TECHNISCHE UNIVERSITÄT CHEMNITZ

Fakultät für Informatik

Professur Verteilte und selbstorganisierende Rechnersysteme

Bachelorarbeit

Analyse von existierenden Widget-Formaten und Entwicklung eines Verfahrens zu deren Transformation in das W3C-Format

Michael Hertel

Chemnitz, den 4. September 2012

Prüfer: Prof. Dr.-Ing. Martin Gaedke
Betreuer: Dipl.-Inf. Olexiy Chudnovskyy

Hertel, Michael
Analyse von existierenden Widget-Formaten und Entwicklung eines Verfahrens zu deren Transformation in das W3C-Format
Bachelorarbeit, Fakultät für Informatik
Technische Universität Chemnitz, September 2012

Zusammenfassung

Nach W3C-Standard sind Widgets eigenständige, klientenseitige Applikationen, welche auf Internettechnologien basieren. Im Internet gibt es mehrere Portale, in denen Widgets veröffentlicht und angeboten werden. Dabei sind die Portale meist an eine spezielle Widget-Engine gebunden und besitzen jeweils ihre eigene Widget-Implementierung. Durch die unterschiedlichen Formate ist die Portabilität der Widgets sehr eingeschränkt.

Im September 2011 hat das World Wide Web Consortium (W3C) eine Empfehlung für den Standard des Widget-Formates herausgegeben. Solange die Anzahl der Widgets nach der W3C-Norm, im Gegensatz zur Anzahl der Widgets in anderen Formaten, noch gering ist, wird auch die Unterstützung dieser begrenzt sein. Durch ein großes Widget-Angebot im W3C-Format könnte sich dieses jedoch gegen die bisherigen Formate durchsetzen.

Um das bestehende Angebot von W3C-Widgets zu erweitern, sollen mit dieser Arbeit Möglichkeiten zur Erzeugung dieser untersucht werden. Dazu wird unter anderem analysiert, ob und inwiefern sich die existierenden Widgets, die in proprietären Formaten vorliegen, in das W3C-Format umwandeln lassen. Die Konvertierung in Widgets nach dem W3C-Standard soll mittels einer Beispielapplikation automatisiert werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CDATA	Character Data
CSS	Cascading Style Sheets
DOM	Document Object Model
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IP	Internetprotokoll
IRI	Internationalized Resource Identifier
JSON	JavaScript Object Notation
LZX	Laszlo
MDD	Model Driven Development
MIME	Multipurpose Internet Mail Extensions
REST	Representational State Transfer
RFC	Requests for Comments
RIA	Rich Internet Application
UA	User Agent
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UWA	Universal Widget API
W3C	World Wide Web Consortium
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language

1. Einleitung

Der Begriff Widget ist im informatischen Kontext ein Mischwort aus den englischen Worten „window“ und „gadget“ [18]. Die Kombination soll darauf hindeuten, dass es sich um eine kleine, nützliche Anwendung handelt, welche in einem Fenster dargestellt wird. Das Wort tritt im informatischen Bereich in unterschiedlichen Zusammenhängen mit jeweils unterschiedlichen Bedeutungen auf, was häufig zu Verwirrungen führt.

Seinen Ursprung hat es jedoch nicht in der Informatik. Die erste bekannte Verwendung des Begriffs Widget stammt aus dem Jahr 1924. Es wurde in dem Theaterstück „Beggar on Horseback“ als Wort für ein kleines, maschinell erzeugtes und bedeutungsloses Objekt, welches in Massen für den Normalverbrauch gefertigt wird, verwendet. [8, 19]

Über die Zeit hinweg wurde das Wort in den normalen englischen Sprachgebrauch aufgenommen. Es ordnet sich in den ökonomischen bzw. wirtschaftlichen Kontext ein. Der Begriff kann in diesem Zusammenhang zwei Bedeutungen haben. Zum einen kann Widget einen Platzhalter für ein mechanisches Produkt, von welchem eine präzise Angabe zur Identität nicht so wichtig ist, darstellen. Außerdem kann es als Bezeichner für ein hypothetisches Produkt, welches eventuell später einmal erfunden oder aktuell in der Entwicklung ist, auftreten, welchem sonst kein Name zugeordnet werden kann. Im umgangssprachlichen englischen Sprachgebrauch findet es eine ähnliche Anwendung, wie das deutsche Wort „Dingsbums“ und bezeichnet einen Gegenstand dessen Name unbekannt ist [18]. [8, 19]

In den informatischen Kontext wurde der Begriff Widget im Jahre 1988 durch das Projekt Athena eingeführt. Athena war ein achtjähriges Projekt vom Massachusetts Institute of Technology (MIT), IBM und Digital Equipment Corporation (DEC). Es diente der Entwicklung eines Systems zur innovativen Integration von Computertechnologie in den Lehrplan der Universität. Die Ergebnisse des Projektes waren u.a. das Authentifizierungsprotokoll Kerberos und das X-Window-System. Zum X-Window-System wurde auch das X-Toolkit eingeführt, welches der möglichst einfachen Entwicklung einer grafischen Benutzerschnittstelle dient. Innerhalb des Toolkits wurden die Komponenten zum Bauen der Nutzeroberfläche als Widget bezeichnet. [20, 24]

Seit der Einführung in das informatische Vokabular hat sich der Begriff Widget im Bereich der Software schon weit verbreitet. Das Wort nimmt mittlerweile je nach Zusammenhang unterschiedliche Bedeutungen an, weshalb eine Unterscheidung der Widgets

in verschiedene Unterarten nötig ist. Im Bereich der Desktop-Software-Entwicklung bezeichnet Widget einen durch das Betriebssystem vorgefertigten Baustein für die Benutzerschnittstelle des Programms. Solche Vorgaben sind zum Beispiel Menüs oder Buttons, die dem Entwickler mittels einer Bibliothek zur Verfügung gestellt werden. Sie entsprechen damit der ursprünglichen Bedeutung bei der Einführung innerhalb des Athena Projektes. [19]

Im Bereich des Webs gibt das World Wide Web Consortium (W3C) eine Definition für Widgets vor. Diese sagt aus, dass Widgets eigenständige, klientenseitige Applikationen sind, welche auf Internettechnologien basieren und für die Verteilung zu einem Packet gebündelt sind [5]. Diese Form der Widgets könnte man auch als Widgets für das Web bezeichnen. Sie treten zum Beispiel bei Plattformen wie iGoogle und Netvibes auf. Auch bei vielen Betriebssystemen wird diese Form der Widgets genutzt. Beispiele dafür sind die Microsoft Gadget Sidebar für Windows[1], das Apple Dashboard[2] für Mac oder gDesklets[3] für Linux- und Unix-Betriebssysteme mit einer Gnome Desktop-Umgebung. Häufig werden diese Widgets allerdings nicht als „Widgets“ oder „Web Widgets“, sondern als „Gadgets“ bezeichnet, da sich Apple den Begriff „web widgets“ lizenzrechtlich schützen lassen hat. Apple bezeichnet mit „web widgets“ eine Software, mit deren Hilfe andere internet- und webbasierte Software entwickelt werden kann [33].

Widgets, im Sinne von kleinen Applikationen, welche nützliche Funktionalitäten zur Verfügung stellen, sind keine neue Erfindung. Sie traten zum Beispiel schon 1984 im Mac-Betriebssystem von Apple auf. Das zu dieser Zeit neue Betriebssystem bot kleine Miniaturapplikationen an, welche in einem eigenen Fenster dargestellt wurden. Der Begriff Widget existierte zu dieser Zeit im informatischen Kontext noch nicht. Apple gab den Minianwendungen den Namen Desk Accessories. Beispiele für Desk Accessories sind Taschenrechner, Kalender, Uhr, Notizzettel und Spiele. Diese fünf Beispiele zählen auch heutzutage zu den bekanntesten Vertretern für Widgets. [20]

Im Internet tritt der Begriff Widget häufig für eine weitere Art dieser auf. Es handelt sich dabei, ähnlich zu den W3C-Widgets, um Applikationen, welche auf Internettechnologien basieren und beim Klienten zur Ausführung kommen. Allerdings unterscheiden sie sich zur W3C-Definition, indem sie nicht als Paket auftreten, sondern durch Kopieren eines HTML-Codestückes (Hypertext Markup Language) in eine Seite eingefügt werden. Das Erstellen dieser Widgets findet meist durch einen Generator statt, welcher vom Nutzer Voreinstellungen für die Konfiguration der Applikation erhält. Ein Beispiel für einen solchen Generator bietet die Firma Flite mit Widgetbox[4] an.

[1] http://windows.microsoft.com/de-de/windows-vista/Windows-Sidebar-and-gadgets-overview
[2] http://www.apple.com/downloads/dashboard/
[3] http://www.gdesklets.de/
[4] http://www.widgetbox.com/

Zusammenfassend lässt sich feststellen, dass das Wort Widget vor allem im Internet für vieles als eine Art Modewort Verwendung findet. Diese Arbeit beschäftigt sich nur mit Widgets, welche konform zu der Definition vom W3C ist. Deshalb ist im Folgenden das Wort Widget stets mit der Begriffserklärung der W3C-Spezifikation gleichzusetzen.

1.1. Motivation

Für Widgets gibt es im Internet mehrere Portale. Einige der bekanntesten Vertreter sind iGoogle, Netvibes, Opera, Yahoo. Jeder Anbieter hat meist sein eigenes Speicherformat für die Widgets und seine eigene Engine für die Darstellung dieser. Dadurch ist der Nutzer immer gezwungen die Widgets vom Portal des Providers zu beziehen, von welchem er die Engine nutzt. Dies schränkt die Auswahl an Widgets für den Anwender ein. Falls ein Benutzer zwei Lieblings-Widgets von jeweils zwei unterschiedlichen Portalen hat, ist es ihm meist nicht möglich beide zugleich zu benutzen. Auch für Widget-Entwickler haben die unterschiedlichen Formate und Engines einen großen Nachteil. Um ein Widget bei mehreren Portalen zur Verfügung zu stellen, muss dieses aufgrund der nicht vorhandenen Portabilität für jedes Format neu geschrieben oder aufwendig angepasst werden. Dies ist jedes Mal mit viel Zeitaufwand und dadurch auch mit höheren Kosten im kommerziellen Bereich verbunden. Deshalb lohnt sich eine Portierung von Widgets meist nur für die bekanntesten Portale, da bei diesen eine höhere Nutzerzahl zu erwarten ist. Das Interesse an kleineren Widget-Anbietern hingegen ist aufgrund der geringeren zu erwartenden Anzahl an Benutzern oft nicht sehr hoch. Dies wirkt sich auch auf die Portale mit wenigen Nutzern negativ aus, da sie kein großes Angebot vorweisen können, was neue Anwender anlocken würde.

Um dem Problem der Portabilität und Kompatibilität entgegenzuwirken arbeitet das W3C seit 2006 an einer Standardisierung des Widget-Formates. Am 27. September 2011 wurde die Standardisierung mit dem Titel „Widget Packaging and XML Configuration“[5] zur Empfehlung für das Widget-Format. Die Umstellung aller vorhandener Engines und Portale auf das W3C-Format hätte eine Abschaffung der Portabilitäts- und Kompatibilitätsprobleme zur Folge. Allerdings ist die Umstellung eines großen Portals von geringer Wahrscheinlichkeit, da es für den Provider sehr hohe Kosten und einen großen Zeitaufwand zur Folge hätte. Es müssten nicht nur Änderungen an der Engine und dem entsprechenden Portal stattfinden, sondern es wäre auch eine Konvertierung aller Widgets des Anbieters von Nöten. Deshalb ist der Einsatz des W3C-Formates in nächster Zeit nur durch neu entwickelte Widget-Umgebungen zu erwarten. Im Moment sind schon einige Widget-Container in der Entwicklung, welche W3C-Widgets nutzen können, wie zum Beispiel Apache Wookie. Bei Apache Wookie

[5] http://www.w3.org/TR/widgets/

handelt es sich um eine in Java geschriebene Serveranwendung für Widgets. Sie ermöglicht das Aufnehmen von W3C-Widgets in eine Datenbank. Mittels einer REST (Representational State Transfer) API (Application Programming Interface) werden die Widgets vom Server zur Verfügung gestellt.

Im Moment gibt es noch nicht viele Nutzer von W3C-Widgets. Einer der Gründe dafür ist, dass es noch keine weitbekannte W3C-Widget-Umgebung gibt. Viele der Umgebungen befinden sich zurzeit in der Entwicklungsphase und sind noch nicht marktreif. Ein weiterer Grund ist sicher das geringe Angebot an W3C-Widgets. Im Gegensatz zu den anderen Widget-Formaten gibt es für das W3C-Format aktuell kein Portal, welches eine Vielzahl von unterschiedlichen Widgets aus einer Datenbank anbietet. Allgemein ergibt eine Suche nach W3C-Widgets zum Download mittels Google kaum sinnvolle Ergebnisse. Bei dem Apache Wookie Paket sind einige Widgets zu Testzwecken enthalten, aber sonst ist das Angebot an Widgets im W3C-Format sehr enttäuschend. Dies ist auch einer der Hauptgründe, weshalb es kaum Benutzer der W3C-Widgets gibt. Ein Beispiel für die geringe Nutzerzahl ist der Opera Browser. Seit Opera Version 9 (2006) bot der Webbrowser eine Unterstützung für W3C-Widgets an [31]. Mit dem Release der Version 12 des Browsers am 14. Juni 2012 stellte Opera die Unterstützung offiziell ein und deaktivierte die Widget-Engine standardmäßig. Bis Ende des Jahres 2012 möchte Opera die Möglichkeit des Verwendens von W3C-Widgets durch den Browser komplett entfernen [34]. Einer der genannten Gründe von Opera ist die geringe Nutzerzahl, was sicher auch vom geringen Angebot an W3C-Widgets abhängig ist. Eine Unterstützung durch häufiger genutzte Browser oder direkt durch Betriebssysteme könnte zwar viele Menschen auf Widgets im W3C-Format aufmerksam machen, solange aber noch nicht genug W3C-Widgets existieren, ergibt der Aufwand wenig Sinn.

1.2. Zielsetzung

Im vorherigen Kapitel fand eine Betrachtung der Probleme für die geringe Verwendung des standardisierten W3C-Widget-Formates statt. Dabei ergaben sich unter anderem das geringe Angebot und die geringe Nutzerzahl der Widgets im W3C-Format als zwei der Hauptprobleme. Um den zwei Problemen entgegenzuwirken, erfordert es Möglichkeiten zur Generierung von W3C-Widgets, damit eine größere Anzahl zur Verfügung steht. Durch ein umfangreicheres Portfolio an Widgets im W3C-Format kann sich die Beliebtheit und somit die Nutzerzahl erhöhen.

Für die Generierung ergeben sich die folgenden fünf Ansätze: das Entwickeln neuer Widgets im W3C-Format, die Nutzung der modellgetriebenen Entwicklung, die Generierung von Widgets aus bereits im Web vorhandenen JavaScript-Anwendungen,

das Erstellen von Widgets aus RSS-Feeds und das Konvertieren bestehender Widgets in unterschiedlichen Formaten in das W3C-Format. Diese fünf Verfahren werden im Abschnitt 2.2 genauer betrachtet.

Um das Widget-Angebot im W3C-Format zu vergrößern, soll in dieser Arbeit das Verfahren der Konvertierung zum Einsatz kommen, da dieses die wenigsten Nachteile aufweist und die Vorteile überwiegen. Die für die Konvertierung ausgewählten Formate sind das Opera-, das UWA- und das iGoogle-Format. Zunächst soll eine Betrachtung der drei verschiedenen Formate im Vergleich zum W3C-Format stattfinden. Aus den ermittelten Unterschieden sollen die Schritte für eine Konvertierung abgeleitet werden. Im Anschluss folgt ein Test auf die Möglichkeit der Automatisierung des Umwandlungsvorganges durch eine Beispiel-Implementierung. Ein wichtiges Kriterium ist vor allem eine geringstmögliche Nutzerinteraktion für das Konvertieren in das W3C-Format.

1.3. Anwendungsfälle

Für das automatisierte Konvertieren eines vorhandenen Widgets in das W3C-Format ergeben sich vor allem die folgenden zwei Anwendungsfälle: erstens die Nutzung des Umwandlungsprogrammes durch einen Autor eines Widgets und zweitens die Verwendung durch einen Anbieter von Widgets bzw. einen Portalbetreiber. Der erste Fall tritt ein, wenn ein Autor bereits Widgets für ein Format, zum Beispiel das iGoogle-Format, entwickelt hat und die mögliche Nutzerzahl erhöhen möchte. Er wandelt das Widget um und kann es somit auch in einem anderen Format anbieten. Das neu gewonnene W3C-Widget kann er im Anschluss auf Portalen, die das W3C-Format nutzen, zur Verfügung stellen. Beim zweiten Fall möchte ein W3C-Widget-Anbieter sein Angebot möglichst schnell und unkompliziert vergrößern, um somit auch das Nutzerinteresse an seinem Widget-Store zu erhöhen. Dazu kann er frei verfügbare Widgets umwandeln und danach auf seinem Portal anbieten.

1.4. Aufbau der Arbeit

Im zweiten Kapitel der Arbeit sollen zuerst eine Betrachtung und Bewertung bereits vorhandener Möglichkeiten zur Generierung von W3C-Widgets stattfinden. Im Anschluss folgt das Kapitel drei zum Entwurf der Konvertierung. Um W3C-Widgets aus anderen Widget-Formaten zu genieren, werden zunächst die Grundbestandteile eines Widgets analysiert. Eine Erläuterung des Aufbaus eines W3C-Widgets in Bezug auf die erklärten Grundbestandteile befindet sich im Anhang A. Anschließend

folgt eine Aufzählung der für die Konvertierung ausgesuchten Widget-Formate mit einer Begründung für deren Auswahl. Aus den Unterschieden zwischen den ausgewählten Formaten zum W3C-Format wird im Folgenden ein Grundkonzept für die Umwandlung abgeleitet. Danach schließt sich das Kapitel vier mit einigen Details zur Konvertierung an. Im fünften Kapitel folgt eine Bewertung des Prototyps des Umwandlungsprogrammes. Dazu werden die konvertierten Widgets mittels Apache Wookie überprüft und die Ergebnisse analysiert. Zuletzt schließen sich mit Kapitel sechs eine Zusammenfassung der Ergebnisse dieser Arbeit und ein Ausblick an. Im Ausblick findet eine Betrachtung der Möglichkeiten zur Erweiterung des Widget-Formatwandlers statt.

2. Stand der Technik

Im Kapitel zum Stand der Technik werden aktuelle Lösungsansätze ausgehend von den in Abschnitt 1.2 genannten Möglichkeiten, außer der Neuentwicklung, zur Widget-Generierung betrachtet. Zunächst findet eine Analyse der Anforderungen für das automatisierte Konvertieren statt. Es schließen sich eine Beschreibung aller genannter Verfahren, eine Betrachtung aktueller Lösungen und deren Bewertung anhand der Anforderungen an.

2.1. Anforderungen

Die vier grundlegenden Anforderungen an eine Generierungsvariante sind Angebot, Aufwand, Anpassung und Verwendbarkeit. Eine der wichtigsten Kriterien ist das Angebot an Eingabeapplikationen. Es sollte eine gewisse Menge an Anwendungen im entsprechenden Format vorliegen, damit sich das Schreiben eines Umwandlungsprogrammes lohnt. Ausschlaggebend für diese Anforderung ist weiterhin, ob die entsprechenden Eingabedaten gebündelt vorliegen. Wenn es für das gewählte Format ein Portal mit einer großen Menge an Applikationen gibt, sodass sie ein Nutzer einfach und schnell zusammensuchen kann, wirkt sich dies positiv auf das Kriterium Angebot aus. Wenn der Benutzer hingegen nach jeder Anwendung für die Konvertierung einzeln und aufwendig suchen muss, ist der Aufwand pro W3C-Widget entsprechend hoch. Die Anforderung Aufwand repräsentiert die Anzahl der Nutzerinteraktionen, die nötig sind um W3C-Widgets zu erzeugen. Am besten ist eine Lösung, die automatisch eine größere Anzahl von Eingabeanwendungen am Stück umwandelt, ohne den Nutzer einzubeziehen. Das Kriterium Angepasstheit stellt dar, wie angepasst die entsprechenden Applikationen an das Umfeld einer Widget-Umgebung sind. Dies betrifft vor allem die Benutzeroberfläche. Bieten die Eingabeprogramme jeweils nur eine sehr umfangreiche Benutzeroberfläche an, die für eine Darstellung auf einer sehr großen Fläche gedacht ist, so sind sie weniger für eine Widget-Umgebung geeignet, da sich häufig mehrere Widgets die Darstellungsfläche teilen. Weiterhin sind verwendete Funktionalitäten ausschlaggebend, welche bei einer W3C-Widget-Umgebung eventuell nicht zur Verfügung stehen bzw. nicht realisierbar sind. Die letzte Anforderung ist die Verwendbarkeit und behandelt, ob das entsprechende Widget-Generierungsprogramm sofort für das W3C-Format einsetzbar ist oder zuvor Modifikationen nötig sind.

