für den übergebenen Knoten wird bei Standardknoten der strukturelle Typ, bei Spezialknoten der funktionale Typ zurückgegeben
<code>tr3x.init</code>	-	Startet die Anwendung (neu)
<code>tr3x.jumpToNode</code>	<i>id</i>	Scrollt zum Knoten mit der übergebenen ID
<code>tr3x.localizeUI</code>	<i>locale</i>	Lokalisiert die Oberfläche entsprechend des übergebenen <code>locale</code> -Kürzels
<code>tr3x.scrollLeftTween</code>	<i>position</i>	Scrollt nach rechts zur angegebenen Positi-

		on (x-Offset)
<code>tr3x.scrollTopTween</code>	<i>position</i>	Scrollt nach unten zur angegebenen Position (y-Offset)
<code>tr3x.searchNodeText</code>	<i>string</i>	Durchsucht den Index nach Übereinstimmungen mit dem übergebenen String, liefert die Ergebnisse zurück und generiert eine Auswahlliste
<code>tr3x.setNodeActive</code>	<i>node</i>	Markiert den übergebenen Knoten als aktiv
<code>tr3x.showContentPreview</code>	<i>node</i>	Lädt den mit dem übergebenen Knoten verknüpfte Content in das Preview-Fenster
<code>tr3x.toggleLinks</code>	-	Ändert die Sichtbarkeit der Link-Verbindungslien
<code>tr3x.toggleNodeState</code>	<i>node</i>	Ändert die Sichtbarkeit aller Kindknoten (minimieren/maximieren) und markiert ggf. den übergebenen Knoten als minimiert
<code>tr3x.toggleSubPanel</code>	<i>node</i>	Ändert die Sichtbarkeit des übergeben Subpanels
<code>tr3x.writeNodeInformation</code>	<i>title, desc, type, choices, additional, tr3class, parent</i>	Schreibt die übergebenen Parameter Titel, Nachfolger, Knotentyp, Auswahlmöglichkeiten, Zusatzinformationen, Klasse und Elternknoten als Knoteninformationen in das entsprechende Subpanel.
<code>tr3x.zoomCanvas</code>	<i>zoom</i>	Vergrößert die Baumgrafik auf den übergebenen Zoom-Wert (default: 30)

Alternative Testumgebung

Als eine clientbasierte alternative Testumgebung steht eine Variante bereit, die das tr3-Format direkt verarbeiten kann und das SVG dynamisch im Browser erzeugt. Damit ist eine vorherige Transformation nicht mehr notwendig.

Die dynamische Generierung des SVGs basiert hierbei auf einem in D3 integrierten Algorithmus zum Aufbau von klappbaren Bäumen, der als Datenquelle ein JavaScript-Objekt verwendet, welches üblicherweise aus einer JSON²¹-Datei geparkt wird.

Um auch XML verarbeiten zu können muss vorher eine Umwandlung in eine JavaScript-kompatible Objektnotation erfolgen (JSON). Konzepte zu einer möglichst verlustfreien Umwandlung der beiden Formate werden unter dem Namen JXON (lossless JavaScript XML Object Notation) zusammengefasst. Ein Standard existiert in diesem Bereich nicht, jedoch haben sich mit der Zeit einige Konventionen etabliert (z.B. *Parker Convention*, *BadgerFish Convention*, etc.).²²