2.2. Möglichkeiten zur Generierung von W3C-Widgets

2.2.1. Entwicklung neuer Widgets

Bei der Entwicklung neuer Widgets handelt es sich um das Erstellen eines Widgets direkt im W3C-Format. Der Programmierer greift eine Idee für die Funktionalitäten eines Widgets auf und designt es passend zum W3C-Widget-Format. Er entwickelt das neue Widget primär oder alleinig für die W3C-Widget-Umgebung. Der Vorteil dieser Methode ist, dass das Widget perfekt an die Umgebung angepasst ist. Es können alle Details der Umgebung beachtet und alle gegebenen API-Funktionen ausgenutzt werden. Allerdings hat diese Methode auch Nachteile. Die Entwicklung eines Widgets kann sehr zeitaufwendig und somit sehr teuer sein. Wenn man eine hohe Nutzerzahl für das neu entwickelte Widget erwarten kann, relativieren sich Zeitaufwand und Kosten selbstverständlich wieder in Bezug auf die Nutzungszeit. Für W3C-Widgets ist jedoch in kurz- und mittelfristiger Zeit eher eine geringere Anzahl an Verwendern zu erwarten. Gerade im kommerziellen Bereich ist eine hohe Nutzeranzahl unabdingbar, weshalb eine Neuentwicklung eines Widgets im W3C-Format wenig Sinn ergibt und bekannte Widget-Formate und Portale vorgezogen werden.

2.2.2. Modellgetriebene Entwicklung

Die zweite Möglichkeit zur Widget-Generierung nutzt den MDD-Ansatz (Model Driven Development). Bei der modellgetriebenen Entwicklung entwirft der Softwareentwickler formale Modelle, aus welchen später automatisiert lauffähige Software erstellt wird. Das Ziel von MDD ist ein hoher Grad an Abstraktion. Vorteile des modellgetriebenen Entwicklungsansatzes ist die höhere Entwicklungsgeschwindigkeit durch den abstrakten Ansatz und die leichtere Verständlichkeit mittls formaler Modelle. Es gibt allerdings noch einen sehr viel wichtigeren Vorteil für die Verwendung mit Widgets. Durch das formale Modell wird eine Art Metabeschreibung zur Verfügung gestellt, welche dann automatisiert in eine konkrete Beschreibung umgewandelt werden kann, wodurch das lauffähige Programm entsteht. In Bezug auf die Widgets bedeutet das, dass die Widget-Entwicklung einmal in Form eines Meta-Widgets stattfinden muss. Im Anschluss ist ein automatisierter Export in jedes Widget-Format, welches die Entwicklungsumgebung zur Verfügung stellt, möglich. Wenn ein Metamodell für Widgets zur Verfügung steht, können die dafür entwickelten Widgets in weitverbreiteten Widget-Portalen, wie beispielsweise iGoogle, aber auch im W3C-Format verwendet werden. Dadurch ergibt sich in der Theorie eine potenzielle Nutzeranzahl, welche der Summe der Nutzerzahlen der entsprechenden unterstützten Formate entspricht.

2.2.3. Weiterverwendung vorhandener JavaScript-Anwendungen

Die dritte Möglichkeit der Widget-Erzeugung basiert auf der Weiterverwendung von bereits Vorhandenem. Im Internet gibt es Millionen von JavaScript-Anwendungen mit HTML-Benutzeroberfläche, welche für die Nutzer Funktionalitäten zur Verfügung stellen. Das Ziel ist es, diese vorhandenen Applikationen zu nutzen und daraus Widgets zu entwickeln. Die Generierung sollte natürlich möglichst automatisch ablaufen, um den Arbeitsaufwand zu minimieren. Allerdings ist eine vollständige Automatisierung nur begrenzt möglich, da für die Widgets häufig Konfigurationen, Metadaten und angepasste Benutzeroberflächen nötig sind, um diese sinnvoll einzusetzen. Ein weiteres Problem ist das Zusammensuchen der JavaScript-Anwendungen, da diese im gesamten Internet auf sehr vielen verschiedenen Internetseiten verteilt liegen.

2.2.4. Erstellen von Widgets aus RSS-Feeds

Der vierte Ansatz zum Generieren von Widgets nutzt die Vorteile des standardisierten Datenformates eines RSS-Feeds. Es müssen lediglich einmal Design, Aufbau und Funktionalität eines RSS-Feed-Reader-Widgets programmiert werden. Dazu empfiehlt es sich ein weiteres kleines Programm zu schreiben, welches als Eingabe einen Uniform Resource Locator (URL) entgegennimmt. Aus dem Feed, welcher sich an der gegebenen URL befindet, kann das Programm alle Metadaten auslesen und für den speziellen Feed ein fertiges W3C-Widget zu Verfügung stellen. Da es im Internet tausende RSS-Feeds gibt, welche zum größten Teil gut sortiert in Verzeichnissen verlinkt sind, ist das Erstellen vieler Widgets mit diesem Ansatz kein Problem. Der Nachteil des Verfahrens ist jedoch, dass die somit erstellten W3C-Widgets auf die Fähigkeiten eines Feeds beschränkt sind. Der Großteil des Ergebnisses beschränkt sich auf unterschiedlichste Nachrichten-Widgets. Interaktive Widgets können auf diese Weise nicht entstehen.

2.2.5. Konvertierung bestehender Widgets

Das Umwandeln von Widget-Formaten ist eine weitere Möglichkeit, welche auf einer Weiterverwendung basiert. Ähnlich den JavaScript-Applikationen gibt es mittlerweile auch Widgets millionenfach im Internet. Das Ziel ist es diese zu nutzen und in das W3C-Format umzuwandeln, um dadurch ein großes Angebot an Widgets zu schaffen. Bei der Verwendung von Widgets ergeben sich einige Vorteile im Gegensatz zur Nutzung von normalen JavaScript-Applikationen mit HTML-Oberfläche. Das erste Argument für die Widgets in anderen Formaten ist die bereits auf Widget-Anforderung angepasste Anwenderschnittstelle. Des Weiteren sind Konfigurationen und Metada-

ten meist schon im anderen Format vorhanden und können für das W3C-Widget weiter genutzt werden. Auch das Suchen nach Widgets ist deutlich einfacher. Meist bieten Portale tausende Widgets an, wodurch schnell eine große Menge zum Konvertieren zusammengesucht ist. Das Umwandeln von Widgets hat allerdings auch Nachteile. Probleme treten zum Beispiel auf, wenn ein Widget-Format durch eine Schnittstelle viele Funktionen anbietet, für welche kein äquivalenter Ersatz bei der W3C-Widget-Umgebung vorhanden ist.

2.2.6. Integration durch HTML-Elemente

Eine weitere Möglichkeit zur Generierung von Widgets ergibt sich aus dem Dokument mit dem Titel „Turning Web Applications into Mashup Components: Issues, Models, and Solutions“, welches innerhalb der Kongressakte des International Conference on Web Engineering (ICWE) veröffentlicht wurde. Es befasst sich mit Mashups, welche eine Webapplikation darstellen, die aus der Kombination bereits vorhandener Inhalte des Webs entstehen. Meistens erstellen derartige Anwendungen neue Informationen, die sich aus der Kombination der bereits vorhandenen Daten miteinander ergeben. Es gibt bereits viele hilfreiche Tools für das Erstellen der Mashups, wobei die Komponenten meist aus Anwendungen des Webs 2.0 bestehen. Im Dokument ist hingegen eine Möglichkeit beschrieben, bei welcher traditionell erstellte Applikationen die Komponenten der Mashups bilden. Für das Integrieren der traditionellen Webanwendungen in den Mashup kann je nach Bedarf eines der drei HTML-Elemente „div“, „span“ oder „iframe“ zum Einsatz kommen. Das Dokument beschäftigt sich im Speziellen mit der Verwendung des „div“-Elementes. Ein sogenannter Wrapper, welcher die Hülle für den Mashup bildet, lädt die Komponente in das „div“ und wendet auf dieses die CSS-Regeln der entsprechenden Applikation an. [9]

Wenn sich mehrere Widgets auf einer Seite befinden handelt es sich auch um einen Mashup. Es lassen sich somit Parallelen von Widget-Umgebungen zum im Dokument beschriebenen Verfahren erkennen. Die Widgets bilden die Komponenten des Mashups. Bei Apache Wookie sind diese beispielsweise in „iframe“-Elemente geschachtelt. Durch das Verfahren des Schachtelns lassen sich auch neue W3C-Widgets generieren. Die erste Möglichkeit besteht darin ein kleines Tool zu programmieren, welches vorhandene Webapplikationen zum Erstellen von Widgets verwendet. Dazu nimmt es URLs entgegen, welche eine Webanwendung adressieren und bindet den Inhalt des Verweises mittels „div“, „span“ oder „iframe“ in das Widget ein. Die Funktionalität ist somit vorhanden. Es fehlen nur die Metainformationen und Konfigurationsdaten, welche das Tool zum Teil aus der vorhandenen Webapplikation auslesen kann und andernfalls vom Nutzer erfragen muss. Eine weitere Möglichkeit für den Ansatz der Einbettung ist deren Verwendung zur Konvertierung vorhandener Widgets in anderen Formaten. Dazu muss sich das umzuwandelnde Widget in einem Portal befinden, wel-

ches dieses auch darstellen kann. Das zu entwickelnde Tool müsste ein W3C-Widget in Form einer Hülle erstellen, welches die Darstellung des Quell-Widgets vom Portal mittels „div“, „span“ oder „iframe“ in die neue Umgebung integriert. Auch im Folgenden der Arbeit spielt das Integrieren von Inhalt mittels dieser HTML-Elemente eine Rolle. Bei der Umwandlung von iGoogle-Gadgets wird beispielsweise ein „iframe“ verwendet. Näheres dazu enthält das Kapitel 4.2.3.

2.3. Modellgetriebene Entwicklung

Eine vollständige modellgetriebene Entwicklungsmöglichkeit für Widgets ist im Moment nicht bekannt. Allerdings gibt es im Internet bereits eine Lösung, welche ähnlich zu MDD funktioniert. Netvibes bietet mit der Universal Widget API (UWA) ein freies Widget-Framework an, welches auf Extensible Hypertext Markup Language (XHTML) für die Strukturierung, Cascading Style Sheets (CSS) für die Gestaltung und JavaScript für das Verhalten des Widgets basiert [26]. Das UWA-Format nimmt dabei die Stellung eines Meta-Widget-Formates ein. Netvibes bietet mit UWA zwar keine höhere Abstraktionsebene und kein Modell für das Erstellen der Widgets an. Allerdings lassen sich die UWA-Widgets direkt automatisiert in verschiedenste Widget-Formate exportieren, wie zum Beispiel iGoogle, Microsoft Gadget Sidebar, Apple Dashboard, Opera und einige mehr. Dafür hat Netvibes für jedes unterstützte Widget-Format eine Kompatibilitätsumgebung programmiert, sodass am Widget selbst wenige Änderungen nötig sind. Im UWA-Format gibt es aktuell ein Angebot von 263,150 Widgets (Stand: 15.07.2012) [25]. Allerdings bietet Netvibes derzeit noch keine Export-Möglichkeit in das W3C-Format an.

Einen ähnlichen Ansatz wie Netvibes besitzt das OpenLaszlo-Projekt[1] von Laszlo Systems. OpenLaszlo ist eine Plattform, welche dem Entwickeln und Veröffentlichen von Rich Internet Applications (RIAs) dient. Um eine RIA zu entwickeln, muss der Nutzer eine Datei in der XML-Sprache (eXtensible Markup Language) Laszlo (LZX) erstellen. Im Anschluss legt der Entwickler die LZX-Datei auf einem OpenLaszlo-Server ab. Dort wird sie zur Laufzeit in eine RIA umgewandelt. Als Ausgabeformate stehen sowohl Flash, als auch HTML in Verbindung mit JavaScript und Asynchronous JavaScript and XML (AJAX) zur Verfügung. Zusätzlich gibt es die Möglichkeit die LZX-Datei als Einzelveröffentlichung vom Server zu exportieren. Eine exportierte Datei ist nicht mehr von einem OpenLaszlo-Server abhängig und über einen normalen Webserver bereitstellbar. Sie nennt sich deshalb auch „SOLO“. Seit 2010 kann der „SOLO“-Modus von OpenLaszlo auch zum Generieren von Widgets genutzt werden. Dazu erstellt der OpenLaszlo-Server automatisch ein fertiges Widget-Paket inklusive der nötigen Metadaten. Beim Widget-Export stehen schon mehrere Formate

[1] http://www.openlaszlo.org/

zur Auswahl. In der Dokumentation[2] von OpenLaszlo wird zwar das W3C-Widget-Format als unterstütztes Format genannt, allerdings handelt es sich bei diesem um das Opera-Widget-Format. Ein Export in das W3C-Format mittels OpenLaszlo ist dennoch möglich, da für das Einbinden anderer Widget-Formate nur das Hinzufügen eines entsprechenden XML-Templates nötig ist. [21, 22]

Die aktuell sehr beliebten Apps ähneln sich ziemlich stark den Widgets. Auch sie sind kleine Anwendungen mit dem Unterschied, dass sie nicht auf Internettechnologie basieren. Sie treten vor allem im Kontext von Smartphones und Tablets auf. Auch bei Apps gibt es mittlerweile viele unterschiedliche Formate. Diese sind meist an bestimmte Geräte gebunden. Im Normalfall existiert für jedes Format ein eigener Store. Die vielen Formate hindern die Portabilität der Apps und führen zu einem hohen Aufwand, falls eine Applikation in mehreren Stores angeboten und somit für unterschiedliche Geräte bereitgestellt werden soll. Um dem Problem entgegenzuwirken, hat sich die Cross-Plattform-Entwicklung ergeben. Das Ziel dieser ist die einmalige Entwicklung einer App und Verwendung dieser in verschiedenen Formaten. Die Cross-Plattform-Entwicklung basiert auf den Ideen des MDDs. Aufgrund der Ähnlichkeiten der Apps zu den Widgets lassen sich aus den Fortschritten dieser auch Ansätze für die MDD für Widgets ziehen. Grundsätzlich lassen sich sowohl Vor- als auch Nachteile aus der Cross-Plattform-Entwicklung für den MDD-Ansatz für Widgets ableiten. Ein Nachteil für Apps ist vor allem, dass oft nicht die gesamte Hardware der Geräte genutzt und die Darstellung nicht perfekt an die Umgebung angepasst werden kann. Auch auf Widgets trifft dieser Nachteil zu. Das Problem der nicht nutzbaren Hardware stellt sich bei Widgets allerdings eher durch Funktionalitäten dar, welche die entsprechenden Container anbieten. Durch derartige Funktionalitäten kann die Umgebung jedoch auch einen Zugriff auf Hardware ermöglichen.

Auf einem Vortrag stellte Heiko Behrens im Rahmen der MobileTechCon 2010 hauptsächlich drei Kategorien zur Cross-Plattform-Entwicklung für Apps vor. Diese teilen sich auf in hybride, interpretierte und generierte Apps. Eine hybride App stellt sich als eine Mischung aus zwei unterschiedlichen Technologien bzw. Programmiersprachen dar. Bei der ersten Technologie handelt es sich um die Programmiersprache, welche für das jeweilige Gerät als native Sprache für die Appentwicklung vorgesehen ist. Als zweite Programmiersprache kommt Webtechnologie (HTML, CSS und JavaScript) zum Einsatz. Diese stellt den plattformunabhängigen Teil der App dar. Der native Teil besteht aus einem Browser, welcher im Vollbildmodus läuft und nicht direkt sichtbar ist. Dieser stellt sämtlichen Webinhalt als App dar. Des Weiteren bietet er eine Schnittstelle an, um einen Zugriff auf die nativen Funktionalitäten des Gerätes zu ermöglichen. Dies betrifft vor allem die Verwendung von Hardwarekomponenten. Ein sehr bekanntes Beispiel ist PhoneGap[3]. [1]

[2] http://www.openlaszlo.org/lps4.9/docs/developers/
[3] http://www.phonegap.com/

Die zweite vorgestellte Möglichkeit sind die interpretierten Apps. Sie sind im Prinzip ein Spezialfall der hybriden Apps. Sie bestehen im Bereich der plattformunabhängigen Technologie allerdings nicht aus Webtechnologie, sondern einer Auszeichnungs- oder Programmiersprache, wie zum Beispiel JavaScript, Ruby oder XML. Der native Teil der App enthält einen Interpreter, welcher den unabhängigen Teil für das entsprechende Gerät interpretiert. Die Benutzeroberfläche entsteht im Gegensatz zur hybriden Lösung aus nativen Möglichkeiten und nicht aus HTML und CSS. Dies verbessert die Darstellung und das Nutzererlebnis, da es besser an die Plattform angepasst ist. Appcelerator Titanium[4] ist ein Beispiel für interpretierte Apps. Es verwendet als unabhängige Sprache JavaScript. Den Code dieser interpretiert zur Laufzeit ein Browser im Hintergrund. Mittels einer API erhält der Entwickler den Zugriff auf die Hardware und die nativen Benutzeroberflächenobjekte. [1]

Bei generierten Apps ist das Ziel echte native Apps automatisch aus Anwendungen in einer bestimmten Sprache zu genieren. Ein Codegenerator cross-kompiliert diese in die nativen Sprachen der Zielplattformen. Ein großer Nachteil ist meist, dass der entstandene Code deutlich als automatisch generierter erkennbar ist und Änderungen von Hand an diesem kaum möglich sind. XMLVM[5] ist ein Beispiel aus der Kategorie der generierten Apps. Es basiert auf Java-Code und kompiliert diesen mittels eines XML-Modells in die nativen Sprachen. Dazu vereinfacht es sämtlichen Code auf die Möglichkeiten einer Stack-Maschine. Probleme treten allerdings bei plattformspezifischen Methoden auf. Ein weiteres Beispiel ist Applause[6]. Es basiert auf der Überlegung einen eigenen Generator für eine Menge ähnlicher Apps zu bauen. Dadurch ist dieser zwar nicht für alle Arten von Apps geeignet, für die speziellen eignet er sich hingegen umso besser. Für die Entwicklung des Generators muss der Entwickler sich auf eine Menge von Apps beschränken, welche aus immer wiederkehrenden Bausteinen aufgebaut sind. [1]

Die drei genannten Verfahren lassen sich auf die MDD-Entwicklungsvariante für Widgets beziehen. Allgemein gestaltet sich die Umwandlung einfacher als bei Apps, da bei Widgets im Allgemeinen die gleichen Technologien (HTML, CSS, JavaScript) zum Einsatz kommen. Das von Netvibes eingesetzte Verfahren ähnelt dem der Kategorie der interpretierten bzw. hybriden Apps. Der vom Entwickler geschriebene JavaScript-Code des Widgets an sich erfährt keine Änderungen. Er wird in ein neues Widget im für den Export ausgewählten Format eingepackt. Dieses enthält zusätzlich JavaScript-Code, welcher eine angepasste Variante aller Methoden der UWA-Umgebung enthält. Eine Hülle bildet die originale Umgebung für das andere Format nach. Die Metainformationen und Konfigurationsdaten werden zwar generisch für das neue Format umgewandelt, allerdings bleiben sie zusätzlich in ihrer originalen

[4] http://www.appcelerator.com/
[5] http://www.xmlvm.org/
[6] http://www.applause-framework.com/

Form im Widget enthalten. Der JavaScript-Code des exportierten Widgets verwendet zur Laufzeit nicht die Methoden der API des exportierten Formats, sondern die redundante Version im UWA-Speicherformat. Eine Variante des MDDs auf Basis des Vorgehens der generierten Apps ist auch denkbar. Es erhöht allerdings die Komplexität der Umwandlung, da der vom Entwickler eingegebene Code angepasst werden muss. Für das Beispiel der Metainformationen und Konfigurationsdaten muss der Generator beispielsweise den kompletten JavaScript-Code durchsuchen und entsprechend verändern. Eine generische Variante ist somit nicht sehr sinnvoll, da JavaScript dynamisch ist und beispielsweise durch AJAX Modifikationen erfahren kann.

2.4. Weiterverwendung vorhandener JavaScript-Anwendungen

Im Bereich der Widget-Generierung aus JavaScript-Anwendungen existieren im Internet bereits einige Ansätze. Robin Berjon bietet auf seiner Webseite[7] einen kurzen Perl-Script-Code mit dem Namen „AppCache to Widget Converter (ac2wgt)“ an, mit welchem W3C-Widgets automatisiert erstellbar sind. Zum Erzeugen der Widgets benutzt es den HTML5 Offline Application Cache. Der Cache ermöglicht das lokale Speichern von gesamten Webseiten inklusive HTML, JavaScript, CSS und Mediendateien beim Nutzer, damit er die Seite auch bei Nichtexistenz einer Internetverbindung weiter verwenden kann. Um dem User Agent (UA) anzuzeigen, welche Dateien er cachen muss, gibt es die Manifest-Datei und das Manifest-Attribut im HTML-Dokument. Im Attribut wird ein Verweis auf die Manifest-Datei gegeben, welche alle zu cachenden Dateien auflistet. Das Script-Programm liest alle benötigten Dateien aus der Manifest-Datei, lädt diese herunter, erstellt eine passende Konfigurationsdatei und packt das Widget zu einem fertigen Widget-Paket zusammen. Es befindet sich aktuell in der Version 0.01 und ist noch nicht an die Empfehlung des Widget-Formates durch das W3C angepasst, sondern basiert auf einem Arbeitsentwurf für die Standardisierung. Die Anwendung stellt eine Prototypen-Implementierung da, welche mehr der Machbarkeitsanalyse dient. [2]

Ein weiterer Ansatz stammt von Scott Wilson, welcher auch an der Entwicklung des W3C-Widget-Formates beteiligt war. Bei einer Untersuchung des Google Chrome Web Stores auf das Speicher-Format der Applikationen, welche in den Google Chrome Browser installiert werden können, entdeckte er, dass sich dieses kaum vom W3C-Widget-Format unterscheidet. Die heruntergeladenen Anwendungen besitzen eine „.crx“-Dateinamenserweiterung und bestehen aus einer 7-zip-Archiv-Datei, welche jeweils eine JSON-Datei mit dem Name „manifest.json“ enthalten. Der Inhalt des 7-zip-Archivs entspricht dabei dem eines W3C-Widgets nahezu komplett, mit dem Unterschied, dass die Manifest-Datei anstatt einer XML-Konfigurationsdatei

[7] http://berjon.com/hacks/ac2wgt/

vorhanden ist. Scott Wilson hat auf Basis seiner Entdeckung ein Java-Programm für die Konvertierung von „.crx“-Dateien in W3C-Widgets programmiert und dieses auf github.com[8] öffentlich zum Download bereitgestellt. Der Formatwandler entpackt das 7-zip-Archiv, erstellt eine Konfigurationsdatei, welche passend zur W3C-Spezifikation ist und packt das Ganze in ein neues Widget-Paket. Wie beim „AppCache to Widget Converter“ dient die zur Verfügung stehende Anwendung einem Machbarkeitsnachweis. [35]

2.5. Erstellen von Widgets aus RSS-Feeds

Für das Erstellen von Widgets aus RSS-Feeds für das W3C-Format ist noch keine Lösung bekannt. Für einige andere Widget-Formate hingegen können Widgets aus RSS-Feeds mittels Netvibes[9] generiert werden. Dazu muss man nur die URL des Feeds angeben. Aus dem Feed werden alle nötigen Metainformationen ausgelesen. Der Nutzer kann danach die Metainformationen noch anpassen und ergänzen. Durch das UWA-Format ist das Widget in vielen Umgebungen nutzbar. Auch Opera bietet die Möglichkeit aus RSS-Feeds Widgets[10] zu generieren. Das Verfahren läuft ähnlich zu Netvibes ab. Allerdings kann der Nutzer zusätzlich das Aussehen durch Auswahl eines Skins an seine Vorlieben anpassen. Der Nachteil an diesem Widget-Generator ist die Bindung an das Opera-Widget-Format, welches nur in einer Opera-Widget-Umgebung lauffähig ist.

2.6. Konvertierung bestehender Widgets

Für das Umwandeln von Widgets in W3C-Widgets sind derzeit noch keine vollständigen Lösungen bekannt, weshalb sich die Arbeit mit dieser Möglichkeit der Widget-Generierung beschäftigt. Es gibt allerdings bereits einen Online-Dienst[11], welcher es ermöglicht die Konfigurationsdateien von verschiedenen Formaten ineinander zu überführen. Die unterstützten Formate sind Bondi, Nokia, Opera und W3C. Nokia nennt die verwendete Konfigurationsdatei „info.plist“, bei allen anderen ist der Name der Datei „config.xml“. Insgesamt unterscheiden sich Bondi-, Nokia- und Opera-Format kaum von dem des W3Cs. In einigen Fällen reicht ein Anpassen der Konfigurationsdatei für das Zielformat und ein Neupacken des Widget-Pakets. Lediglich bei den jeweiligen JavaScript-APIs kann es zu Problemen durch die Konvertierung

[8] https://github.com/scottbw/crx2widget
[9] http://de.eco.netvibes.com/apps/create
[10] http://widgets.opera.com/widgetize/start
[11] http://mediaworks.metropolia.fi/wp-content/xml/form.php

kommen, da die Anbieter zum Teil unterschiedliche Funktionalitäten durch ihre Umgebung zur Verfügung stellen. Bei Tests des Onlinedienstes zeigten sich jedoch einige Fehler. Bei der Verwendung mehrerer Icons für ein Widget beachtet das Konvertierungstool beispielsweise nur das Erste. Weitere Probleme traten bei der Umwandlung von Angaben zu benötigten Features eines Widgets und bei der Übertragung der ID in ein anderes Format auf.

2.7. Bewertung

In der Tabelle 2.1 sind die Bewertungen der Lösungen Netvibes, OpenLaszlo, „ac2wgt“ und „crx2widget“ aufgelistet. Die zwei betrachteten Lösungen zum Generieren von Widgets sind nicht mit in der Tabelle aufgelistet, da das Endergebnis kein W3C-Widget ist, sondern noch einen Konvertierungsvorgang vom jeweiligen Format in das W3C-Format erfordert. Deshalb ergibt sich die Bewertung in Analogie zum jeweiligen Umwandlungsprogramm mit dem Unterschied, dass der Aufwand etwas gestiegen ist, da zunächst das jeweilige Widget aus dem RSS-Feed generiert und im Anschluss umgewandelt werden muss. Weiterhin befindet sich kein Eintrag zum in Kapitel 2.6 vorgestellten Online-Dienst in der Tabelle, da nur durch das Umwandeln der Konfigurationsdatei kein fertiges nutzbares W3C-Widget entsteht.

Eingabedaten	**Angebot**	**Aufwand**	**Anpassung**	**Verwendbarkeit**
Modellgetriebene Entwicklung:				
Netvibes	+ +	+	+ +	-
OpenLaszlo	- -	+	O	O
Weiterverwendung vorhandener JavaScript-Anwendungen:				
ac2wgt	-	+	O	+
crx2widget	+ +	+	O	+
+ + sehr gut, + gut, O teilweise, - unzureichend, - - nicht gegeben				

Tabelle 2.1.: Bewertung bereits vorhandener Lösungen zur Generierung eines W3C-Widgets

Die für die Konvertierung günstigsten Angebote bieten das UWA- und das CRX-Format. Für beide gibt es jeweils ein großes Portal mit tausenden Anwendungen. Auf Basis des HTML5 Offline Application Caches entstehen seit neuestem einige Applikationen. Allerdings sind diese überall im Internet verteilt und es gibt kein Portal, wo eine größere Menge dieser zur Verfügung stehen. Deshalb ist die Generierung von W3C-Widgets mit diesem Verfahren mit Suchaufwand verbunden, was sich negativ

auf die Angebotsbewertung auswirkt. Bei OpenLaszlo sind allgemein wenige Anwendungen auffindbar. An Portalen gibt es für OpenLaszlo lediglich eine kleine Seite[12] mit weniger als 50 Applikationen, wobei viele Angebote nur Demonstrationen für die verschiedenen Möglichkeiten sind. Der Konvertierungsvorgang ist bei allen vier bewerteten Verfahren mit jeweils einem Klick ausgeführt, jedoch wird immer jeweils nur ein Widget pro Klick generiert. Es gibt bei keinem die Möglichkeit mehrere Applikationen am Stück zu W3C-Widgets zu konvertieren. An das Widget-Umfeld ist natürlich ein Widget-Format am besten angepasst. Bei den anderen drei Lösungen handelt es sich bei der Eingabe mitunter um Anwendungen, die eine große Anzeigefläche benötigen. Bei einem Test der vier Verfahren war leider keines der vier Verfahren sofort verwendbar. Das Konvertierungsprogramm für Applikationen aus dem Chrome Web Store benötigt zwar keine Änderungen, allerdings funktionierte das Greasemonkey-Skript für das Downloaden der Apps im Test nicht. Beim „ac2wgt“-Verfahren sind für die Verwendbarkeit auch ein paar Anpassungen nötig, da es nicht auf der aktuellen Version des W3C-Formates basiert. OpenLaszlo benötigt für eine Ausgabe von W3C-Widgets ein neues XML-Template, da aktuell nur das Opera-Format vorhanden ist. Die umfangreichsten Änderungen sind für die Netvibes-Lösung zu erwarten.

2.8. Fazit

Die Bewertung hat gezeigt, dass aktuell keine Lösung existiert, die alle Anforderungen wenigstens gut erfüllt. Bei Netvibes könnte ein Umwandlungstool dazu führen, dass es allen gestellten Erfordernissen genügt, weshalb es auch in die Lösung als eines der Formate aufgenommen wurde. Vor allem das Problem der Anpassung stellt für die Verfahren ein Hindernis dar, da zum Beispiel die Optimierung einer Nutzeroberfläche ohne menschliche Interaktion nicht möglich ist. Deshalb eignet sich das Verfahren der Konvertierung für die W3C-Widget-Generierung besonders, da die Anpassung an das Widget-Umfeld bereits vorhanden ist.

[12] http://laszlocode.com/index.php

3. Lösungskonzept

3.1. Grundbestandteile eines Widgets

Damit die entsprechenden Widget-Formate für die Entwicklung einer Konvertierungsapplikation besser miteinander verglichen werden können, soll zunächst eine Betrachtung der Grundbestandteile eines Widgets stattfinden. Die fünf Hauptbestandteile sind die Metainformationen, die Konfigurationsdaten, der Inhalt, die Nutzereinstellungen und die Lokalisierung. Die drei erstgenannten Komponenten bilden die Grundvoraussetzungen für ein Widget-Format. Die anderen beiden Bestandteile sind nicht zwangsweise notwendig.

3.1.1. Metainformationen

Die Metadaten eines Widgets dienen vor allem zur Präsentation in einem Widget-Store und der Verwaltung. Zu den Standard-Metainformationen gehören der Name des Widgets und eine Kurbeschreibung der Funktionalität. Diese Daten sind notwendig, damit ein Nutzer bei der Auswahl eines Widgets weiß, was ihm dieses nützt. Weiterhin gehören Informationen über den Autor zu den wichtigsten Metadaten. Häufig genutzt sind Name, E-Mail-Adresse und Adresse der Homepage des Autors. Außerdem gibt es bei allen wichtigen Widget-Formaten einen Verweis auf ein Icon in den Metainformationen. Zusätzlich bieten manche Formate die Möglichkeit Verweise auf Screenshots bzw. Thumbnails für das Widget in den Metadaten zu setzen. Die Fähigkeit zum Ablegen einer ID mittels Angabe einer eindeutigen URI ist auch bei einigen Widget-Formaten vorhanden. Zusätzlich ist es bei manchen Formaten möglich dem Widget innerhalb der Metainformationen eine Kategorie zuzuweisen. Wenn die Zuweisung einer Kategorie in den Metadaten nicht unterstützt wird, ist dies meist beim Hochladen in einen Widget-Store möglich. Eine weitere oft genutzte Angabe ist die Versionsnummer eines Widgets. Selbstverständlich ist dies nur eine Auflistung der am häufigsten genutzten Metainformationen. Einige Widget-Formate bieten noch einige Möglichkeiten mehr.

3.1.2. Konfigurationsdaten

Die Konfigurationsdaten sind für die Widget-Umgebungen notwendige Daten, damit diese die Widgets ordnungsgemäß darstellen können. Zu den wichtigsten Konfigurationsinformationen zählen die Angaben zu Höhe und Breite das Widgets. Häufig stellen die Widget-Umgebungen (Container) mehrere Darstellungsmodi für die Widgets zur Verfügung. Bei solchen Containern kann oft auch der entsprechende Standardmodus mittels einer Angabe in den Konfigurationsdaten spezifiziert werden. Ein weiterer Konfigurationsparameter ist in den meisten Fällen für das Anfordern von Features für das Widget vorhanden. Features sind Laufzeit-Komponenten, welche durch die Umgebung angeboten werden, um dem Widget zusätzliche Funktionalitäten zur Verfügung zu stellen. Auch bei den Konfigurationsdaten bieten manche Widget-Formate noch einige Möglichkeiten mehr, beispielsweise einen Parameter zum Aktivieren eines Debug-Modus oder die Auswahl der Startdatei des Widgets.

3.1.3. Inhalt

Der Inhaltsbereich dient der Spezifikation des eigentlichen Widgets, welches der Anwender benutzt. Er enthält sämtlichen HTML-, CSS- und JavaScript-Code aus welchem das Widget besteht. Der HTML-Code beschreibt den Aufbau, der CSS-Code die Gestaltung und der JavaScript-Code die Funktionalität des Widgets.

3.1.4. Nutzereinstellungen

Die Nutzereinstellungen gehören im Prinzip zum Bereich der Konfigurationsdaten. Allerdings sind sie nicht als statische Informationen im Widget abgelegt, sondern können jederzeit dynamisch verändert werden. Die Präferenzen sind persistent gespeicherte Paare aus einem Schlüssel und einem zugehörigem Wert. Sie sind stets zu einer spezifischen Widget-Instanz zugeordnet. Wenn ein Nutzer in einem Widget beispielsweise die Möglichkeit hat die Schriftgröße seinen Vorlieben anzupassen, so kann der vom Benutzer ausgewählte Wert als eine Nutzereinstellung gespeichert werden. Sollte der Anwender das Widget daraufhin schließen und später wieder öffnen, findet er die Schrift wieder in der ausgewählten Größe vor. Die Nutzereinstellungen bekommen den Wert im JavaScript-Programm-Ablauf zugewiesen. Bei einigen Widget-Formaten können sie aber auch direkt vom Benutzer einen Wert erhalten. Das Setzen einer Standardeinstellung für die Präferenzen ist bei allen betrachteten Widget-Formaten möglich.

3.1.5. Lokalisierung

Die Lokalisierung dient der automatischen Anzeige des Widgets in einer vom Nutzer ausgewählten Sprache, falls diese vom Widget unterstützt ist. Dies erspart dem Benutzer das separate Einstellen der Sprache für jedes Widget. Bei den meisten Widget-Formaten ist nicht nur eine Lokalisierung durch eine Sprachauswahl, sondern auch eine Selektion nach der Region vorhanden. So können beispielsweise Übersetzungen für amerikanisches und britisches Englisch im Wieder enthalten sein. Häufig ist zudem ein Rückfallverhalten für die Lokalisierung implementiert. Findet sich zu einer speziellen Sprach- und Regionsauswahl keine Übersetzung, nutzt das Widget die nächste allgemeinere Auswahl. Wenn in einem Widget beispielsweise jeweils eine Lokalisierung für amerikanisches Englisch und Standardenglisch vorhanden sind, der Nutzer aber britisches Englisch eingestellt hat, so nutzt das Widget automatisch die Darstellung in Standardenglisch. Falls durch das Rückfallverhalten keine zu den Nutzereinstellungen passende Übersetzung vorhanden ist, enthalten die Widgets meist eine Standard-Lokalisierung, welche in solchen Fällen zum Einsatz kommt.

3.2. Ausgewählte Widget-Formate

3.2.1. Opera-Widget

Das erste für die Konvertierung ausgewählte Format ist das Opera-Widget-Format. Im April 2007 veröffentlichte Opera die erste Version einer Spezifikation für ihre Widgets [32]. Sie basiert auf einem frühen Arbeitsentwurf der W3C-Widget-Spezifikation. Das Opera-Widget-Format hat sich mit der Zeit weiterentwickelt. Die neueste Version der Spezifikation ist die 4. Edition vom 09.02.2010 [23]. Das Opera-Widget-Format ist dem W3C-Format sehr ähnlich, hat zur aktuellen Version der W3C-Spezifikation jedoch einige Unterschiede, da sich auch diese weiterentwickelt hat. Aufgrund der hohen Ähnlichkeiten eignet es sich sehr gut für das Konvertierungstool, da generell am Grundaufbau keine größeren Änderungen zu erwarten sind. Die Modifikationen beschränken sich auf einige Details.

3.2.2. iGoogle-Gadget

Im Jahr 2005 veröffentlichte Google das Produkt iGoogle. Es stellt eine personalisierbare Einstiegs-Webseite dar, die der Nutzer am besten als Startseite des Browsers nutzen sollte. Die Komponenten zur Individualisierung der Seite sind die sogenannten Gadgets. Von diesen gibt es tausende verschiedene. iGoogle ist im Moment wahr-

scheinlich die bekannteste Plattform für Widgets im Internet. Die meistgenutzten Gadgets haben alleine mehrere Millionen Nutzer [10]. Am 04.07.2012 hat Google angekündigt iGoogle am 01.11.2013 einzustellen, da die Nutzerzahlen fortwährend sinken und das gesamte System, trotz einiger Updates in den letzten Jahren, veraltet ist. Das iGoogle-Format wurde aufgrund der hohen Popularität und dem großen Angebot für die Konvertierung ausgewählt. [16]

3.2.3. UWA-Widget

Das UWA-Format von Netvibes ist das letzte für die Konvertierung betrachtete Widget-Format dieser Arbeit. Wie bei iGoogle existieren bereits sehr viele Widgets. Da es sich von Netvibes aus bereits schon in viele andere Formate konvertieren lässt und den modellgetriebenen Entwicklungsansatz besitzt, ist es auch für eine Konvertierung in das W3C-Format interessant und sollte ohne größere Schwierigkeiten möglich sein. Einen großen Vorteil für die Umwandlung bietet auch die Exportmöglichkeit in das W3C-ähnliche Opera-Format, da sowieso eine Untersuchung der Opera-Widgets im Rahmen der Arbeit stattfindet.

3.3. Allgemeines Konvertierungskonzept

Ein Tool für die Konvertierung eines Widgets in das W3C-Format sollte alle fünf Teilbereiche betrachten. Das Umwandeln lässt sich grundsätzlich in vier Schritte aufteilen. Zunächst sollte das Tool die Konfigurationsdatei „config.xml“ erstellen und diese mit so vielen Konfigurationsdaten und Metainformationen wie möglich bereichern. Bei den Metainformationen anderer Formate sind häufig Referenzen auf entfernte Dateien wie zum Beispiel Icons vorhanden. Das Umwandlungsprogramm kann die referenzierten Dateien mit in das Widget-Paket aufnehmen und die Referenz durch eine lokale Variante dieser ersetzen. Im zweiten Schritt integriert es den Inhalt aus dem ehemaligen Widget in das neue W3C-Widget. Dazu muss die Applikation an diesem eventuell Anpassungen vornehmen. Der Inhalt aller in der Arbeit betrachteten Formate teilt sich in HTML, CSS und JavaScript auf. Dies ermöglicht es, sich auf eine HTML-Startdatei zu beschränken, welche die entsprechenden JavaScript- und CSS-Abschnitte besitzt. Bei der Umwandlung des Inhalts sollte das Tool auch das Lokalisierungsverfahren des Quelldatei-Formats beachten. Es muss alle Lokalisierungen erkennen und für jede entsprechend angepasste Kopien der Inhaltsdateien im „locales“-Ordner anlegen. Im sich anschließenden Schritt kümmert sich die Anwendung um die jeweilige JavaScript-API, die der Inhalt verwendet. Dafür müssen Dateien mit JavaScript-Code zur Verfügung stehen, welche die Funktionalitäten wie die originalen Widget-Umgebungen anbieten. Diese bindet die Applikation in das

Widget-Paket ein, indem sie die Dateien kopiert und mit „script“-Elementen Referenzen auf die neuen Dateien setzt. Im letzten Schritt muss das Tool nur noch alle bisher gesammelten Dateien und Ordner zum richtigen Widget-Paket zusammenpacken.