²¹ JSON: Java Script Object Notation

²² <https://developer.mozilla.org/en-US/docs/JXON>

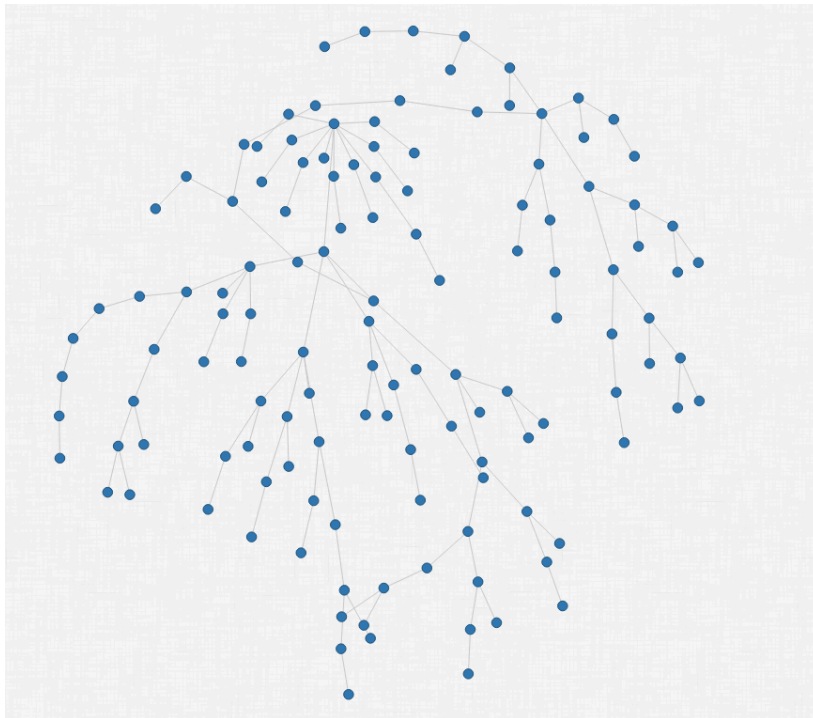
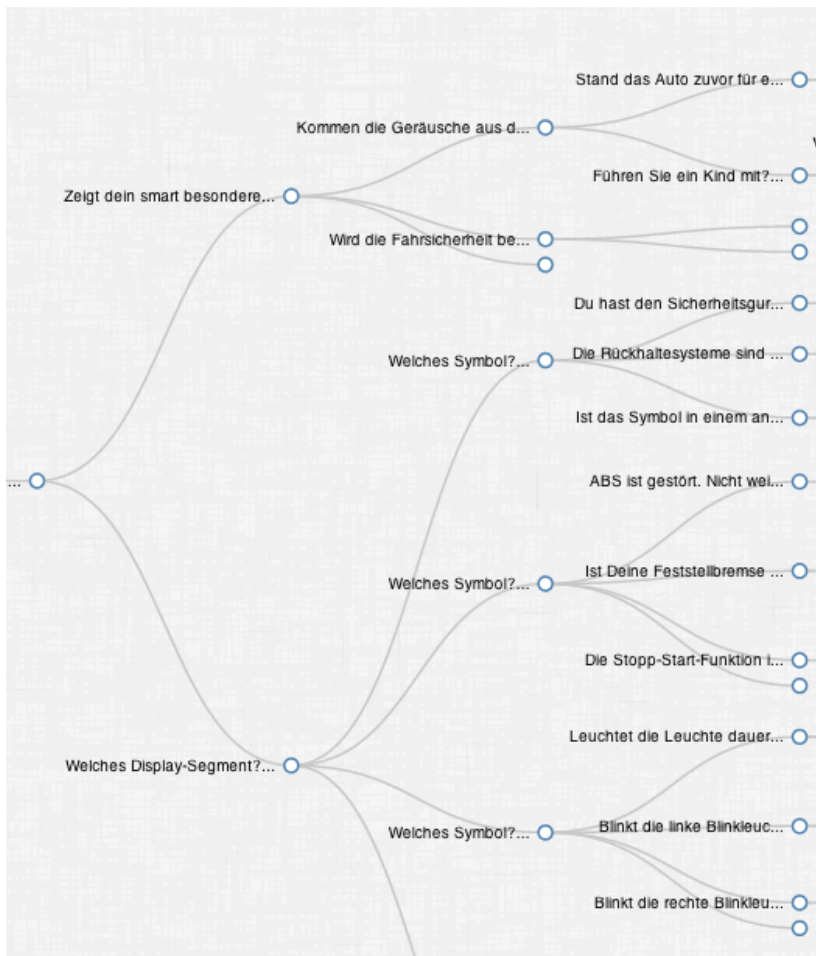
D3 benötigt als Grundlage für das Zusammensetzen des Baums ein speziell aufgebautes JavaScript-Objekt, das Hierarchien durch eine *children*-Property erzeugt, welche die untergeordneten Knoten enthält (die wiederum *children*-Properties haben können):

```
▼ Object {xmlns: "http://www.hs-karlsruhe.de/kmm-m/tr3", title: "smarthelp",  
  out: "Warnblinkanlage einschalten", children: Array[1], x0: 580...} ⓘ  
  ▼ children: Array[1]  
    ▼ 0: Object  
      ▼ children: Array[2]  
        ► 0: Object  
        ► 1: Object  
          length: 2  
        ► __proto__: Array[0]  
      depth: 1  
      id: 92  
      in: "OK"  
      out: "Kann dein smart noch fahren?"
```

Durch Abwandlung der JXON-Konventionen kann ein so aufgebautes Objekt direkt aus einer XML-Datei erzeugt werden und anschließend von D3 verarbeitet werden. Diese Form der Darstellung hat den Vorteil, dass bei jedem auf- oder zuklappen eines Astes die Positionen aller Knoten neu berechnet wird und sich dadurch übersichtlicher positionieren. Des Weiteren bietet D3 weitere (z.B. graphenorientierte) Visualisierungsformen an (*siehe Grafiken nächste Seite*).

Allerdings stößt die Initial- und Neuberechnung der Positionen aller Knoten bei sehr großen Bäumen (> 2000 Knoten) an Ihre Grenzen und das Layout des Gesamtbaumes kann zerstört aussehen.