Die Implementation sollte zwei Möglichkeiten für die Umwandlung anbieten. Beim ersten Verfahren übergibt ihr der Nutzer ein einzelnes Widget. Dieses konvertiert sie einzeln, wobei durchaus Nutzerinteraktionen für sonst unbekannte Werte oder nicht unterstützte API-Funktionen während dem Vorgang auftreten können. Der zweite Modus sollte eine Konfigurationsdatei in Form einer „txt“-Datei entgegennehmen. In dieser kann der Benutzer Standardwerte, den Eingangspfad mit den Quell-Widgets, das Ausgabeverzeichnis u. ä. definieren. Die Anwendung verarbeitet bei diesem Verfahren mehrere Widgets am Stück und verzichtet auf Nutzerinteraktion. Es handelt sich bei diesem Modus um eine Stapelverarbeitung. Falls ein Wert unbestimmt aber benötigt ist, setzt sie den Standardwert ein. Bei nicht unterstützten API-Funktionen oder anderen Problemen bricht sie die Umwandlung des einzelnen Widgets ab und fährt mit dem nächsten fort. Die Ergebnisse dieses Modus sollten sich nach Abschluss des Vorgangs in einer Logdatei in Form einer „txt“-Datei im Zielverzeichnis befinden.

Zuordnung der Grundbestandteile zu den Konvertierungsschritten

Die ersten beiden Grundbestandteile (Meta- und Konfigurationsdaten) lassen sich eindeutig in die erste Phase des Umwandelns einordnen. Aus dem Ursprungs-Widget extrahiert die Anwendung alle Meta- und Konfigurationsinformationen und sucht entsprechende äquivalente Elemente und Attribute aus dem Namensraum der W3C-Spezifikation für die „config.xml“-Datei. Zu beachten ist, dass eventuell das Zuordnen nicht genügt, sondern durchaus auch Anpassungen an den Werten notwendig sind. Alle extrahierten Daten, die sich nicht konvertieren lassen, kann die Applikation als proprietäre Erweiterungen in die Konfigurationsdatei des W3C-Widgets einbinden. Sie definiert einen Namensraum mit Präfix und zeichnet die neuen Elemente und Attribute mit diesem aus. Eine entsprechend erweiterte Widget-Umgebung kann aus den zusätzlichen Angaben eventuell einen Nutzen gewinnen. Das Ziel sollte jedoch immer sein, so viele Daten wie möglich in den Elementen und Attributen des Widget-Namensraumes unterzubringen, ohne eine Ergänzung zu gebrauchen. Eine Ausnahme unter den Konfigurationsdaten bilden die jeweiligen Komponenten für das Anfordern von Zusatzfunktionen. Das Tool sollte sich alle angeforderten Features für den dritten Schritt vormerken. Dadurch weiß es später, welche Funktionalitäten es bereitstellen muss.

Die Umwandlung des dritten Bestandteiles beschränkt sich nicht auf eine Phase. Zunächst findet die eventuelle Anpassung des Inhalts in Phase zwei statt. Diese kann

sich auf das Kopieren der Dateien bzw. des Codes in das neue Widget beschränken. Anschließend muss sich die Anwendung im dritten Schritt um die Bereitstellung der benötigten API-Funktionen kümmern. Der vierte Bestandteil ordnet sich mit in die erste und dritte Phase ein. Zunächst sollte die Anwendung die Konfigurationsdaten der Nutzereinstellungen konvertieren. Im Anschluss fügt sie im dritten Schritt die Funktionalitäten, zum Laden und Speichern der Präferenzen aus dem JavaScript-Code, hinzu. Die Aufrufe müssen sich analog zu ihren Vorbildern des Quellformates verhalten und sollten zugleich die Schnittstellen aus der W3C-Spezifikation benutzen. Auch der letzte Bestandteil lässt sich nicht auf eine Phase reduzieren. Es kann sogar vorkommen, dass er Einfluss auf die ersten drei Schritte nimmt. Hauptsächlich ist der Inhalt betroffen. Für jede angebotene Lokalisierung erstellt die Anwendung eine angepasste Kopie der zu lokalisierenden Dateien. Des Weiteren muss sie bei einer Lokalisierung der Metainformationen im Original-Format auch die Metadaten des W3C-Widgets anpassen. Abhängig vom jeweiligen Datum kommt es zum Einsatz der ordnerbasierten Lokalisierung oder des „xml:lang“-Attributs. Alle Änderungen durch die Nutzung des „xml:lang“-Attributs sollte das Tool in Phase eins abarbeiten. Weiterhin sollte es das Herunterladen und Kopieren von Metadateien, welche von der ordnerbasierten Lokalisierung betroffen sind, in diesem Schritt durchführen. In Phase eins kann zusätzlich der Konfigurationsparameter „defaultlocale“ von Änderungen betroffen sein. Änderungen am vorletzten Schritt durch den fünften Bestandteil ergeben sich aufgrund eventuell vorhandener JavaScript-Funktionen durch eine Lokalisierung. Für diese Funktionalitäten sollte selbstverständlich auch ein Ersatz im W3C-Widget vorhanden sein.

3.4. Konvertierung des Opera-Widget-Formates

Für Opera-Widgets der Strategie eins ist aufgrund der hohen Ähnlichkeit zum W3C-Format keine strukturelle Änderung nötig. Die Umwandlung begrenzt sich auf einige Details. Opera-Widgets der Strategie zwei muss ein Konfigurationstool hingegen zunächst in die Strategie eins überführen. Es muss dazu alle Dateien aus dem einzigen Verzeichnis des Widget-Pakets in das Wurzelverzeichnis kopieren. Im Anschluss unterscheiden sich die Konvertierungsschritte beider Strategien nicht mehr voneinander. Für die Umwandlung muss die Applikation vor allem an vielen Metainformationen und Konfigurationsdaten eine Anpassung vornehmen. Bei Opera sind einige Parameter als Elemente hinterlegt, für die beim W3C-Format nur ein Attribut vorhanden ist. Außerdem hat Opera einige zusätzliche Konfigurationsparameter hinzugefügt, welche maximal als proprietäre Erweiterung in das W3C-Format passen.

Im Bereich des Inhalts außer der JavaScript-API sind nur geringe Anpassungen am CSS-Code notwendig. Opera hat für die eigenen Widgets neue Ansichtsmodi defi-

niert. Diese sind nicht mit den beim W3C-Format verwendeten CSS-Ansichtsmodi kompatibel, weshalb die Anwendung die Modi ineinander überführen sollte. Durch die Überführung sind allerdings auch die definierten „-o-widget-mode“-CSS-Regeln nicht mehr in Gebrauch. Deshalb sollte das Programm die CSS-Definitionen in das „view-mode“-Medien-Feature umwandeln. Dazu muss es auch die in den Medien-Anfragen genutzten Ansichtsmodi ineinander konvertieren. Alle restlichen nötigen Änderungen beziehen sich auf die JavaScript-API. Da das W3C die Spezifikation der JavaScript-API für die Widgets in den letzten Jahren stark verändert hat, sind nicht nur die von Opera hinzugefügten Funktionalitäten betroffen. Bei einigen aus der W3C-Spezifikation entnommenen Funktionen genügt das Anpassen der Aufrufe. Andere hat das W3C komplett gestrichen, weshalb eine neue Bereitstellung dieser nötig ist. Dies trifft natürlich auch auf die opera-spezifischen Funktionen zu. Das Hinzufügen einiger Methoden würde allerdings eine Modifikation der standardmäßigen Widget-Umgebung mit sich führen. Deshalb ist das Konvertieren derartiger Opera-Widgets nur durch Anpassungen des Nutzers möglich, welche eventuell auch die Funktionalität dieser einschränken können.

3.5. Konvertierung des iGoogle-Gadget-Formates

Bei iGoogle-Gadgets muss das Umwandlungs-Tool den kompletten Aufbau modifizieren. Zunächst kann es die Widget-Datei mit dem von Google gegebenen XML-Schema validieren, wodurch es das Konvertieren von fehlerhaften Widgets sofort abbrechen kann. Danach sollte es sämtliche Metainformationen und Konfigurationsdaten extrahieren. Alle in den Metadaten verlinkten Dateien sollte es zusätzlich herunterladen und mit in das neu zu erstellende Widget integrieren. Eine besondere Behandlung benötigen vor allem alle in iGoogle lokalisierten Metadaten. Die Applikation muss die Daten je nach vorgesehenem Verfahren mit dem Attribut „xml:lang“ oder dem „locales“-Verzeichnis lokalisieren. Bei manchen Metainformationen ist eine mehrsprachige Version im W3C-Format nicht vorgesehen. Dies betrift das „author“-Element und dessen Attribute. Falls die entsprechenden Daten im iGoogle-Gadget durch eine Substitutionsvariable lokalisiert sind, sollte die Anwendung die Standardversion der Übersetzung nutzen. Falls keine derartige Lokalisierung gegeben ist, kann sie auf die englische Edition ausweichen. Im Falle, dass auch keine englische Version vorhanden ist, kann sie entweder die erstmögliche Lokalisierung verwenden oder den Nutzer bitten eine der vorhandenen auszuwählen. Gerade im Bereich der Metainformationen bietet das iGoogle-Format einige mögliche Angaben mehr als ein W3C-Widget. Deshalb sollte das Tool auch bei iGoogle proprietäre Erweiterung einsetzen.

Der gesamte Inhalt eines iGoogle-Widgets kann von einem Ansichtsmodus abhängig sein. Selbst der JavaScript-Code wird an solch einen gebunden. Beim W3C-Format

kann der Ansichtsmodus mit den gegebenen Mitteln nur die Auswahl der CSS-Regeln beeinflussen. Um das Verhalten von iGoogle nachzubilden, nutzt die Applikation eine etwas veränderte Version der Funktion zum Ermitteln des aktuellen Ansichtsmodus aus der Konvertierung des Opera-Formats. Diese gibt anstatt der Opera-Werte „home“, „canvas“ oder „default“ zurück. In Abhängigkeit zum Rückgabewert der Funktion sollte der JavaScript-Code des neuen Widgets den Inhalt zum entsprechenden Ansichtsmodus darstellen. Dazu kann er verschiedene Ansätze nutzen, wie zum Beispiel „innerHTML“ oder einen eingebetteten Frame. Aus dem Inhalt muss die Anwendung auch alle Substitutionsvariablen für die Lokalisierung entfernen und durch die entsprechend hinterlegten Übersetzungstexte austauschen. Dies sollte sie für alle Dateien, die den Inhalt des Widgets enthalten, für jeweils alle angebotenen Lokalisierungsversionen durchführen. Die entsprechend angepassten Dateien muss sie natürlich im passenden Unterordner des „locales“-Verzeichnisses speichern. Bei der Auswahl der einzelnen Übersetzungstexte sollte sie auch das Zurückfallverhalten des iGoogle-Gadgets emulieren. Bei Widgets des Inhaltstyps „url“ muss die Applikation als Inhalt für das W3C-Widget eine „html“-Datei erstellen, welche einen eingebetteten Frame mit der gegebenen URL enthält.

Die vielen Funktionalitäten, welche die umfangreiche JavaScript-API inklusive der Features bereitstellt, sollten JavaScript-Dateien zur Verfügung stellen. Diese sollte das Tool während der Konvertierung in das Widget-Paket integrieren. Außerdem sollte es die JavaScript-Dateien in den Inhaltsdateien mittels „script“-Element referenzieren. Da die API sehr umfangreich ist, ist ein hoher Programmieraufwand zur Bereitstellung aller Funktionen nötig. Allerdings kann sehr viel JavaScript-Code aus dem Apache Shindig - Projekt[1] weiterverwendet werden. Apache Shindig implementiert einen OpenSocial-Container für OpenSocial-Gadgets[2]. OpenSocial-Gadgets entsprechen den iGoogle-Gadgets mit der Ausnahme, dass sie eine OpenSocial-API besitzen und an ein soziales Netzwerk gebunden sind. Da sie auf iGoogle-Gadgets basieren, stehen ihnen auch alle JavaScript-API-Funktionen von der iGoogle-Standard-Umgebung zur Verfügung. Diese sind somit in Apache Shindig programmiert. Da der Quellcode von Shindig offen ist, kann er mit einigen Anpassungen für die Konvertierung sehr nützlich sein und Arbeit ersparen. Die Änderungen beziehen sich vor allem auf die andere Widget-Umgebung. Das völlig unterschiedliche Lokalisierungsmodell erfordert zum Beispiel einige Anpassungen. Außerdem sind gerade die Methoden zur Inter-Widget-Kommunikation und Widget-Container-Kommunikation nicht übernehmbar, da diese zunächst Änderungen an der Widget-Umgebung erfordern.

Ein weiterer Schritt den die Umwandlung abdecken muss, ist das Bereitstellen eines Nutzereinstellungsmenüs. Es gibt grundsätzlich zwei unterschiedliche Verfahren dafür. Bei der ersten Möglichkeit erstellt die Anwendung während der Konvertierung

[1] http://shindig.apache.org/
[2] docs.opensocial.org/

das Menü. Es besteht aus einem „div“-Element, welches alle nötigen Einstellungen enthält. Dieses ist standardmäßig versteckt und kann ein Nutzer beispielsweise durch einen Klick auf einen Button, den das Tool in den Inhalt des Widgets integriert hat, sichtbar machen. Beim zweiten Verfahren übernimmt es die XML-Elemente für die Konfiguration des Einstellungsmenüs mit in das W3C-Widget. Zur Laufzeit erstellt der JavaScript-Code aus diesem XML-Code dynamisch das Menü, welches wieder durch Nutzerinteraktion angezeigt wird.

3.6. Konvertierung des UWA-Widget-Formates

In Bezug auf den grundsätzlichen Aufbau ist bei UWA-Widgets eine ähnliche Änderung wie beim iGoogle-Format nötig, da deren Aufbau ebenfalls aus nur einer Datei besteht. Allerdings gibt es bei Netvibes weder ein Lokalisierungsmodell noch verschiedene Ansichtsmodi, wodurch sich die Konvertierung entsprechend unkomplizierter als bei iGoogle darstellt. Zunächst sollte das Tool wieder sämtliche Metainformationen aus der Widget-Datei extrahieren. Dazu muss es die XML-Elemente „meta“, „title“ und „link“ beachten. Die Bilddateien sind wie beim iGoogle-Format nur durch eine Referenz in das Widget integriert und liegen als entfernte Ressource im Internet vor. Deshalb sollte die Applikation die referenzierten Icons herunterladen, in das Widget-Paket einfügen und die Referenzen an die neuen Speicherorte anpassen. Beim UWA-Format gibt es keine Konfigurationsdaten, für welche die W3C-Spezifikation äquivalente Elemente oder Attribute besitzt. Das Einführen von proprietären Erweiterungen für diese ist auch nicht nötig, da die Daten nur den JavaScript-Code beeinflussen. Das Beachten der Konfigurationen findet zum einen bei der Konvertierung und zum anderen im JavaScript-Teil statt. Es gibt bei UWA-Widgets keine Möglichkeit die Größe für die Darstellung zu bestimmen. Diese legt der Container stets dynamisch fest. Die Höhe ist vom Inhalt abhängig und die Breite von der Widget-Umgebung. Deshalb ist die Anwendung auf Standardwerte oder Benutzereingaben für die Konfigurationsattribute „width“ und „height“ angewiesen.

Der Inhalt des neuen W3C-Widgets entsteht durch das Kopieren der eigentlichen UWA-Datei in das Widget-Paket. Zusätzlich muss das Tool durch das „content“-Element einen Verweis auf diese setzen. In den UWA-Widgets sind meist die Dateien für den Standalone-Modus integriert. Diese Dateien kann die Konvertierungs-Applikation zum Bereitstellen der JavaScript-API und der vordefinierten CSS-Gestaltung für die W3C-Umgebung nutzen. Da die JavaScript-API sehr umfangreich ist, spart dies viel Aufwand und führt zu einem perfekt nachempfundenen Verhalten, da es von Netvibes selbst stammt. Es sind allerdings ein paar Modifikationen am Code nötig, um diesen an die W3C-Umgebung anzupassen. Die entsprechend veränderten Standalone-Dateien muss die Anwendung in das Widget-Paket kopieren. Zusätzlich

muss sie die Verweise auf die lokalen Dateien umändern. Falls eine Referenz auf die Standalone-Dateien im Quell-Widget nicht enthalten ist, muss sie die modifizierten Versionen der Dateien auch in das neue W3C-Widget integrieren. Aus der kopierten UWA-Datei darf sie allerdings die Metainformationen, trotz Übernahme dieser in die Konfigurationsdatei, nicht entfernen, da der JavaScript-Code der API diese weiterverwendet. Allerdings sollte sie bei den „meta“-Elementen mit Verweisen auf Dateien die Werte der „content“-Attribute anpassen, sodass diese auf die lokalen Versionen zeigen.

Für das Konvertieren der UWA-Widgets gibt es noch eine zweite Alternative, welche auf den jeweils im Opera-Format exportierten Dateien basiert. Die Opera-UWA-Widgets werden vom bereits vorhandenen Konvertierungsablauf für Opera-Widgets umgewandelt. Allerdings ist eine kleine Erweiterung des Umwandlungsverfahrens nötig. Opera-UWA-Widgets inkludieren ähnlich zum Standalone-Modus JavaScript- und CSS-Dateien, welche nicht im Widget-Paket enthalten sind und die UWA-Umgebung nachbilden. Diese Dateien benötigen zum Teil kleine Anpassungen für das W3C-Format. Deshalb muss die Applikation modifizierte Kopien der Dateien in das Widget-Paket einbinden und die Verweise auf diese anpassen.

3.7. Architektur des Tools

Die Abbildung 3.1 zeigt ein Anwendungsfalldiagramm für die Applikation basierend auf den in Kapitel 1.3 erfassten Anwendungsfällen. Es gibt die Möglichkeit zur Umwandlung eines einzelnen Widgets, welche durch die unteren drei Anwendungsfälle dargestellt ist. Die Stapelverarbeitung verwendet die drei anderen Module der einzelnen Konvertierung, falls Widgets in den entsprechenden Formaten ausgewählt sind.

Eine Beschreibung des Aufbaus der grundlegenden Module zeigt das UML-Zustandsdiagramm in Abbildung 3.2. Es sind wieder beide Varianten beachtet. Zunächst muss das Tool erkennen, ob es ein einzelnes Widget umwandeln oder eine Stapelverarbeitung durchführen soll. Bei der Stapelverarbeitung muss die Applikation zunächst eine Konfigurationsdatei auslesen, welche die grundsätzliche Einstellung für die Konvertierung enthält. Bei beiden Verfahren erfolgt eine Vorbereitung der Umwandlung. Diese sorgt bei der Stapelverarbeitung für das Sammeln aller Eingabe-Widgets und bei beiden Varianten zur jeweiligen Erstellung der Arbeitsverzeichnisse. Die Arbeitsverzeichnisse dienen dem Sammeln aller notwendiger Dateien für ein W3C-Widget um diese anschließend zu einem Zip-Archiv verpacken zu können. Danach erfolgt die Erkennung der Formate der umzuwandelnden Widgets. Je nach Format wird das entsprechende Konvertierungsmodul aufgerufen. Dieses sorgt für die Umwandlung des Inhalts und stellt die JavaScript-API für die W3C-Umgebung bereit. Es

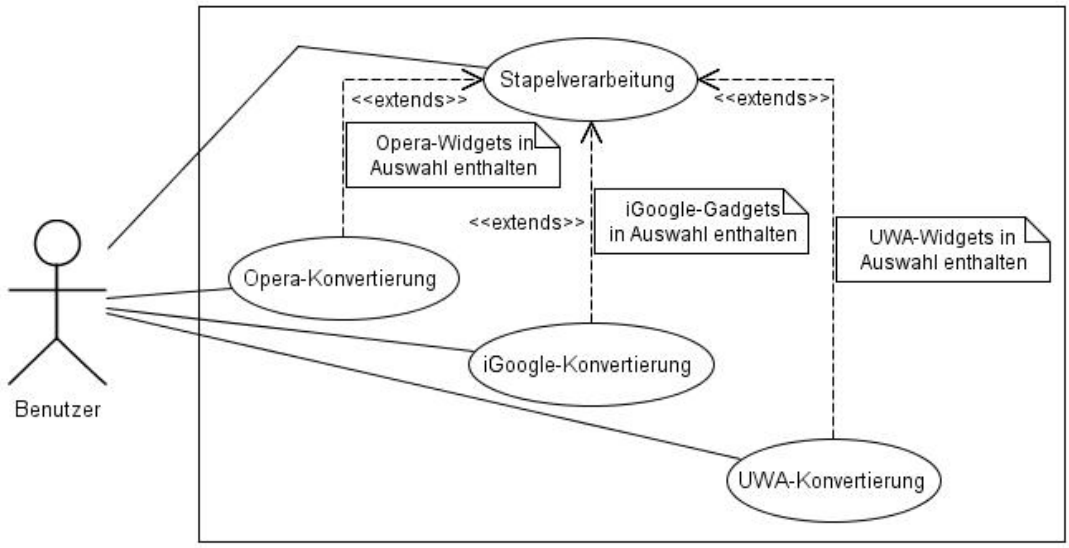

Abbildung 3.1.: Anwendungsfalldiagramm für das Umwandlungstool

kümmert sich somit um die in Kapitel 3.3 erarbeiteten Schritte zwei und drei des allgemeinen Konvertierungskonzepts. Außerdem ruft es ein Modul auf, welches sämtliche Metainformationen und Konfigurationsdaten extrahiert, bei Bedarf modifiziert und schließlich in eine neu erstellte „config.xml“-Datei schreibt. Im letzten Schritt findet der Abschluss der Konvertierung statt, wobei alle gesammelten Dateien im Arbeitsverzeichnis in das fertige W3C-Widget-Archiv gepackt werden. Bei der Stapelverarbeitung schließt sich in diesem Schritt zusätzlich das Schreiben der Log-Daten des Umwandlungsvorgangs in eine Datei an.

3.8. Zusammenfassung

Nachdem im Lösungskonzept zunächst die Grundbestandteile eines allgemeinen Widgets und die für die Umwandlung ausgewählten Formate vorgestellt wurden, folgte anschließend die Beschreibung des allgemeinen Konzeptes. In dieser ergaben sich vier grundlegende Schritte für die automatische Konvertierung eines Widgets in das W3C-Format. Diese sind das Erstellen der Konfigurationsdatei „config.xml“ inklusive dem Eintragen aller Werte, das Integrieren des Inhalts aus dem alten Widget in die neuen Inhaltsdateien, das Nachbilden der originalen Umgebung durch Einbinden von JavaScript-APIs und das Zusammenpacken der Dateien und Ordner zum fertigen W3C-Widget-Paket. Es folgte eine Zuordnung der fünf Grundbestandteile an

die vier Schritte. Danach schloss sich jeweils ein Konvertierungskonzept für die drei ausgesuchten Formate an. Dieses behandelt die jeweiligen groben Schritte zur Umwandlung. Im folgenden Kapitel findet eine genauere Betrachtung der Konvertierung statt, bei der einige Details dargestellt werden.

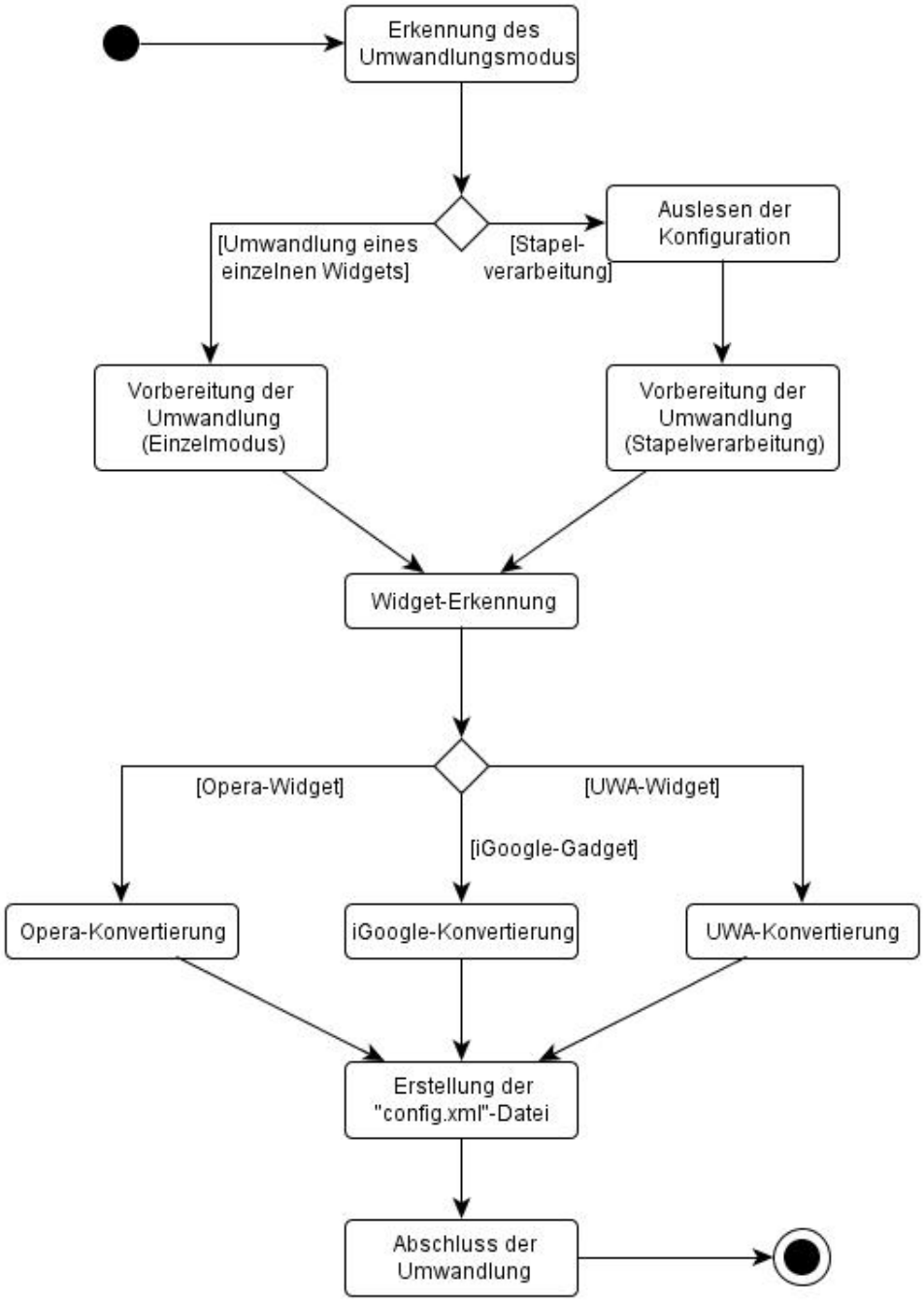

Abbildung 3.2.: Zustandsdiagramm für die Widget-Konvertierung

4. Realisierung

4.1. Konvertierung von Opera-Widgets

4.1.1. Metainformationen

In Tabelle 4.1 sind alle Metainformationen des Opera-Formats zu passenden Attributen und Elementen aus der W3C-Spezifikation zugeordnet. Alle Metadaten, die keine äquivalente Repräsentation im W3C-Format erhalten können, sind durch eine proprietäre Erweiterung eingefügt. Zur Erkennung tragen sie die Abkürzung „prop.“.

Opera-Format			**W3C-Format**		
Name	**Typ**	**Eltern-element**	**Name**	**Typ**	**Eltern-element**
widgetname	Element	widget	name	Element	widget
author	Element	widget	author	Element	widget
name	Element	author	-	Textknoten	author
organization	Element	author	organization	Attribut (prop.)	author
email	Element	author	email	Attribut	author
link	Element	author	href	Attribut	author
description	Element	widget	description	Element	widget
icon	Element	widget	icon	Element	widget
width	Attribut	icon	width	Attribut	icon
height	Attribut	icon	height	Attribut	icon
-	Textknoten	icon	src	Attribut	icon
id	Element	widget	id	Attribut	widget
host	Element	id	id	Attribut	widget
name	Element	id	id	Attribut	widget
revised	Element	id	revised	Attribut (prop.)	widget

Tabelle 4.1.: Zuordnung der Metainformationen vom Opera-Format zum W3C-Format

Beim „id“-Element des Opera-Formats ist vor Übernahme von Informationen ein Test auf das Vorhandensein valider Werte aller drei Attribute nötig. Erst danach darf das Konvertierungstool dessen Daten übernehmen. Die Werte von „host“ und „name“ muss es zu einem Wert für das „id“-Attribut konkatenieren. Falls der „host“ noch keinen Schrägstrich als letztes Zeichen besitzt, muss die Applikation zwischen die beiden Teile zusätzlich einen einfügen, um eine valide URI zu erhalten.

4.1.2. Konfigurationsdaten

Alle Überführungen von Konfigurationsdaten des Opera-Widgets in das W3C-Format sind in Tabelle 4.2 aufgelistet. Proprietäre Erweiterungen sind durch die Abkürzung „prop.“ kenntlich gemacht.

Opera-Format			**W3C-Format**		
Name	**Typ**	**Eltern-element**	**Name**	**Typ**	**Eltern-element**
defaultmode	Attribut	widget	viewmodes	Attribut	widget
dockable	Attribut	widget	dockable	Attribut (prop.)	widget
transparent	Attribut	widget	transparent	Attribut (prop.)	widget
network	Attribut	widget	network	Attribut (prop.)	widget
width	Element	widget	width	Attribut	widget
height	Element	widget	height	Attribut	widget
widgetfile	Element	widget	content	Element	widget
-	Textknoten	widgetfile	src	Attribut	content
feature	Element	widget	feature	Element	widget
name	Attribut	feature	name	Attribut	feature
required	Attribut	feature	required	Attribut	feature
param	Element	feature	param	Element	feature
name	Attribut	param	name	Attribut	param
value	Attribut	param	value	Attribut	param
security	Element	widget	-	-	-
access	Element	security	access	Element	widget
protocol	Element	access	origin	Attribut	access
protocol	Element	access	protocol	Element (prop.)	access
host	Element	access	origin	Attribut	access
port	Element	access	origin	Attribut	access

Opera-Format			W3C-Format		
Name	**Typ**	**Eltern-element**	**Name**	**Typ**	**Eltern-element**
port	Element	access	port	Element (prop.)	access
path	Element	access	path	Element (prop.)	access
content	Element	security	-	-	-
plugin	Attribut	content	plugin	Attribut (prop.)	widget

Tabelle 4.2.: Zuordnung der Konfigurationsdaten vom Opera-Format zum W3C-Format

Für das Konvertieren des „defaultmode“-Attributs ist zuerst eine Zuordnung der Opera-Ansichtsmodi zu den W3C-Ansichtsmodi von nöten. Die einzelnen Modi besitzen keinen 100-prozentigen Ersatz im jeweilig anderen Format, weshalb der nächstbeste passende als Ersatz dient. Zur Auswahl des bestmöglich passenden Ansichtsmodus muss die Anwendung zusätzlich den Wert des „transparent“-Attributs beachten, da bei W3C-Widgets die Hintergrundtransparenz vom jeweiligen Modus abhängig ist. Der Wert „application“ entspricht weitestgehend „windowed“ und „fullscreen“ und wird stets in „fullscreen“ umgewandelt. Beide Zuordnungen sind vom „transparent“-Attribut unabhängig. Der Wert „widget“ bekommt bei erlaubter Hintergrundtransparenz „floating“ zugeordnet. Ansonsten entspricht „widget“ weitestgehend „windowed“. Um das Rückfallverhalten von Opera-Widgets in Bezug auf die Auswahl des Ansichtsmodus nachzubilden, muss das Tool die in der Reihenfolge jeweils folgenden Werte zusätzlich per Leerzeichen separiert an den ausgewählten Wert anfügen. Die umgewandelte Grundreihenfolge ist „windowed“, „fullscreen“ und „floating“. Wenn ein transparenter Hintergrund verboten ist, verkürzt sich die Liste auf „windowed“ und „fullscreen“.

Der Wert des „dockable“-Attribut entspricht der Angabe, ob das Widget den Ansichtsmodus „minimized“ der W3C-Spezifikation unterstützt. Da dies für ein W3C-Widget nicht konfigurierbar ist, fügt die Applikation wenigstens eine proprietäre Erweiterung hinzu. Eine solche Ergänzung führt sie zusätzlich für das „transparent“-Attribut ein, damit bei Änderungen des Ansichtsmodus zur Laufzeit die Einstellung zur Hintergrundtransparenz Einfluss nimmt. Des Weiteren bilet das „network“-Attribut eine proprietäre Erweiterung, da derartige Sicherheitskonfigurationen im W3C-Format standardmäßig nicht möglich sind. Zusätzlich beeinflusst es die Umwandlung des „access“-Elementes. Bei der Konvertierung der Konfigurationsdaten für die Breite und Größe des Widgets sollte die Anwendung den Standardwert von 300

Pixeln bei Abwesenheit der Angaben im Opera-Format verwenden. Beim „content“-Element sollte sie die Kodierung der Startdatei beachten. Falls diese nicht UTF-8-kodiert ist, sollte das „encoding“-Attribut des Elementes deren Namen besitzen. Somit kann „content“ auch bei Verwendung der Standardstartdatei auftreten.

Die umfangreichsten Anpassungen erfordert das „access“-Element, da das W3C-Format nur das „origin“-Attribut besitzt. Durch die Bildung aller möglicher Kombinationen kann die Anwendung die Regeln der Elemente „protocol“, „host“ und „port“ grundsätzlich ohne eine Ergänzung beachten. Der Wert des „origin“-Attributs bildet sich aus der Formel „protocol + „://“ + host + „:“ + port“, wobei das Pluszeichen eine Konkatenation darstellt. Da jeder mögliche Wert mit jedem anderen kombiniert werden, muss können sehr viele Kombinationen und somit auch „access“-Elemente entstehen. Eine Beschränkung auf einen bestimmten Pfad ist ohne proprietäre Erweiterung jedoch nicht möglich. Weiterhin kann man bei einem unbeschränkten Host-Teil nicht auf eine Ergänzung verzichten. Wenn ein Zugriff auf alle Hosts erlaubt sein soll, muss das „origin“-Attribut den Wert „*“ erhalten. Allerdings repräsentiert der Stern auch alle Protokolle und Ports. Um diese daraufhin einzuschränken, muss das Tool neue Elemente als Kind von „access“ nutzen, welche nicht dem Widget-Namensraum entstammen. Bei Zugriffsbegrenzung auf bestimmte Hosts kann es eventuell den Wert des „network“-Attributs beachten. Voraussetzung dafür sind Hostangaben, die sich eindeutig dem privaten Netzwerk zuordnen lassen. Dazu gehören der Wert „localhost“ und alle IP-Adressen, die nach Requests for Comments (RFC) 3330[1] auf das lokale Netz verweisen. Bei Anforderung eines derartigen Hosts und Abwesenheit von „private“ im „network“-Attribut kann die Applikation diese streichen. Die Werte des „port“-Elements kann sie problemlos beachten, solange es sich nicht um Bereiche handelt. Wenn ein Bereich viele Angaben umfasst, müsste die Anwendung eventuell ein Vielfaches derer Anzahl an neuen „access“-Elementen erstellen. Dadurch erhöht sich dementsprechend unnötigerweise die Dateigröße des Widgets, weshalb eine proprietäre Erweiterung die sinnvollere Lösung darstellt.

4.1.3. Inhalt

Bis auf die Dateien, die CSS enthalten, zu welchen CSS-Dateien und HTML-Dateien mit enthaltenem CSS-Code zählen, muss das Konvertierungstool lediglich alle Inhaltsdateien in das neue Widget-Paket kopieren. Im CSS-Code muss es alle „-o-widget-mode“-Aufrufe durch „view-mode“ ersetzen und die darauf folgenden Ansichtsmodi durch die entsprechenden W3C-Werte ersetzen: „widget“ → „floating“, „application“ → „windowed“, „fullscreen“ → „fullscreen“ und „docked“ → „minimized“. Zusätzlich sollte der Konvertierungsvorgang die Dateien mit der Ersatz-JavaScript-API inklu-

[1] http://tools.ietf.org/html/rfc3330

dieren. JavaScript-Hilfsfunktionen, die nur die Aufrufe umwandeln, sind „originURL“, „preferenceForKey“, „setPreferenceForKey“ und eventuell „openURL“. Die Methode „originURL“ löst die Hilfsfunktion in die Eigenschaft „widget.id“ auf. Die Funktionen zum Laden und Speichern von Nutzereinstellungen sollten jeweils zum Aufruf ihrer entsprechenden Methoden aus der W3C-Spezifikation führen. Allerdings muss bei „setPreferenceForKey“ zunächst eine Vertauschung der Parameter stattfinden, da bei Opera der erste Parameter den neuen Wert der Präferenz repräsentiert. Beim W3C-Format stellt dieser hingegen den Identifikator des Schlüssel-Wert-Paares dar. Die Methode „openURL“ stammt aus einer frühen Version der W3C-Spezifikation und wurde aus dieser aufgrund von Sicherheitsbedenken entfernt. Das W3C empfiehlt den Aufruf durch ein „a“-Element mit „href“-Attribut im HTML-Teil des Widgets zu ersetzen. Allerdings ist dies in automatischer Form nicht möglich. Ein „a“-Element hinterlegt immer ein anderes DOM-Element mit einem Verweis. Auf dieses muss zunächst ein Klick erfolgen um den Verweis zu öffnen. Da der Aufruf der „openURL“-Funktion allerdings an kein DOM-Element gebunden ist, müsste ein Autor zunächst eines zum Binden des Links auswählen. Das W3C bietet außerdem die Nutzung von „window.open()“ als Alternative an. Die Applikation sollte auf diese zurückgreifen, da das Einbinden einer simplen Hilfsfunktion zur automatischen Konvertierung genügt.

Größere Probleme bereiten die restlichen Methoden und Eigenschaften der Opera-API, da es keinen Ersatz für diese im W3C-Format gibt. Der Aufruf „widget.identifier“ gibt bei Opera-Widgets die ID der Widget-Instanz zurück. Auch im W3C-Format erhalten die Widget-Instanzen einen Identifikator. Allerdings ist dieser den Widgets selbst nicht bekannt. Deshalb wäre zur Bereitstellung dieser Funktion mindestens ein Feature für die Widget-Umgebung notwendig. Die Eigenschaft „widget.widgetMode“ existierte früher unter dem Namen „currentMode“ in der W3C-Spezifikation [4]. Um sie für die konvertierten Widgets verfügbar zu machen, wurde eine JavaScript-Funktion programmiert. Diese basiert auf der recht neuen „matchMedia“-Methode des „window“-Objektes. Als Parameter nimmt sie eine Zeichenkette mit einer CSS-Medienanfrage entgegen und gibt darauf ein „MediaQueryList“-Objekt zurück. Dieses besitzt die Methode „matches“, welche zurückgibt, ob die Bedingungen der Medienanfrage zutreffen. Die JavaScript-Hilfsfunktion wandelt den mit „matchMedia“ ermittelten Anzeigemodus im Anschluss in einen passenden Opera-Modus um, damit keine Änderungen am JavaScript-Code des Widgets nötig sind. Da die „matchMedia“-Funktion noch nicht von allen Browsern unterstützt ist, existiert zusätzlich eine weitere auskommentierte Implementierungsvariante. Diese erstellt zunächst ein neues „div“-Element, welches nicht sichtbar ist. Danach ändert sie die Größe des Elements mittels einer CSS-Medienanfrage. Wenn sie im Anschluss eine veränderte Größe am „div“ feststellt, weiß sie, dass der Fall der übergebenen Medienanfrage eintrifft.

Das Implementieren der „show“- und der „hide“-Methode sind in einer W3C-Widget-Umgebung ohne das Einführen eines Features nicht realisierbar, da in einer solchen

der Container die Kontrolle über den Ansichtsmodus hat. Die Eigenschaften „onshow“ und „onhide“ sind normalerweise an die beiden Methoden gebunden und in dieser Form für W3C-Widgets somit auch nicht sinnvoll. Allerdings könnte der Container die damit verbundenen Events beim Wechsel in den Ansichtsmodus „minimized“ und zurück auslösen. Diese Zustandsänderung ist jedoch von der Widget-Umgebung abhängig und es gibt aktuell keine Standardisierung dafür. Deshalb ist eine Unterstützung der beiden Eigenschaften an das Bereitstellen eines Features gebunden. Die „getAttention“-Methode ist schon im Opera-Format von der Plattform abhängig und das Gewinnen der Nutzeraufmerksamkeit nach Wunsch realisierbar. Die Funktion beeinflusst die grundsätzliche Funktionalität des Widgets nicht und kann somit auch ein leerer Funktionsrumpf sein.