Visualisierungen der alternativen Testumgebung



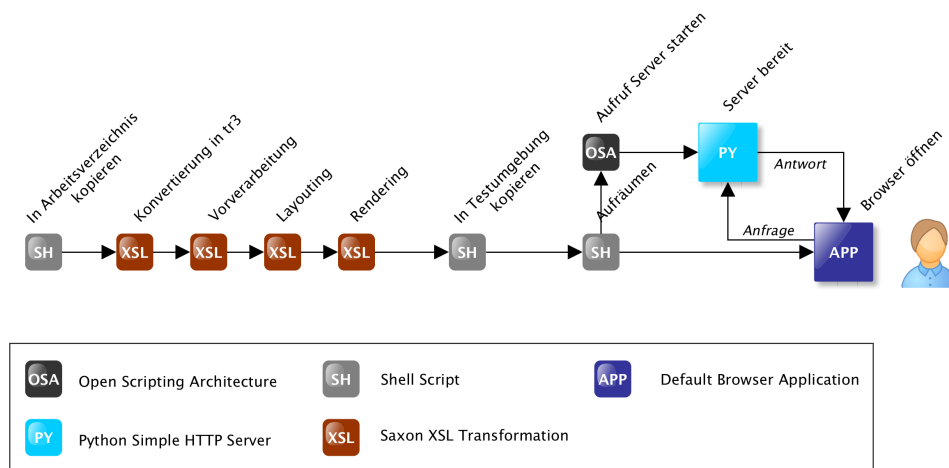
Veröffentlichung

Automatisierte Workflows

Um oft wiederkehrende Prozesse zu beschleunigen, wurden automatisierte Workflows in Shell-Skripten zur Ausführung auf unixoiden²³ Systemen verfasst, welche die Einzelschritte nacheinander ausführen und Dateien verwalten.

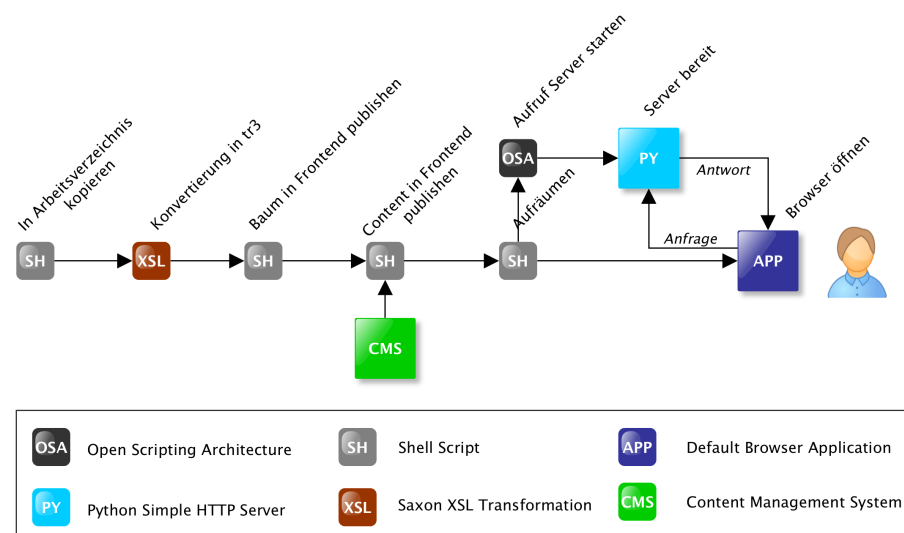
Testen (aus GraphML)

Nach dem Import der GraphML-Datei wird über eine mehrstufige Transformation eine Testumgebung erzeugt. Die generierte Datei wird in die tr3x-Ordnerstruktur eingebettet. Ein HTTP-Server wird parallel zum Aufruf eines Browsers gestartet.



Publishing (aus GraphML)

Nach dem Import der GraphML-Datei wird zunächst der fertige tr3-Baum und danach der benötigte Content in das Frontend kopiert. Ein HTTP-Server wird parallel zum Aufruf eines Browsers gestartet.



²³ getestet unter Mac OS X. Als Batch-Files auch unter Windows-Systemen zu verwenden.