4.2. Konvertierung von iGoogle-Gadgets

4.2.1. Metainformationen

In Tabelle 4.3 sind alle Metadaten aus den Attributen des „ModulePrefs“-Elementes eines iGoogle-Gadgets zu passenden Attributen und Elementen des W3C-Formats zugeordnet. Alle Metainformationen, die keine äquivalente Repräsentation im W3C-Widget erhalten können, sind durch eine proprietäre Erweiterung eingefügt. Sie sind durch die Abkürzung „prop.“ kenntlich gemacht. Die einzige durch ein XML-Element repräsentierte Metainformation „Icon“ konvertiert die Applikation in ein „icon“-Element. Wenn es sich um ein „Icon“-Element ohne „mode“-Attribut handelt, muss sie die Datei lediglich herunterladen, in das Widget-Paket kopieren und dem „href“-Attribut eine Referenz auf diese zuweisen. Zusätzlich kann die Anwendung noch die Auflösung des Bildes ermitteln und die Attribute „width“ und „height“ mit den herausgefundenen Werten anfügen. Beim „base64“-Modus muss sie hingegen zuerst eine neue Datei aus den base64-kodierten Daten erstellen. Die Dateiendung ermittelt sie durch den Wert des „type“-Attributs. Anschließend kopiert sie die Datei, setzt die Referenz und optional zusätzlich die Auflösung des Bildes durch ein neues „icon“-Element.

Die Attribute „screenshot“ und „thumbnail“ konvertiert das Tool jeweils in ein neues eigenes XML-Element. Die beiden neuen Elemente „screenshot“ und „thumbnail“ gehören dem durch die proprietäre Erweiterung definierten Namensraum an. Sie besitzen jeweils ein „src“-Attribut. Dieses enthält eine relative URL zum Verweis auf die entsprechende Bilddatei. Das „thumbnail“-Attribut des iGoogle-Gadgets besitzt in der Tabelle 4.3 zwei Zeilen, da es zusätzlich als weiteres Icon für das W3C-Widget eingesetzt werden kann. Bei iGoogle spezifiziert ein „Icon“-Element meist ein 16*16 Pixel großes Bild. Im W3C-Format treten dagegen häufig mehrere „icon“-Elemente auf, wobei Referenzen auf sowohl kleinere Icons als auch auf Dateien mit größerer

iGoogle-Format	**W3C-Format**		
Name	**Name**	**Typ**	**Elternelement**
Title	-	Textknoten	name
title_url	url	Attribut (prop.)	name
directory_title	directory	Attribut (prop.)	name
description	-	Textknoten	description
Author	-	Textknoten	author
author_email	email	Attribut	author
author_location	location	Attribut (prop.)	author
author_affiliation	affiliation	Attribut (prop.)	author
author_photo	photo	Attribut (prop.)	author
author_aboutme	aboutme	Attribut (prop.)	author
author_link	href	Attribut	author
author_quote	quote	Attribut (prop.)	author
screenshot	src	Attribut (prop.)	screenshot (prop.)
thumbnail	src	Attribut (prop.)	thumbnail (prop.)
thumbnail	src	Attribut	icon

Tabelle 4.3.: Zuordnung der Metainformationen der Attribute des „ModulePrefs“-Elementes vom iGoogle-Format zum W3C-Format

Auflösung auftreten. Das Ziel ist es, durch unterschiedlich große Bilder möglichst allen Anforderungen unterschiedlicher Widget-Umgebungen zu genügen. Um somit auch ein Icon größerer Auflösung bereitzustellen, kann die Applikation das Thumbnail des iGoogle-Gadgets nutzen. Sie sollte allerdings in diesem Fall bei allen „icon“-Elementen die Attribute „width“ und „height“ mit den jeweiligen Breiten und Höhen der Bilder angeben.

4.2.2. Konfigurationsdaten

Alle Überführungen von Konfigurationsdaten eines iGoogle-Gadgets in das W3C-Format sind in Tabelle 4.4 aufgeführt. Proprietäre Erweiterungen sind erneut durch die Abkürzung „prop.“ angezeigt. Sämtliche Elemente und Attribute für die Konfiguration der OAuth-Authentifizierung und das Vorladen von Ressourcen erfahren keine Umwandlung. Für die Authentifizierung eines W3C-Widgets mittels OAuth gibt es noch keine Spezifikation. Eine derartige Lösung benötigt allerdings die Unterstützung der Widget-Umgebung und ist somit nicht durch ein Widget alleine lösbar. Außerdem können für die Authentifizierung nötige Daten bei Google selbst hinterlegt sein, auf welche man folglich keinen Zugriff hat, wodurch die automatisierte Konvertierung dieser unmöglich ist [17]. Da die Zahl der Gadgets mit Verwendung von OAuth im

Vergleich zum Rest der Widgets gering ist, soll in dieser Arbeit keine Betrachtung der Umwandlung dieser folgen. Das Vorladen von Ressourcen entlastet den iGoogle-Proxy-Server und beschleunigt deshalb das Nachladen von Daten der Gadgets. In der W3C-Widget-Spezifikation ist kein Proxy-Server erwähnt. In Apache Wookie benötigen die Widgets hingegen einen Proxy für Ajax. Wenn dieser Server einen Cache und eine Schnittstelle für das Vorladen von Ressourcen besitzt, kann das Umwandeln dieser Möglichkeit von iGoogle-Gadgets Sinn ergeben.

iGoogle-Format			**W3C-Format**		
Name	**Typ**	**Eltern-element**	**Name**	**Typ**	**Eltern-element**
width	Attribut	ModulePrefs	width	Attribut	widget
preferred_width	Attribut	Content	width	Attribut	widget
height	Attribut	ModulePrefs	height	Attribut	widget
preferred_height	Attribut	Content	height	Attribut	widget
Require	Element	ModulePrefs	feature	Element	widget
Optional	Element	ModulePrefs	feature	Element	widget
feature	Attribut	Require oder Optional	name	Attribut	feature
Param	Element	Require oder Optional	param	Element	feature
-	Textknoten	Param	value	Attribut	param
name	Attribut	Param	name	Attribut	param

Tabelle 4.4.: Auflistung der Zuordnungen der Konfigurationsdaten von iGoogle-Gadgets zu denen eines W3C-Widgets

Die Werte für die Attribute „height“ und „weight“ des W3C-Formats sind zunächst von den gleichnamigen Attributen des „ModulePrefs“-Elements abhängig. Sie stellen eine allgemeine Größenangabe für das Gadget dar. Allerdings sind sie selten benutzt. Deshalb betrachtet die Anwendung bei deren Abwesenheit die Attribute „preferred_width“ und „preferred_height“. Diese geben jedoch die Größe für einen speziellen Ansichtsmodus an. Da sich die Definition der Größe bei W3C-Widgets vor allem auf die verkleinerten Modi bezieht, sollte das Tool zunächst die Werte für den „widget“-Inhalt nutzen. Falls kein derartiges Element existiert bzw. dieses die entsprechenden Attribute nicht besitzt, sollte es die Angaben des „Content“-Elementes für den „default“-Modus verwenden. Im Falle, dass die Suche nach Werten wieder erfolglos war, kann die Applikation jedes „preferred_width“- und „preferred_height“-Attribut unabhängig vom Ansichtsmodus betrachten. Ist sie auch danach nicht fündig geworden, bleiben nur die Möglichkeiten der Verwendung einer Nutzereingabe oder eines Standardwertes. Ansonsten müssen die Attribute „width“ und „height“ ungenutzt bleiben.

Die Elemente „Require“ und „Optional“ integriert die Anwendung durch Bereitstellen der entsprechenden API-Funktionalitäten mittels einer JavaScript-Datei. Sobald eines der beiden Elemente auftritt, überprüft sie den Wert von „feature“ mit der Liste der unterstützten Features. Falls eine Ersatz-API bereitsteht, stellt sie diese dem Widget natürlich zur Verfügung. Wenn das entsprechende Feature dagegen nicht unterstützt ist, hängt das Vorgehen vom jeweiligen Element ab. Handelt es sich um eine optionale Zusatzfunktionalität, kann das Tool dieses einfach ignorieren oder als „feature“-Element mit dem Wert „false“ für das Attribut „required“ übernehmen. Bei einem unbedingt benötigten Feature sollte es entweder ein W3C-„feature“-Element zur Konfigurationsdatei hinzufügen oder die Konvertierung abbrechen. Die Parameter muss es je nach Vorgehensweise als „param“-Elemente oder direkt im JavaScript-Code übernehmen.

Die Funktionalität des „view“-Attributs des „Content“-Elements übernimmt das neue W3C-Widget indirekt durch die Auswahl des anzuzeigenden Inhalts in Abhängigkeit zu einer JavaScript-Funktion, die den aktuellen Ansichtsmodus ermittelt. Deshalb ist eine Übertragung des Attributs in die Konfigurationsdaten des W3C-Formats unnötig. Allerdings benötigt die JavaScript-Funktion eine Zuordnung der iGoogle-Ansichtsmodi zu denen der W3C-Spezifikation. Bei W3C-Widgets gibt es keinen Demonstrationsmodus, weshalb „preview“ keine äquivalente Darstellung in diesem findet. Der nächstbeste passende Modus ist „windowed“. Der Wert „profile“ entspricht „home“ bei iGoogle-Gadgets. Beide löst die Applikation wie „preview“ in „windowed“ auf. Explizit sollte sie durch die dreifache Belegung aber nur den „home“-Ansichtsmodus aufrufen. Der „canvas“-Modus entspricht am ehesten „maximized“.

4.2.3. Inhalt

Die Anwendung nutzt für die Darstellung des Inhalts in Abhängigkeit zu den Ansichtsmodi ein „iframe“-Element. Sie speichert jeweils den Inhalt für einen speziellen Modus in einer eigenen HTML-Datei. Dazu verwendet sie ein vorgefertigtes HTML-Code-Grundgerüst, welches Elemente wie „html“, „head“ und „body“ bereits besitzt. In den „body“ der Datei fügt sie den jeweiligen Inhalt anschließend ein. Das Grundgerüst kann zusätzlich JavaScript-Code bzw. Verweise auf solchen enthalten, welcher dem Widget Funktionalitäten bereitstellt. Beim Laden des Widgets ermittelt zunächst eine JavaScript-Funktion aus dem Grundgerüst den aktuellen Darstellungsmodus. In Abhängigkeit zu diesem wird der Verweis des „src“-Attributs auf das HTML-Dokument für den festgestellten Modus gesetzt. Der Vorteil des eingebetteten Frames im Gegensatz zur „innerHTML“-Lösung ist vor allem, dass der JavaScript standardmäßig zur Ausführung kommt. Für die „innerHTML“-Variante ist eine Funktion nötig, die aus dem einzufügenden Inhalt alle „script“-Elemente extrahiert und ausführt. Bei Gadgets, die den „url“-Typ besitzen, füllt das Tool das „body“-Element

des Grundgerüsts mit einem weiteren eingebetteten Frame. Dessen URL entspricht dem mittels „href“-Attribut definierten Wert.

4.2.4. Nutzereinstellungen

Während der Umwandlung des iGoogle-Gadgets übernimmt die Applikation zunächst die in den „UserPref“-Elementen definierten Nutzereinstellungen in die Konfigurationsdaten des W3C-Widgets. Für jedes „UserPref“-Element erstellt sie genau ein „preference“-Element. Den Wert des Attributs „name“ ordnet sie dem gleichnamigen Attribut des W3C-Formats zu und „default_value“ überführt sie in das „value“-Attribut. Die unterschiedliche Behandlung der verschiedenen Datentypen findet über den JavaScript-Code statt. In den Konfigurationen werden die Präferenzen alle in Form einer Zeichenkette repräsentiert. Bei Listen entsteht der String durch die Konkatenation der einzelnen Listenelemente mit einem „|“-Zeichen. Die JavaScript-Methoden können, wie auch der Großteil der restlichen API, von Apache Shindig übernommen werden. Gerade bei den Funktionen für die Präferenzen sind allerdings einige Modifikationen nötig, damit diese die Methoden des „widget“-Objektes der W3C-API verwenden. Im JavaScript-Code folgt auch die Umwandlung von der Zeichenkettendarstellung in die entsprechenden Datentypen.

Das Nutzereinstellungsmenü erstellt die Anwendung für das W3C-Format. Sie fügt an den Anfang sämtlichen Inhalts des „body“-Tags der HTML-Datei einen leeren „div“-Bereich. Dieser ist der Platzhalter für das Menü. Danach liest sie alle „UserPref“-Elemente aus und erweitert in Abhängigkeit zum jeweiligen Typ den HTML-Code, welcher URL-kodiert den Wert einer Variable darstellt. Weiterhin erstellt sie einen „div“-Bereich, welcher eine in einem weiteren „div“ geschachtelte Bilddatei enthält, die dem Betrachter einen Link zum Einstellungsmenü in Form eines Icon-Symbols visualisiert. Das äußere „div“-Element ist absolut in der rechten oberen Ecke des Widgets positioniert und besitzt einen erhöhten „z-index“, sodass es den restlichen Inhalt überdeckt. Außerdem besitzt es eine feste Größe. Der innere „div“-Bereich ist standardmäßig durch die CSS-Eigenschaft „visibility“ mittels dem Wert „hidden“ ausgeblendet. Bewegt der Nutzer seine Maus über das äußere „div“-Element, zeigt dessen „onmouseover“-Handler das innere „div“ mit dem Bild an. Beim Verlassen des Bereichs durch die Maus ändert sich die „visibility“ wieder zurück zu „hidden“. Wenn der Betrachter einen Klick in den inneren „div“-Bereich ausführt, füllt der „onclick“-Handler das als Platzhalter eingesetzte „div“-Element mit dem dekodierten Wert der Variable. Somit präsentiert sich dem Nutzer das Einstellungsmenü. Die einzelnen Präferenzen zeigt dieses dem Betrachter durch die gleichen HTML-Elemente wie iGoogle an. Eine Ausnahme bildet lediglich die „list“-Einstellung. Sie wird wie der „string“-Typ als „input“-Element mit dem Wert „text“ für das „type“-Attribut dargestellt. Der Nutzer muss zum Ändern dieser Einstellung eine „|“-separierte Zeichenkette be-

arbeiten. Das Menü besitzt zu den Elementen zum Tätigen der Einstellungen zwei „input“-Elemente des Typs „button“. Mit einem der beiden speichert der Benutzer die durchgeführten Änderungen. Durch den anderen kann er die Modifikationen der Werte verwerfen. Bei beiden Buttons schließt sich anschließend das Einstellungsmenü durch Leeren des „div“-Elementes. Wenn der Nutzer seine Änderungen abspeichert, löst der JavaScript zusätzlich ein Neuladen des Widget-Inhalts aus. Ist bei mindestens einer der Präferenzen der Wert des „required“-Attributs „true“ und kein Startwert für dieselbe definiert, löst der JavaScript sofort nach Laden des Widgets automatisch einen Klick auf das „div“-Element zum Öffnen des Einstellungsmenüs aus. Weiterhin lässt sich bei Vorhandensein einer Präferenz mit dem Wert „true“ für das „required“-Attribut das Menü nur bei Vorhandensein eines Wertes für diese schließen.

4.2.5. Lokalisierung

Die Lokalisierung der Metainformationen und Konfigurationsdaten findet je nach Element durch die ordnerbasierte Lokalisierung oder das „xml:lang“-Attribut statt. Für jedes Konvertieren eines Attributwertes oder Textknotens muss das Tool den Wert erst auf die Existenz einer Substitutionsvariable prüfen. Falls eine derartige vorhanden ist, muss es diese durch die entsprechende Übersetzung austauschen. Für nicht lokalisierbare Werte verwendet es das unter Kapitel 3.5 beschriebene Verfahren zur Auswahl der Standardübersetzung. Das „xml:lang“-Attribut nutzt die Applikation für die Attribute „title“, „title_url“, „directory_title“ und „description“. Für die Bilddateien, der für die „icon“-Elemente konvertierten Attribute bzw. Elemente, wendet sie die ordnerbasierte Lokalisierung an. Für die Dateien der Attribute „screenshot“ und „thumbnail“ für die proprietäre Erweiterung nutzt sie im Moment die Standardübersetzung. Sie könnte für diese aber auch die ordnerbasierte Lokalisierung einsetzen.

Den Inhalt passt die Anwendung durch die ordnerbasierte Lokalisierung an. Zuerst sammelt sie alle in den „Locale“-Elementen definierten Kombinationen aus Region und Sprache und erstellt für jede einen neuen Ordner im „locales“-Verzeichnis. In die neuen Ordner kopiert sie jeweils die Dateien für den Inhalt, welche eine Anpassung durch Lokalisierung erhalten. Die Substitutionsvariablen ersetzt sie für jede Datei direkt durch den Wert der entsprechenden Übersetzungsnachricht, wobei sie das Rückfallverhalten nachbildet. Bei der JavaScript-Methode „getMsg“ ist ein direktes Austauschen gegen einen Wert zur Umwandlungszeit nicht möglich, da sich der Parameter erst zur Laufzeit ergeben kann. Deshalb integriert das Tool während der Konvertierung ein JavaScript-Wörterbuch in eine ordnerlokalisierte Datei. Die Datenstruktur besitzt jeweils die Identifikatoren der Übersetzungen als Schlüssel und die Übersetzung als Wert. Für das Erstellen des Wörterbuchs sammelt die Applikation zunächst die Identifikatoren aller Lokalisierungen. Im Anschluss sucht sie für jede

Lokalisierung für jeden Schlüssel eine Nachricht. Dazu betrachtet sie zuerst genau die Sammlung der Übersetzungen für die aktuelle Kombination aus Sprache und Region. Anschließend nutzt sie falls vorhanden die Generalisierung und danach die Standardlokalisierung. Falls sie immer noch nicht fündig war, kann sie jedes angegebene Übersetzungspaket nach einem Wert zum aktuellen Schlüssel durchsuchen oder eine Fehlermeldung ausgeben bzw. in eine Logdatei schreiben oder für den Wert eine leere Zeichenkette verwenden. Damit es zu keinem Problem im JavaScript-Code durch ungültige Zeichen kommt, wendet die Anwendung auf jeden Wert für ein Wörterbuch-Paar vor dem Eintragen die URL-Kodierung an. Beim Laden einer Übersetzung muss die Zeichenkette wieder dekodiert werden. Die JavaScript-Funktion „getMsg“ benötigt natürlich noch eine Anpassung, damit sie die Werte zum gegebenen Schlüssel aus dem Wörterbuch lädt.

In die ordnerlokalisierte JavaScript-Datei, welche das Wörterbuch für die Übersetzungen enthält, fügt das Tool weiterhin die Schreibrichtung, die Sprache und die Region mittels einer Variable ein. Die Werte der aktuellen Sprache und Region benötigt die JavaScript-API. Die Schreibrichtung fügt es für das Setzen des „dir“-Attributs des Inhalts hinzu. Der Wert für die Variable ergibt sich aus dem „language_direction“-Attribut des zugehörigen „Locale“-Elements. Das Grundgerüst der HTML-Inhaltsdatei enthält eine Anweisung, welche zur Ausführung kommt, sobald das Widget geladen ist. Sie weist dem „dir“-Attribut des „body“-Elements den Wert dieser Variablen zu.

4.3. Konvertierung von UWA-Widgets

4.3.1. Metainformationen

Alle Überführungen von Metainformationen des UWA-Widgets in das W3C-Format sind in Tabelle 4.5 aufgelistet. Proprietäre Erweiterungen sind durch die Abkürzung „prop.“ gekennzeichnet. Bei den „meta“-Elementen gibt die Spalte „Name“ der „UWA-Format“-Spalte nicht den Attributnamen, sondern den Wert des Attributs „name“ an.

Die „meta“-Elemente des Typs „screenshot“ oder „thumbnail“ bilden im W3C-Widget jeweils ein neues Element, welches dem Namensraum einer proprietären Erweiterung angehört, wobei dieses den Namen des „name“-Attributs der Quelle besitzt. Es besitzt das Attribut „src“. Dieses enthält den Verweis auf die zugehörige lokale Datei. Der Wert des normalen „title“-Elementes aus dem Kopfteil des XHTML-Dokumentes stellt den Wert für den Textknoten des „name“-Elementes des W3C-Widgets. Von den „link“-Elementen des UWA-Formats betrachtet die Applikation nur diejenigen, welche auf ein Icon bezogen sind. Dies erkennt sie durch das „rel“-Attribut, welches den Wert „icon“ tragen muss. Für alle auf ein Icon bezogenen „link“-Elemente erstellt

UWA-Format			W3C-Format		
Name	**Typ**	**Eltern-element**	**Name**	**Typ**	**Eltern-element**
author	Attribut	meta	-	Textknoten	author
email	Attribut	meta	email	Attribut	author
website	Attribut	meta	href	Attribut	author
description	Attribut	meta	-	Textknoten	description
version	Attribut	meta	version	Attribut	widget
screenshot	Attribut	meta	src	Attribut (prop.)	screenshot (prop.)
thumbnail	Attribut	meta	src	Attribut (prop.)	thumbnail (prop.)
keywords	Attribut	meta	keywords	Attribut (prop.)	widget
-	Textknoten	title	-	Textknoten	name
href	Attribut	link	src	Attribut	icon

Tabelle 4.5.: Zuordnung der Metainformationen vom UWA-Format zum W3C-Format

sie jeweils ein neues „icon“-Element in der Konfigurationsdatei, welches auf die lokale Version des mit „href“ referenzierten Bildes verweist. Zusätzlich kann sie den Attributen „width“ und „height“ die Breite bzw. Höhe des Icons zuweisen.