Einsatz in Anwendung

Vorbemerkung

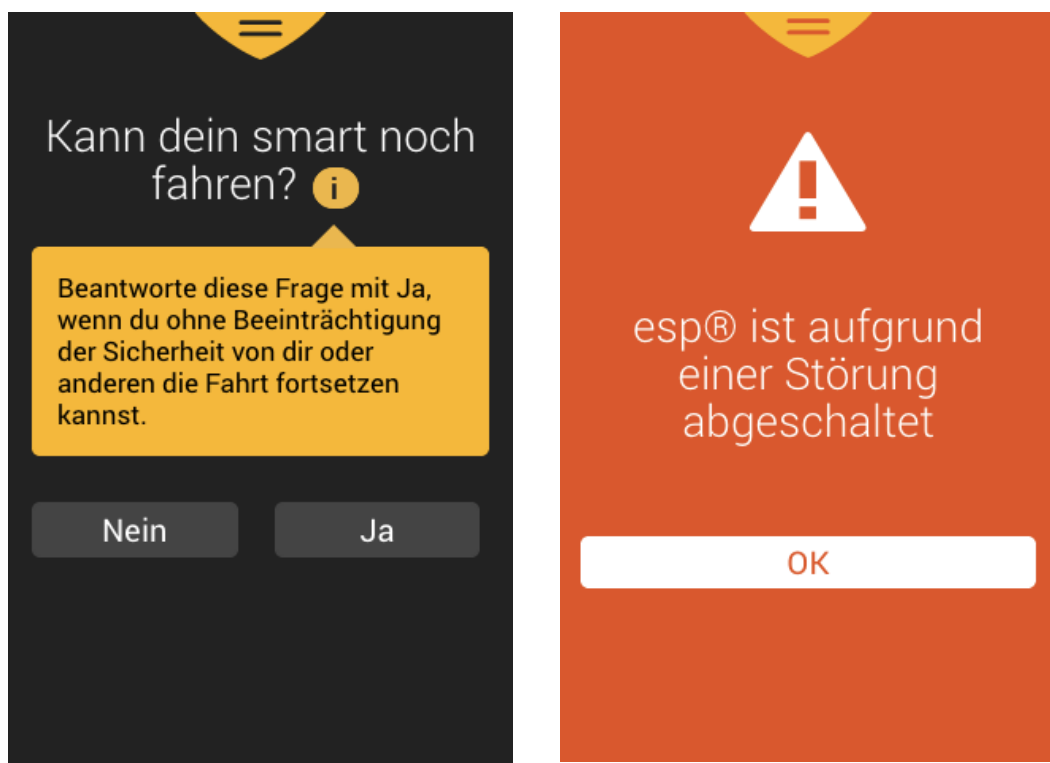
Die Arbeiten dieses Kapitels sind zum größten Teil im Projekt ‚smart help – Geführte Pan-nenhilfe‘ der Vorlesung *Media Engineering* bei Prof. Schober entstanden.

Laufzeitumgebung / Interpreter

Für den Einsatz in webbasierten Umgebungen (Websites, Browser-Apps, Web-Apps für Smartphones) wird Laufzeitumgebung *tr3.js* zur Verfügung gestellt, die eine *tr3*-Datei direkt einlesen und verarbeiten kann. Nach Einbinden der Datei kann die Verarbeitung des Baums mit einem Befehl gestartet werden:

```
tr3.init('pfad/zum/baum.xml');
```

Dies bewirkt, dass nun beginnend bei der Wurzel des Baums ein sog. *Screen* für den aktuellen Knoten gerendert wird, der dem Nutzer Frage und Auswahlmöglichkeiten präsentiert und bei einer Auswahl zum nächsten passenden *Screen* springt. Links und Wiederverwendungen werden automatisch aufgelöst bzw. verarbeitet und chronologisches Zurückspringen ist möglich. Verknüpfter Content wird automatisch nachgeladen und je nach Knotenklasse sofort oder nach Klick angezeigt.

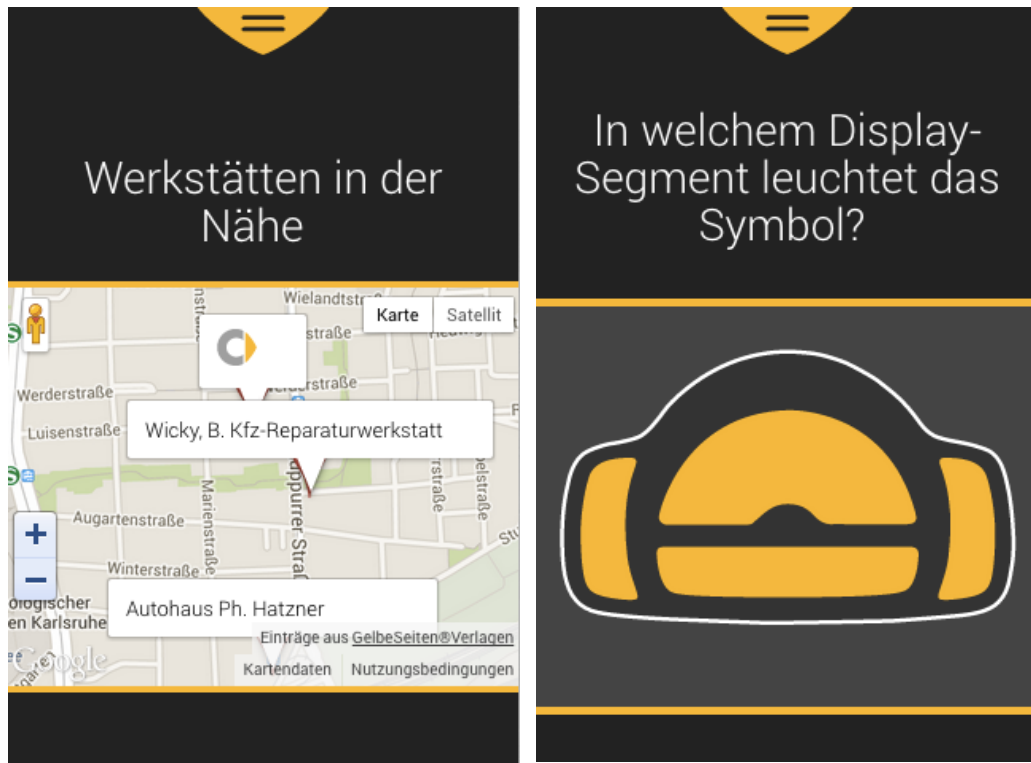


Alle automatisch generierten Elemente haben zugewiesene CSS-Klassen, mit deren Hilfe sich das Aussehen der Anwendung individuell gestalten lässt. *tr3*-Knotenklassen werden zum DOM durchgeschleust damit z.B. verschieden eingestufte Diagnosen unterschiedlich ausgezeichnet werden können (siehe Beispiel oben).

tr3-Dateien müssen für die Anwendung in der Laufzeitumgebung nicht vorverarbeitet werden. Links und nicht aufgelöste Wiederverwendungen (Kopien) werden gleich interpretiert (beide wie Links), da sie sich auf den Endanwender gleich auswirken.

Verknüpfter Content

Verknüpfter Content kann auch dazu benutzt werden den Entscheidungsbaum mit Spezialanwendungen zu erweitern. Dabei werden zusätzliche Script- und Styling-Anweisungen in die HTML-Datei des verknüpften Contents eingebettet und dieser mit einer speziellen Knotenklassen (z.B. *app*) ausgezeichnet um eine andere Darstellung zu erzielen.



Dieser verknüpfte Content wird bei Einsatz der tr3.js-Laufzeitumgebung zunächst über einen Dateinamen referenziert und dann im Moment der Anzeige des entsprechenden Knotens als HTML-Datei dynamisch nachgeladen (via AJAX-Request).

Der Export des Contents aus dem CMS kann bei der Veröffentlichung automatisch angestoßen werden (siehe Grafik: *Publishing (aus GraphML)*).

Fazit

Neues Format

Die Untersuchung bisher bestehender XML-Formate zur Abbildung von Graphen kam zu dem Ergebnis, dass bisherige Lösungen nicht den geforderten Anforderungen für den Einsatz in (mobilen) Web-Applikationen genügen. Diese Arbeit hat mit der Spezifikation des tr3-Formates diese Lücke gefüllt. Dabei ist kein zusätzlicher Standard²⁴ geschaffen worden, sondern ein Abstraktionsschicht bestehender Formate, die auch weiterhin zur Erfüllung anderer Zwecke verwendet werden können (GraphML zur Erstellung, RDF zum Austausch).

Passendes Framework

Mit dem tr3-Format als zentralen Informationsträger kann ein Framework bereitgestellt werden, das die nötigen Werkzeuge zur Entwicklung von auf XML-Entscheidungsbäumen basierten Applikationen bereitstellt: Möglichkeiten zum Import und Export, eine Testumgebung (tr3x.js) sowie eine Laufzeitumgebung (tr3.js). Durch die Verknüpfung von HTML-basiertem externen Content können entstandene Applikationen beliebig erweitert werden. Mit Hilfe dieser Mittel wird Redaktionen das Erstellen und Pflegen von Bäumen erleichtert und die Anzahl potentieller Fehlerquellen (z.B. manuelles Übertragen) verringert.

Neue Erfahrungen

Während der Arbeit am vorliegenden Projekt konnten viele neue Erfahrungen aus den Bereichen der Baum- und Graphentheorie gesammelt werden. Viele Formatspezifikationen wurden auf Ihre Ansätze hin untersucht, unterschiedliche Arten der Datenvisualisierung ausprobiert und die Verarbeitung und Konvertierung von XML-basierten Dateiformaten realisiert. Dabei konnte auf die bewährten Standards XML (Datenspeicherung), XSLT (Transformation) und JavaScript (Skripting) zurückgegriffen werden, aber auch neue Technologien wie SVG (Visualisierung), D3 (SVG-Verarbeitung) oder Shell-Scripting (Automatisierung) ausprobiert werden.

²⁴ Zur Erschaffung neuer Standards siehe: <http://xkcd.com/927/>

Ausblick

Offener Test

Nach internen Testläufen mit Entscheidungsbäumen verschiedener Größe und verknüpften Content-Arten muss sich die entstandene Lösungen offenen Tests stellen, die auf Mögliches Verbesserungspotential hinweisen.