4.3.2. Konfigurationsdaten

Die Angabe zur „apiVersion“ ist ausschlaggebend dafür, ob die Anwendung das Widget konvertieren kann oder nicht. Wenn die angegebene Versionsnummer höher als die der durch die Umwandlung bereitgestellten API ist, bricht sie den Vorgang mit einem Warnhinweis darauf ab. Die anderen beiden Konfigurationsdaten muss sie nicht beachten, da sie diese einfach in der Widget-Datei, welche beim W3C-Format zur Inhaltsdatei wird, unberührt belässt. Die Konfigurationsinformationen darf sie nicht entfernen, da diese vom JavaScript-Code nötig sind. Dieser liest die „meta“-Elemente zur Laufzeit aus und beeinflusst sein Verhalten durch die Werte.

4.3.3. Inhalt

Die Inhaltsdatei des Widgets ergibt sich aus der gesamten UWA-Widget-Datei. An der Datei selbst findet nur ein Anpassen der Verweise auf die Bild- und die Standalone-Dateien statt. Die CSS-Datei benötigt vor allem eine Anpassung der Verweise

auf andere Dateien, wie zum Beispiel sämtliche durch CSS-Regeln eingefügten Bilder für die Gestaltung. Wenn hingegen von allen referenzierten Dateien und auch von deren Referenzen lokale Versionen existieren, ist eine Änderung der relativen Pfade nicht mehr nötig. Dafür müssen eventuell vorhandene absolute Verweise in relative Angaben getauscht werden. Zusätzlich sind geringe Änderungen an den CSS-Regeln zur besseren Darstellung notwendig. Dies betrifft vor allem die äußere Hülle, welche die Darstellung des Widgets im Browser auf eine bestimmte Fläche begrenzt. In einer Widget-Umgebung sollte das Widget hingegen die gesamte ihm zur Verfügung gestellte Fläche verwenden. Die benötigten Änderungen an der JavaScript-Standalone-API entstehen hauptsächlich durch die Umgebungsbedingungen für ein W3C-Widget. Es ist allerdings auch eine Modifikation nötig, da es standardmäßig beim Standalone-Modus zu Anzeigeproblemen bei Präferenzen des Typs „list“ kommt [28]. Weiterhin erfordert vor allem der benötigte Proxy für Ajax in Apache Wookie eine Veränderung des JavaScript-Codes. Außerdem ist eine Anpassung für die Nutzung der lokalen Version des Icons nötig.

4.3.4. Nutzereinstellungen

Sowohl die JavaScript-API als auch das Nutzereinstellungsmenü stellen die Standalone-Dateien bereit. Somit muss das Tool das Menü im Gegensatz zum iGoogle-Format nicht selbst zusammenstellen. Es muss lediglich die entsprechenden Präferenz-Konfigurationsdaten in die Inhaltsdatei des W3C-Widgets übernehmen. Allerdings kommt es in Apache Wookie bei den „option“-Elementen von Nutzereinstellungen des Typs „list“ zu Problemen. Diese übernimmt es beim Laden des Widgets nicht in den DOM-Baum. Das führt jedoch zu Problemen bei der Darstellung der Präferenz im Einstellungsmenü. Deshalb benennt die Applikation alle „option“-Elemente innerhalb von „preference“-Elementen in „preferenceoption“ um. Diese übernimmt Apache Wookie in den DOM-Baum. Dadurch kann sie der JavaScript-Code verarbeiten. Allerdings erfordert es eine weitere Anpassung des Standalone-JavaScripts, da dieser nun nach anderen Elementen suchen muss. Ein weiteres Problem ist, dass trotz Nutzung des „defaultValue“-Attributs der Startwert dieser keinen Wert erhält. Es tritt im Standalone-Modus bei Nutzereinstellungen des Typs „list“ auf. Dies erfordert eine zusätzliche Modifikation der JavaScript-Standalone-Datei. Sie löst nach Erstellen des Einstellungsmenüs für jede Präferenz des Typs „list“ die Methode „setValue“ mit dem Startwert aus. Weiterhin könnte eine Anpassung der Nutzereinstellungen erfolgen, dass diese die W3C-Präferenzen verwenden und nicht nur auf den Standalone-Methoden basieren.

4.4. Zusammenfassung

In diesem Kapitel wurden die Umwandlungsdetails für die drei ausgesuchten Formate betrachtet. Es zeigten sich dabei einige Probleme und Verbesserungsmöglichkeiten. Eine Analyse der Verbesserungsansätze erfolgt im Kapitel 6. Eine Untersuchung der Auswirkung der Probleme auf die Konvertierung schließt sich im folgenden Kapitel an. Dazu wird ein Test des im Rahmen der Arbeit entwickelten Prototyps durchgeführt. Er untersucht mehrere Widgets der unterschiedlichen Formate in der umgewandelten Form auf die Funktionalität, um die Auswirkungen der Probleme festzustellen.

5. Evaluation

Nachdem eine Implementierung des Widget-Formatwandlers erfolgt ist, soll in Kapitel 5.1 eine Bewertung des Prototyps stattfinden. Dazu wurden zuerst von allen drei Formaten eine bestimmte Menge an Widgets umgewandelt und anschließend in Apache Wookie auf Funktionalität getestet. Das Kapitel 5.2 behandelt eine Bewertung der Konvertierung durch den Prototypen in Bezug auf die im Stand der Technik festgestellten Anforderungen.

Zunächst folgen zur Visualisierung des Prototypen zwei Screenshots. Die Hauptansicht, welche direkt nach dem Start der Applikation angezeigt wird, ist in Abbildung 5.1 dargestellt. Der Nutzer muss im Textfeld den Pfad zu einer Widget-Datei für die einzelne Konvertierung oder den Pfad zu einer Konfigurationsdatei für die Stapelverarbeitung angeben. Die Abbildung 5.2 zeigt ein Beispiel einer Konfigurationsdatei.

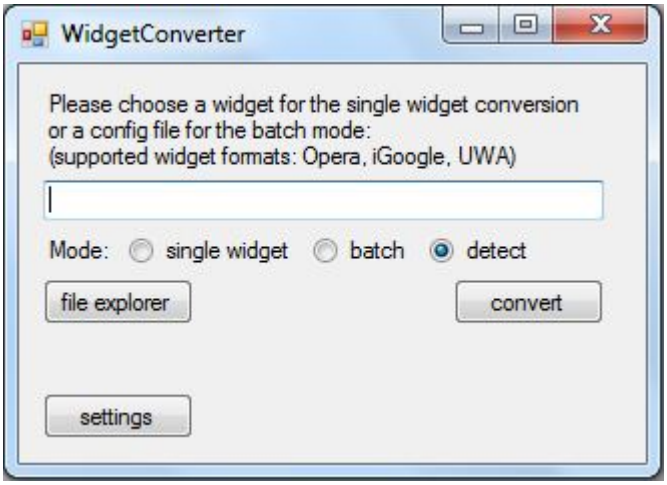

Abbildung 5.1.: Hauptansicht des Prototypen

5.1. Test des Konvertierungs-Tools

Der Test des Prototypen betrachtet 25 Opera-Widgets, 25 iGoogle-Gadgets und 25 UWA-Widgets im exportierten Opera-Format. Er verwendet jeweils die bestbewer-

```
#this is an example config file for the widget converter
#open this configuration in the widget converter to use it for the conversion

#Set the path to the folder which contains the widgets for the conversion
widgetinput=C:\Users\Michael\Desktop\widgetInput

#Set the folder search option for the file for the conversion: 'TopDirectoryOnly' or 'AllDirectories'
searchoption=AllDirectories

#Set the path to the folder for the converted widgets
widgetoutput=C:\Users\Michael\Desktop\widgetOutput

#Set to true to overwrite the files of the output folder with the converted widgets if a name clash occurs
widgetoutputoverwrite=true

#Set the default widget width
widgetwidth=400

#Set the default widget height
widgetheight=400

#Set to false to not delete the working directory and its files for the converted widgets (optional)
widgetworkdelete=true
```

Abbildung 5.2.: Beispiel-Konfigurationsdatei für die Stapelverarbeitung

teten oder aber meist heruntergeladenen Widgets. Der Test basiert auf dem Modus für die Konvertierung mehrerer Widgets am Stück.

Beim Opera-Format nutzt der Test die ersten 25 funktionierenden Widgets der Kategorie höchste Bewertung[1] mit Stand vom 13.07.2012. Zum Sammeln dieser war eine Betrachtung von insgesamt 35 Opera-Widgets nötig, da zehn der 35 Widgets selbst in der Widget-Umgebung des Opera-Browsers zu keiner Anzeige führten. Insgesamt scheinen ungünstigerweise relativ viele Opera-Widgets Bugs zu besitzen. Bei Betrachtung der Detailinformationen zu einem Widget im Opera-Store[2] fallen bei etwa jedem zweiten gemeldete Probleme auf. Die Opera-Widgets scheinen relativ schlecht gewartet zu sein. Die Applikation wandelte von den 25 übergebenen 21 Widgets erfolgreich in das W3C-Format um. Zwei konnte sie aufgrund von JavaScript-API-Aufrufen nicht konvertieren, für welche sie eine Umwandlung nicht unterstützt und somit keine Ersatzfunktion bereitstellen kann. Außerdem waren zwei der Widgets nach der Opera-Spezifikation nicht gültig. Diese sind in der Opera-Umgebung hingegen vollständig lauffähig, weshalb der Nutzer erwartet, dass sie auch konvertierbar sind. Bei beiden war die Konfigurationsdatei nicht valide. Von den 21 erfolgreich umgewandelten Widgets funktionierten nur 13 unter Apache Wookie. Drei zeigten unter Wookie aufgrund von JavaScript-Problemen keine Funktionalität. Probleme ergaben sich vor allem durch den benötigten Zugriff auf lokale Dateien des Nutzers und durch opera-spezifische Funktionen. Weiterhin kam es bei einem zu leichten Anzeige- und Funktionsfehlern, welche in der Opera-Umgebung nicht auftraten. Zwei der 21 Wid-

[1] http://widgets.opera.com/rated/
[2] http://widgets.opera.com/

gets funktionierten wegen kleinen Bugs von Apache Wookie im Test nicht und sind höchstwahrscheinlich grundsätzlich in Form der konvertierten Ausgabe funktional. Die letzten beiden waren nur im Opera-Browser nutzbar. In anderen Browsern kam es wieder aufgrund opera-spezifischer JavaScript-Methoden zu Problemen. Somit ergibt sich im Test eine Quote von 13 funktionierenden aus 25 Widgets. Dies entspricht 52 % für die in Apache Wookie nutzbaren konvertierten W3C-Widgets.

Für das iGoogle-Format verwendet der Test die ersten 25 Treffer aus dem iGoogle-Widget-Store[3] (Stand: 13.07.2012) mit Ausnahme der in iGoogle eingebauten Gadgets, da für diese keine Quelldateien öffentlich zur Verfügung stehen. Die Reihenfolge der Widgets im Store entspricht zum Großteil einer Sortierung nach der Nutzerzahl. Allerdings sind einige Gadgets dabei, die sich nicht in die Reihe einordnen lassen. Sie besitzen für ihre Position entweder zu viele oder zu wenige Benutzer. Bei iGoogle gibt es keine Auswahl zur Sortierung der Einträge. Die genauen Sortierungskriterien sind deshalb auch nicht erkennbar. Von den 25 Widgets konnte das Tool alle erfolgreich konvertieren. Allerdings funktionierten nur 23 der 25 Gadgets unter Apache Wookie. Bei einem trat ein Problem im JavaScript-Code auf. Das andere nicht funktionierende Widget ließ sich aufgrund eines Bugs von Apache Wookie nicht in dessen Widget-Store einfügen. Dieser tritt auf, wenn der Startwert für eine Präferenz eine bestimmte Länge überschreitet. Er ist bereits beim Wookie-Entwicklungsteam gemeldet und sollte bei späteren Versionen nicht mehr auftreten. Wenn vor der Umwandlung des betroffenen Gadgets der Startwert der entsprechenden Nutzereinstellung gekürzt wird, führt die Konvertierung dessen zu einem unter Apache Wookie funktionierendem Widget. Insgesamt ergab der Test von 25 Widgets 23 ordnungsgemäß arbeitende, was einer Quote von 92 % entspricht. Die Anzeige eines Beispiel-Widgets im iGoogle-Format in der Umgebung von iGoogle ist im oberen Teil der Abbildung 5.3 dargestellt. Das zugehörige automatisch aus der Konvertierung entstandene W3C-Widget in der Apache Wookie-Umgebung zeigt diese unterhalb des Pfeils.

Der in dieser Arbeit entwickelte Prototyp bietet zwei Möglichkeiten für die Konvertierung eines Widgets von Netvibes. Die Applikation kann diese sowohl im originalen UWA-Format als auch im exportierten Opera-Format verarbeiten. Für den Umwandlungstest der UWA-Widgets wurde das exportierte Opera-Format genutzt, da der Aufwand zum Herunterladen geringer im Gegensatz zur anderen Möglichkeit ist. Aus dem Netvibes Ecosystem[4] sind lediglich zwei Klicks zum Downloaden der Opera-Datei nötig. Außerdem ist das Ergebnis dieser Variante tendenziell erfolgreicher, da der sonst verwendete Standalone-Modus laut Netvibes keine perfekte Nachbildung ist und Fehler enthalten kann. Zudem erfolgt in den Prototypen keine Abbildung der Nutzereinstellungen auf die W3C-Präferenzen für das originale UWA-Format. Der Test bestand aus den 25 meistgenutzten UWA-Widgets (Stand: 13.07.2012), wobei

[3] http://www.google.de/ig/directory
[4] http://de.eco.netvibes.com/

für die Auswahl die Ländereinstellung auf Deutschland gesetzt war. Die Anwendung wandelte alle Test-Widgets erfolgreich in das W3C-Format um. Von den konvertierten Widgets funktionierten 22 in Apache Wookie. Der Grund dafür, dass drei nicht verwendbar sind, scheint seinen Ursprung vor allem im Proxy-Server von Wookie zu haben. Dieser führt auch manchmal dazu, dass sonst funktionierende konvertierte Widgets kurzfristig unfunktional sind. Das Ergebnis 22 von 25 ergibt eine Quote von 88 % für das UWA-Format.

5.2. Bewertung

In Bezug auf die drei betrachteten Widget-Formate ergibt sich für das Kriterium Angebot eine sehr gute Bewertung. Für alle drei Formate existieren jeweils Portale, in denen mehrere tausende Widgets zum Herunterladen bereitstehen. Es ist somit kein Suchen auf anderen Webseiten nach diesen notwendig. Der Aufwand für die Umwandlung ist zunächst vom Zusammensuchen der zu konvertierenden Widgets abhängig. Unter der Voraussetzung, dass die Widget-Dateien bereits gesammelt sind, ist der Aufwand vom gewählten Modus des Prototyps abhängig. Bei der Umwandlung eines einzelnen Widgets muss der Nutzer den Vorgang für jedes einzeln starten. Außerdem kann zusätzliche Nutzerinteraktion nötig sein. Dementsprechend ist der Aufwand höher. Bei der Konvertierung mehrerer Widgets am Stück ist nur ein einziges Auslösen des Vorgangs notwendig. Der Aufwand ist dadurch sehr gering. Die Anpassung an eine Widget-Umgebung ist aufgrund der Tatsache, dass es sich bei den Eingangsdaten bereits um Widgets handelt, vorhanden. Des Weiteren steht der Prototyp zur sofortigen Verwendung bereit, ohne dass Änderungen nötig sind. Durch Modifikationen kann allerdings die Funktionalität der Implementierung erweitert werden. Die Ergebnisse sind zusammenfassend in der Tabelle 5.1 aufgelistet. Im Vergleich zu den anderen bewerteten Verfahren schneidet das verwendete Verfahren am besten ab. Allerdings ergibt sich aus den Quoten, dass stets eine Person ein konvertiertes Widget vor der Veröffentlichung von Hand auf Funktionalität prüfen sollte.

Eingabedaten	**Angebot**	**Aufwand**	**Anpassung**	**Verwendbarkeit**
Opera-Widget, iGoogle-Gadget, UWA-Widget	+ +	O/+ +	+ +	+ +
+ + sehr gut, + gut, O teilweise, - unzureichend, - - nicht gegeben				

Tabelle 5.1.: Bewertung des Prototyps zur W3C-Widget-Generierung

5.3. Zusammenfassung

Nachdem in diesem Kapitel eine Bewertung des Konvertierungstools anhand eines Tests mit unterschiedlichen Widgets und anhand der in Kapitel 2.1 ausgearbeiteten Bewertungskriterien erfolgt ist, soll sich zuletzt eine abschließende Zusammenfassung der Arbeit und ihrer Ergebnisse anschließen. Außerdem findet eine Betrachtung der Möglichkeiten zur Erweiterung und Verbesserung des Prototypen statt.

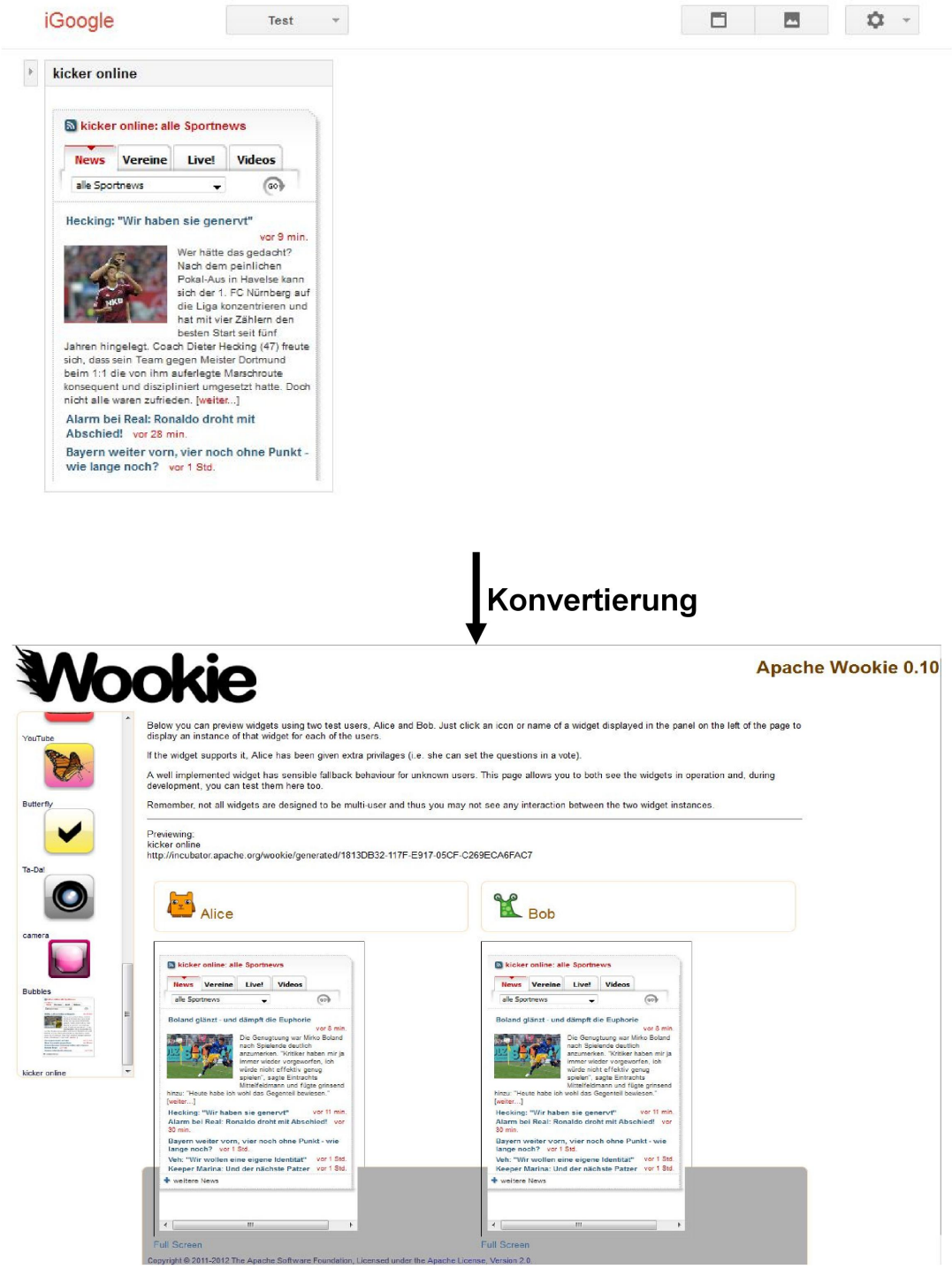

Abbildung 5.3.: iGoogle-Gadget in der iGoogle-Umgebung und nach erfolgter Umwandlung in der Wookie-Umgebung

6. Ausblick

Im Rahmen der Arbeit wurden verschiedene Verfahren zur automatisierten Generierung von W3C-Widgets untersucht und bewertet, da das Angebot für dieses Format aktuell sehr gering ist. Die Arbeit verwendet als Vorgehen die Konvertierung von anderen Widget-Formaten, weil sich dieses bei der Untersuchung als die am vielversprechendste Möglichkeit herausstellte. Im Anschluss wurden die Grundbestandteile eines Widgets betrachtet. Daraufhin stellt die Arbeit das W3C-Format detailliert vor und ordnet dessen Komponenten in die Grundbestandteile ein. In Bezug auf dieses schließt sich eine Betrachtung der drei Widget-Formate Opera-Widget, iGoogle-Gadget und UWA-Widget an. Danach stellt die Arbeit die grundlegenden Schritte für die Umwandlung eines Widgets in das W3C-Format vor. Die Grundbestandteile ordnet sie den einzelnen Prozessen zu. Anschließend wurden das Konzept für die Konvertierung der drei Widget-Formate in das W3C-Format herausgearbeitet und aus diesem ein Prototyp entwickelt. Bei der Betrachtung stellten sich auch einige Probleme der Umwandlung von Widgets heraus. Schließlich folgt eine Evaluation des Prototyps und eine Bewertung des Verfahrens. Bis auf das Opera-Format ergaben sich annehmbare Quoten für die lauffähigen im Gegensatz zu den unter Apache Wookie nicht funktionierten generierten Widgets. Allerdings stellte sich heraus, dass der Benutzer ein konvertiertes Widget stets vor der Veröffentlichung in der Ziel-Widget-Umgebung auf Funktionalität prüfen sollte. Während der Ausarbeitung ergaben sich auch einige Möglichkeiten zu Erweiterung des Prototyps zur Verbesserung des Ergebnis-Widgets und der Quote. Im Folgenden sind einige Ansätze zur Ergänzung der Lösung aufgeführt.