Tracking

Um das Nutzerverhalten analysieren zu können und somit auch mögliche Hinweise auf (Bedien-)Probleme am Produkt selbst zu bekommen, kann ein Tracking integriert werden, das den vom Nutzer gewählten Pfad zu einer Diagnose aufzeichnet und in einer Auswertungsumgebung über alle Anwender hinweg vergleicht. Das Tracking kann auf der schon bestehenden *Zurück*-Funktionalität der Laufzeitumgebung aufbauen, die den bisher zurückgelegten Pfad eines Nutzers aufzeichnet²⁵ um ihm die Möglichkeit zu geben zurückzuspringen. In seiner jetzigen Implementierung wird dieser vom Benutzer zurückgelegte Pfad allerdings am Ende eines Durchlaufs verworfen. Dieser könnte zurück an den Server des Herstellers übertragen werden und somit ein Tracking ermöglichen.

Referenzierungsarten

In seiner jetzigen Form bietet das tr3-Format zwei verschiedene Arten der Referenzierung schon bestehender Knoten oder Teilbäume: Verlinkung und Kopie. Unterschiedliche Auswirkungen haben die unterschiedlichen Arten allerdings nur beim Untersuchen und Testen des Baums. Dort ist die Kopie von Teilbäumen ist dann sinnvoll, wenn Pfadlängen und Wiederverwendungsgrad in einer tr3x-Testumgebung untersucht werden sollen. Für den Endbenutzer wirken sich die beiden Referenzierungsarten nicht unterschiedlich aus (was an der Interpretation der Laufzeitumgebung liegt).

Auf Erstellungs- und Anwenderseite sind Links einfacher und effizienter zu handhaben, da Kopien nicht rekursiv sein dürfen (d.h. Referenzknoten dürfen nicht Teilbäume kopieren, denen sie selbst angehören) und einen Baum in seiner Dateigröße und visuellen Repräsentation um ein vielfaches vergrößern. Das Ziel zukünftiger Implementierungen soll sein, die verschiedenen Referenzierungsarten zu vereinheitlichen und dem Anwendungsfall entsprechend zu interpretieren.

²⁵ Indem der momentan vor ihm liegenden Teilbaum in das Array `tr3.history` gepusht wird

Anhang

Ordnerstruktur

Im Folgenden ist die Ordnerstruktur des Projektes mit Kommentaren aufgelistet:

Unterteilung:

Name	Kommentare	Größe	Art
▼ Backend	Erstellung und Testing	7,5 MB	Ordner
▼ cmd	Automatisierte Prozesse	23 KB	Ordner
graphml.buildTest.command	Testumgebung aufrufen (GraphML)	1 KB	Terminal-Shell-Skript
graphml.publishToApp.command	Publizieren und App starten (GraphML)	868 Byte	Terminal-Shell-Skript
▼ other	Weitere Prozesse	8 KB	Ordner
graphml.import.command	GraphML in tr3 konvertieren	274 Byte	Terminal-Shell-Skript
tr3.buildTest.command	Testumgebung aufrufen (tr3)	983 Byte	Terminal-Shell-Skript
tr3.buildVisualisation.command	Exp. Testumgebung aufrufen (tr3)	412 Byte	Terminal-Shell-Skript
tr3.exportRDF.command	tr3 in RDF konvertieren	254 Byte	Terminal-Shell-Skript
tr3.preprocessDefault.command	tr3 vorverarbeiten	389 Byte	Terminal-Shell-Skript
▶ edit	Arbeitsordner für Automatisierung	320 KB	Ordner
▼ tr3	Framework	6,4 MB	Ordner
▼ examples	Beispiele in verschiedenen Formaten	546 KB	Ordner
graphml	Beispiele im GraphML-Format	419 KB	Ordner
rdf	Beispiele im RDF-Format	84 KB	Ordner
tgf	Beispiele im tgf-Format	307 Byte	Ordner
tr3	Beispiele im tr3-Format	21 KB	Ordner
▼ schema	Dokumentschema für das tr3-Format	302 KB	Ordner
tr3.dtd	Dokumentschema als DTD	294 KB	Sublime Text 2.app-Dokument
tr3.xsd	Dokumentschema als XSD	2 KB	XML
▼ xsl	Stylesheets	5,5 MB	Ordner
export.rdf.xsl	tr3 -> RDF	4 KB	XSL
import.graphml.xsl	GraphML -> tr3	7 KB	XSL
import.tgf.xsl	tgf -> tr3	4 KB	XSL
tr3.layouting.xsl	tr3.pp -> tr3.layout	3 KB	XSL
tr3.preprocessing.xsl	tr3 -> tr3.pp	6 KB	XSL
tr3.rendering.xsl	tr3.layout -> html/svg (tr3x)	5 KB	XSL
transform	XSL-Transformator (Saxon)	5,5 MB	Ordner
▼ tr3x	Testumgebung	722 KB	Ordner
app	Bausteine der Anwendung	8 KB	Ordner
content	Simulierte CMS-Ablage	172 KB	Ordner
gui	Oberfläche der Anwendung	90 KB	Ordner
index.html	Generierte tr3x-Basisdatei	128 KB	HTML document
▼ script	Anwendungs-Skripte	297 KB	Ordner
exp	Experimentelle Skripte	21 KB	Ordner
jxon.js	Umwandlung von XML in JSON	2 KB	JavaScript
tree.js	Erzeugt exp. Baumvisualisierungen	13 KB	JavaScript
lib	Bibliotheken & Frameworks	249 KB	Ordner
tr3x.js	Anwendungslogik Testumgebung	20 KB	JavaScript
vis	Experimentelle Visualisierung	12 KB	Ordner
▶ Frontend	App-Frontend	945 KB	Ordner

Das Projekt wurde in seine logischen Bestandteile gegliedert. Im Bereich des Backends findet sich ein Prozessverzeichnis (`cmd`), ein Arbeitsverzeichnis (`edit`), ein Verzeichnis mit Dateien zum Format und dessen Umwandlungen (`tr3`) und eine Verzeichnis für die Testumgebung (`tr3x`). Die Laufzeitumgebung `tr3.js` befindet sich im Ordner `/Frontend/js`.

Hinweis: Alle Dateien, die sich im Ordner `Frontend` befinden, sind im Rahmen des Projekts ‚smart help – Geführte Pannenhilfe‘ der Vorlesung *Media Engineering* bei Prof. Schöber entstanden und nicht mehr direkt dem eigentlichen Projektumfang zuzurechnen. Sie wurden beigelegt, um den ganzheitlichen Ansatz des Konzepts zu demonstrieren.

Systemanforderungen

Darstellung: Moderner Webbrowser (MS IE 9+, Moz. Firefox 26+, Google Chrome 31+)

Bearbeitung: yWorks yED 3.1+ (frei verfügbar)

Transformation: XSLT-2.0-fähiger XSL-Prozessor (Saxon 9 HE mitgeliefert)

Workflows: Shell (Unix-basiertes Betriebssystem), Python 2.x (frei verfügbar), OSA

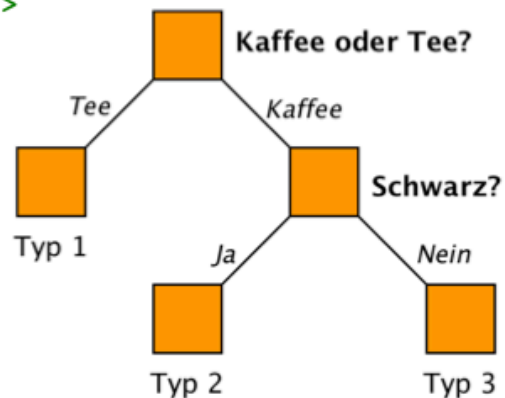
Getestet unter OS X 10.9, Google Chrome 32

tr3: Grundlagen

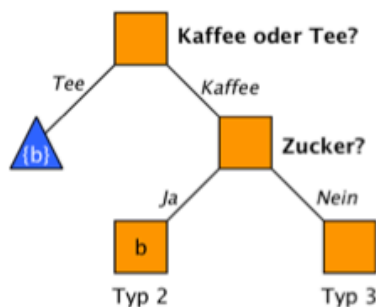
```
<tr3 out="Kaffee oder Tee?" ...>
  <branch in="Kaffee" out="Schwarz?">
    <leaf in="Ja" out="Typ 2" />
    <leaf in="Nein" out="Typ 3" />
  </branch>
  <leaf in="Tee" out="Typ 1" />
</tr3>
```

zusätzliche Attribute:

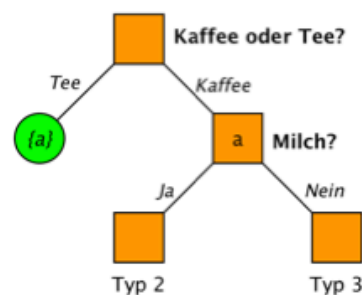
```
id="f67"
content="lorem.html"
class="danger"
```



tr3: Wiederverwendung



```
<tr3 out="Kaffee oder Tee?" ...>
  <branch in="Kaffee" out="Zucker?">
    <leaf in="Ja" out="Typ 2" id="b" />
    <leaf in="Nein" out="Typ 3" />
  </branch>
  <ref in="Tee" link="b" />
</tr3>
```



```
<tr3 out="Kaffee oder Tee?" ...>
  <branch in="Kaffee" out="Zucker?" id="a">
    <leaf in="Ja" out="Typ 2" />
    <leaf in="Nein" out="Typ 3" />
  </branch>
  <ref in="Tee" copy="a" />
</tr3>
```

Quellen

BOSTOCK, Mike (2012): Working with Transitions (D3).

[<http://bost.ocks.org/mike/transition/>](http://bost.ocks.org/mike/transition/)

BOSTOCK, Mike (2013): How Selections Work (D3). *(sehr zu empfehlen)*

[<http://bost.ocks.org/mike/selection/>](http://bost.ocks.org/mike/selection/)

BRANDES, Ulrik et al (2010): „Graph Markup Language (GraphML)“.

In: TAMASSIA, Roberto (Hrsg): Handbook of Graph Drawing and Visualization.
London : Chapman & Hall

[<http://kops.ub.uni-konstanz.de/bitstream/handle/urn:nbn:de:bsz:352-244261/Brandes_244261.pdf>](http://kops.ub.uni-konstanz.de/bitstream/handle/urn:nbn:de:bsz:352-244261/Brandes_244261.pdf)

BRANDES, Ulrik / LERNER, Jürgen / PICH, Christian (2004): „GXL to GraphML and Vice Versa with XSLT“. In: Proceedings of the 2nd International Workshop Graph-Based Tools (GraBaTs ,04), 113-125

[<http://kops.ub.uni-konstanz.de/bitstream/handle/urn:nbn:de:bsz:352-opus-81185/GXL_to_GraphML_and_Vice_Versa_with_XSLT.pdf>](http://kops.ub.uni-konstanz.de/bitstream/handle/urn:nbn:de:bsz:352-opus-81185/GXL_to_GraphML_and_Vice_Versa_with_XSLT.pdf)

DREWER, Petra / ZIEGLER, Wolfgang (2011): Technische Dokumentation.

Würzburg : Vogel

DUCHARME, Bob (2001): XSLT quickly. Greenwich, CT : Manning

DUCARME, Bob (2003): „XSLT 2 and Delimited Lists“. In: O'REILLY (Hrsg) XML.com

[<http://www.xml.com/pub/a/2003/05/07/tr.html>](http://www.xml.com/pub/a/2003/05/07/tr.html)

GALLES, David (2011): Data Structure Visualizations. Computer Science Dept.
University of San Francisco. *(sehr zu empfehlen)*

[<http://www.cs.usfca.edu/~galles/visualization/Algorithms.html>](http://www.cs.usfca.edu/~galles/visualization/Algorithms.html)

HAROLD, Elliotte Rusty / MEANS, W. Scott (2003): XML in a Nutshell.

Deutsche Ausgabe. 3. Auflage. Köln : O'Reilly

HAUSER, Tobias (2010): XML Standards. 2. Auflage.

Frankfurt/Main : Entwickler.Press

KOSEK, Jirka (2004): „Automated Tree Drawing: XSLT and SVG“.

In: O'REILLY (Hrsg) XML.com *(sehr zu empfehlen)*

[<http://www.xml.com/pub/a/2004/09/08/tree.html>](http://www.xml.com/pub/a/2004/09/08/tree.html)

MOZILLA DEVELOPER NETWORK (2012): JXON

[<https://developer.mozilla.org/en-US/docs/JXON>](https://developer.mozilla.org/en-US/docs/JXON)

OMG (2010): BPMN 2.0 by Example

[<http://www.omg.org/spec/BPMN/20100601/10-06-02.pdf>](http://www.omg.org/spec/BPMN/20100601/10-06-02.pdf)

OMG (2011): Business Process Model and Notation (BPMN) - Version 2.0.

[<http://www.omg.org/spec/BPMN/2.0/PDF/>](http://www.omg.org/spec/BPMN/2.0/PDF/)

RESIG, John (2011): Dictionary Lookups in JavaScript.

[<http://ejohn.org/blog/dictionary-lookups-in-javascript/>](http://ejohn.org/blog/dictionary-lookups-in-javascript/)

SKULSCHUS, Marco / WIEDERSTEIN, Marcus (2005): XSLT und XPath für HTML, Text und XML. Bonn : mitp

W3C (1997): Resource Description Framework (RDF) - Model and Syntax

[<http://www.w3.org/TR/WD-rdf-syntax-971002/>](http://www.w3.org/TR/WD-rdf-syntax-971002/)

W3C (2004): RDF Primer - W3C Recommendation

[<http://www.w3.org/TR/rdf-primer/>](http://www.w3.org/TR/rdf-primer/)

WOLF, Lina (2003): Zeichnen von Bäumen. Seminararbeit an der FU Berlin.

[<http://linawolf.de/fileadmin/user_upload/papers/zeichnenVonBaeumen.pdf>](http://linawolf.de/fileadmin/user_upload/papers/zeichnenVonBaeumen.pdf)

YWORKS: yED Graph Editor Manual – File Formats

[<http://yed.yworks.com/support/manual/fileformat.html>](http://yed.yworks.com/support/manual/fileformat.html)

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig angefertigt, keine anderen als die angegebenen Mittel benutzt und alle wörtlichen oder sinngemäßen Entlehnungen als solche gekennzeichnet habe.

Karlsruhe, 7. Februar 2014