Eine Möglichkeit zur allgemeinen Erweiterung des Prototyps ist das Unterstützen weiterer Formate für die Konvertierung. Dabei kann der Entwickler einige der bereits vorhandenen Methoden und auch die Klasse für die W3C-Konfigurationsdatei weiterverwenden. Durch die Unterstützung neuer Eingangs-Formate erhöht sich das Angebot an neuen W3C-Widgets zusätzlich. Zur Erweiterung bieten sich auch Dateien an, bei denen es sich nicht um Widgets handelt. Das im Kapitel 2.4 kurz genannte „crx“-Dateiformat stellt eine Möglichkeit dafür dar. Der Entwickler kann für die Umwandlung der Anwendungen aus dem Google Chrome Web Store die bereits vorhandene Applikation von Scott Wilson weiterverwenden. Allerdings liegt diese als JavaScript vor, während der Prototyp in C# programmiert ist, weshalb zuerst eine Konvertierung des Codes nötig wäre. Eine weitere allgemeine Erweiterungsmöglichkeit ist eine Unterstützung von URLs als Eingabeparameter. So muss der Benutzer

die umzuwandelnden Widgets nicht erst herunterladen. Außerdem kann das Tool dann mit relativen URLs innerhalb des Widgets, welche sich auf Dateien außerhalb des Widget-Pakets beziehen, umgehen, da es den Kontext kennt. Des Weiteren kann es im Widget vorkommende URLs umwandeln, welche die Zeichenkette „//“ zur Verwendung des gleichen Protokolls in Bezug auf ihre Umgebung nutzen.

Die weiteren aufgeführten Erweiterungsvorschläge beziehen sich jeweils auf die Verbesserung der Konvertierung eines aktuell unterstützten Formats. Für Opera-Widgets kann ein Entwickler die Quote durch Nachprogrammierung weiterer JavaScript-Funktionen erhöhen. Zusätzlich wäre ein Nachbilden diverser opera-spezifischer Funktionalitäten mittels allgemeinen JavaScript-Codes für die Erhöhung der Quote sehr förderlich. Bei iGoogle-Gadgets kann ein Entwickler die Quote durch Aufsuchen weiterer undokumentierter API-Funktionen und die Bereitstellung von Hilfsfunktionen für diese für das Tool, welches sie in die neuen W3C-Widgets einbindet, erhöhen. Außerdem kann er zusätzlich nicht dokumentierte Metainformationen als proprietäre Erweiterungen für die W3C-Konfigurationsdatei verwenden, um über das Widget eine bessere Metabeschreibung zu erhalten. Ein anderer Ansatz zur Verbesserung der Quote ist das Unterstützen der iGoogle-OAuth-Gadgets. Dafür kann der Entwickler das vorhandene Wookie-Feature für OAuth nutzen. Allerdings ist das Widget daraufhin nur in W3C-Umgebungen lauffähig, die dieses Feature besitzen. Eine Verbesserung der Nutzeroberfläche eines konvertierten iGoogle-Gadgets kann ein Autor durch eine genauere Nachbildung von Präferenzen des Typs „list“ im Einstellungsmenü erzielen. Außerdem kann er bei Bedarf eine ordnerbasierte Lokalisierung für die per proprietärer Erweiterung eingeführten XML-Elemente „screenshot“ und „thumbnail“ einführen. Bisher erfolgt bei diesen nur eine Übernahme der Bilddateien für die Standardlokalisierung. Für die Konvertierung von Widgets im UWA-Format könnte der Entwickler zur Verbesserung die Standalone-Dateien modifizieren, dass sie für die Nutzereinstellungen das W3C-Präferenz-Objekt verwenden. Dadurch erfolgt eine persistente Speicherung der Daten gebunden an eine spezielle Widget-Instanz. Eine weitere Erweiterung könnte die Applikation dahingehend modifizieren, dass die neuen W3C-Widgets nur lokale Versionen der Standalone-Dateien nutzen, was auch schon im Kapitel 4.3.3 aufgeführt ist.

Literaturverzeichnis

[1] Behrens, Heiko: *Cross-Platform App Development for iPhone, Android & Co. - A Comparison I Presented at MobileTechCon 2010*. 2010. – URL http://heikobehrens.net/2010/10/11/cross-platform-app-development-for-iphone-android-co-%E2%80%94-a-comparison-i-presented-at-mobiletechcon-2010/. – Zugriffsdatum: 21.08.2012

[2] Berjon, Robin: *AppCache to Widget Converter (ac2wgt) - Robin Berjon*. 2009. – URL http://berjon.com/hacks/ac2wgt/. – Zugriffsdatum: 05.07.2012

[3] Berjon, Robin ; Cáceres, Marcos: *The 'view-mode' Media Feature*. 2012. – URL http://www.w3.org/TR/view-mode/. – Zugriffsdatum: 07.07.2012

[4] Bersvendsen, Arve ; Cáceres, Marcos: *Widgets 1.0: APIs and Events*. 2009. – URL http://www.w3.org/TR/2009/WD-widgets-apis-20090205/#the-currentmode-attribute. – Zugriffsdatum: 19.07.2012

[5] Cáceres, Marcos: *Widget Packaging and XML Configuration*. 2011. – URL http://www.w3.org/TR/widgets/. – Zugriffsdatum: 02.07.2012

[6] Cáceres, Marcos: *Widget Interface*. 2012. – URL http://www.w3.org/TR/widgets-apis/. – Zugriffsdatum: 07.07.2012

[7] Cáceres, Marcos ; Byers, Paddy ; Knightley, Stuart ; Hirsch, Frederick ; Priestley, Mark: *XML Digital Signatures for Widgets*. 2011. – URL http://www.w3.org/TR/widgets-digsig/. – Zugriffsdatum: 07.07.2012

[8] Conjecture Corporation: *What is a Widget?*. – URL http://www.wisegeek.com/what-is-a-widget.htm. – Zugriffsdatum: 02.07.2012

[9] Daniel, Florian ; Matera, Maristella: Turning Web Applications into Mashup Components: Issues, Models, and Solutions. In: *Policy* 5648 (2009), S. 45–60. – URL http://dx.doi.org/10.1007/978-3-642-02818-2_4. ISBN 9783642028175

[10] Google: *Gadgets zu Ihrer Startseite hinzufügen*. – URL http://www.google.de/ig/directory?dpos=top&root=/ig. – Zugriffsdatum: 16.07.2012

[11] Google: *Development Fundamentals - Gadgets API - Google Developers*. 2012. – URL https://developers.google.com/gadgets/docs/fundamentals. – Zugriffsdatum: 17.07.2012

[12] Google: *Gadgets and Internationalization (i18n) - Gadgets API - Google Developers*. 2012. – URL https://developers.google.com/gadgets/docs/i18n. – Zugriffsdatum: 18.07.2012

[13] Google: *Gadgets API Reference - Gadgets API - Google Developers*. 2012. – URL https://developers.google.com/gadgets/docs/reference/. – Zugriffsdatum: 16.07.2012

[14] Google: *Gadgets XML Reference - Gadgets API - Google Developers*. 2012. – URL https://developers.google.com/gadgets/docs/xml_reference. – Zugriffsdatum: 16.07.2012

[15] Google: *iGoogle Developer's Guide - iGoogle - Google Developers*. 2012. – URL https://developers.google.com/igoogle/docs/igoogledevguide?hl=de-DE. – Zugriffsdatum: 16.07.2012

[16] Google: *Was geschieht mit iGoogle? - Websuche-Hilfe*. 2012. – URL http://support.google.com/websearch/bin/answer.py?hl=de&answer=2664197. – Zugriffsdatum: 16.07.2012

[17] Google: *Writing OAuth Gadgets - Gadgets API - Google Developers*. 2012. – URL https://developers.google.com/gadgets/docs/oauth. – Zugriffsdatum: 17.07.2012

[18] Hemetsberger, Paul: *dict.cc | widget | Wörterbuch Englisch-Deutsch*. – URL http://www.dict.cc/?s=widget. – Zugriffsdatum: 02.07.2012

[19] Kennedy, Niall: *Widget nomenclature | Niall Kennedy*. 2007. – URL http://www.niallkennedy.com/blog/2007/09/widget-nomenclature.html. – Zugriffsdatum: 02.07.2012

[20] Kennedy, Niall: *Widget timeline*. 2007. – URL http://www.niallkennedy.com/blog-asides/timelines/widgets/. – Zugriffsdatum: 02.07.2012

[21] Laszlo Systems: *Chapter 1. OpenLaszlo Architecture*. – URL http://www.openlaszlo.org/lps4.9/docs/developers/architecture.html. – Zugriffsdatum: 05.07.2012

[22] Laszlo Systems: *OpenLaszlo Application Developer's Guide*. – URL http://www.openlaszlo.org/lps4.9/docs/developers/. – Zugriffsdatum: 05.07.2012

[23] Mills, Chris: *Opera Widgets specification 1.0, fourth edition - Dev.Opera*. 2010. – URL http://dev.opera.com/articles/view/opera-widgets-specification-fourth-ed/. – Zugriffsdatum: 08.07.2012

[24] MIT Museum: *MIT Project Athena, 1983-1991 | The MIT 150 Exhibition*. 2011. – URL http://museum.mit.edu/150/26. – Zugriffsdatum: 02.07.2012

[25] Netvibes: *All apps for Netvibes, Mac, Google*. – URL http://de.eco.netvibes.com/all-apps/any. – Zugriffsdatum: 15.07.2012

[26] Netvibes: *Netvibes : Developers*. – URL http://dev.netvibes.com/. – Zugriffsdatum: 05.07.2012

[27] Netvibes: *Netvibes Documentation - uwa [Documentation]*. 2010. – URL http://documentation.netvibes.com/doku.php?id=uwa. – Zugriffsdatum: 19.07.2012

[28] Netvibes: *Anatomy of a UWA widget [doc]*. 2012. – URL http://dev.netvibes.com/doc/uwa/documentation/anatomy_of_a_uwa_widget. – Zugriffsdatum: 18.07.2012

[29] Netvibes: *Netvibes REST API [doc]*. 2012. – URL http://dev.netvibes.com/doc/api/rest. – Zugriffsdatum: 19.07.2012

[30] Netvibes: *UWA JavaScript Framework [doc]*. 2012. – URL http://dev.netvibes.com/doc/uwa/documentation/javascript_framework. – Zugriffsdatum: 19.07.2012

[31] Opera Software: *Web Specifications Supported in Opera 9*. – URL http://www.opera.com/docs/specs/opera9/. – Zugriffsdatum: 03.07.2012

[32] Opera Software: *Opera Widgets Specification 1.0 - Dev.Opera*. 2007. – URL http://dev.opera.com/articles/view/opera-widgets-specification-1-0/. – Zugriffsdatum: 08.07.2012

[33] TARR: *Latest Status Info*. – URL http://tarr.uspto.gov/servlet/tarr?regser=serial&entry=75945338. – Zugriffsdatum: 02.07.2012

[34] Teigene, Arnstein: *Opera Add-ons - Increased focus on Opera extensions and ending support for Unite applications and Widgets*. 2012. – URL http://my.opera.com/addons/blog/2012/04/24/sunsetting-unite-and-widgets. – Zugriffsdatum: 03.07.2012

[35] Wilson, Scott: *Converting Chrome Installed Web Apps into W3C Widgets | Scott's Workblog*. 2011. – URL http://scottbw.wordpress.com/2011/02/17/

converting-chrome-installed-web-apps-into-w3c-widgets/. – Zugriffsdatum: 05.07.2012

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Chemnitz, den 4. September 2012

Michael Hertel

Anhang

Anhang A.

Aufbau des W3C-Widget-Formates

Beim W3C-Widget-Format handelt es sich um eine Sammlung von Dateien aus denen das Widget besteht, welche in einem Zip-Archiv zu einer Datei gepackt sind. Als Zeichenvorrat ist für alle Dokumente des Widgets UTF-8 empfohlen. Die Dateiendung des Zip-Archivs sollte „.wgt“ entsprechen. Als Speichermethoden für die Zip-Datei sind das Kompressionsverfahren Deflate und das Speicherverfahren Stored, bei welchem keine Kompression stattfindet, erlaubt. Im Archiv sind Ordner und eine Verteilung der Dateien des Widgets in diesen Ordnern erlaubt. Es gibt allerdings ein paar reservierte Namen, welche nur spezielle Dateien benutzen dürfen. Die reservierten Dateinamen sind in der Tabelle A.1 aufgelistet. In der linken Spalte ist der Name angegeben und in der Mitte der Verwendungszweck für welchen der Name reserviert ist. Die rechte Spalte gibt den MIME-Typ (Multipurpose Internet Mail Extensions) der Datei an. Die letzte Zeile der Tabelle enthält eine Grammatik anstelle eines Namens. Alle mit dem Startsymbol „dateiname“ erzeugbaren Namen sind für den angegebenen Zweck reserviert.

Die Konfigurationsdatei config.xml muss in jedem W3C-Widget enthalten sein. Sie enthält die Konfigurationsdaten und die Metainformationen. Die Standard-Icon-Dateien müssen hingegen nicht zwangsweise im Zip-Archiv enthalten sein. Sie ermöglichen das Hinzufügen von Icons für das Widget, ohne dass Verweise auf diese mittels der Konfigurationsdatei vorhanden sind. Die Startdatei eines W3C-Widgets dient als Einstiegspunkt für den Inhalt. Jedes Widget im W3C-Format muss eine gültige Startdatei enthalten. Zum Definieren einer Startdatei gibt es zwei Möglichkeiten, ähnlich dem Verhalten der Icons. Die Auswahl der Startdatei findet entweder über einen reservierten Namen oder mittels Verweis auf diese in der Konfigurationsdatei statt. Der „locales“-Ordner dient der Lokalisierung des Widgets. Die reservierten Dateinamen für die Signaturen werden für das digitale Signieren des Widget-Paketes benötigt. Die „author-signature.xml“-Datei enthält die Signatur des Autors des Widgets und die anderen Signatur-Dateien enthalten jeweils die Signatur eines Vertreibers für das Widget. Durch das digitale Signieren eines Widgets kann ein UA feststellen, welche Entität ein Autor hat, ob mehrere Widgets wirklich vom gleichen Autor stammen, ob ein bestimmter Anbieter wirklich der Vertreiber des Widgets ist, ob mehrere Widgets wirklich vom gleichen Vertreiber stammen und ob die Integrität des Widgets

Dateiname	**Verwendungszweck**	**MIME-Typ**
config.xml	Konfigurationsdatei	application/xml
icon.png	Standard-Icon	image/png
icon.gif	Standard-Icon	image/gif
icon.jpg	Standard-Icon	image/jpeg
icon.ico	Standard-Icon	image/vnd.microsoft.icon
icon.svg	Standard-Icon	image/svg+xml
index.html	Standard-Startdatei	text/html
index.htm	Standard-Startdatei	text/html
index.svg	Standard-Startdatei	image/svg+xml
index.xhtml	Standard-Startdatei	application/xhtml+xml
index.xht	Standard-Startdatei	application/xhtml+xml
locales	Ordner für die ordnerbasierte Lokalisierung	
author-signature.xml	Signatur des Autors	application/xml
dateiname ::= „signature“ ziffer-ohne-null ziffer* „.xml“ ziffer ::= „0“ | ziffer-ohne-null ziffer-ohne-null ::= „1“ | „2“ | „3“ | „4“ | „5“ | „6“ | „7“ | „8“ | „9“	Signatur des Widget-Vertreibers	application/xml

Tabelle A.1.: Auflistung der reservierten Dateinamen

gewahrt ist. Damit ein Widget digital signiert ist, müssen natürlich nicht nur die Signatur-Dateien im Widget-Paket enthalten sein, sondern auch alle Nicht-Signatur-Dateien aus dem Widget-Archiv digital signiert werden. Details über das Signieren von W3C-Widgets sind über die Seite[1] des W3Cs abrufbar. [5, 7]

Die Konfigurationsdatei muss eine XML-Datei mit einem „widget“-Element als Wurzelknoten sein. Außerdem muss das Dokument den Widget-Namensraum „http://www.w3.org/ns/widgets“ innerhalb des „widget“-Elementes als Wert des Attributs „xmlns“ enthalten. Die Elemente und Attribute des Namensraumes sind in dieser Arbeit in die Bereiche Konfigurationsdaten und Metainformationen aufgeteilt. Falls die Möglichkeiten des Widget-Namensraumes nicht genügen, können Ergänzungen der Konfigurationsdatei mittels einer proprietären Erweiterung stattfinden. Für das Hinzufügen eigener Elemente und Attribute zum XML-Dokument muss zunächst ein neuer Namensraum definiert sein. An den neu spezifizierten Namensraum bindet der Autor einen Präfix. Danach kann der Entwickler nach Bedarf eigene Elemente und Attribute in der Konfigurationsdatei verwenden, wenn er diese durch das Verwenden

[1] http://www.w3.org/TR/widgets-digsig/

des Präfixes in den Namensraum setzt. Bei der Standard-Verarbeitung der Konfigurationsdaten werden die Elemente mit einem Präfix einfach ignoriert. Im Anhang E befindet sich eine Beispiel-Konfigurationsdatei für ein W3C-Widget. [5]

A.1. Metainformationen

Die Tabelle A.2 enthält alle zum W3C-Format gehörigen Metainformationen für ein Widget. In der Auflistung sind alle Attribute und Elemente benannt, welche in den Bereich der Metadaten einzuordnen sind. Bei Elementen zeigt die Spalte „Elternelement“ das Element an, welches das angegebene XML-Element als Kind besitzen kann. Bei Attributen gibt die Spalte „Elternelement“ das Element an, welchem sie zugeordnet werden können. Die Elemente „name“, „description“, „license“, „icon“ und „span“ dürfen jeweils mehrfach als Kind des gleichen Elternelementes auftreten.

Name	**Typ**	**Eltern-element**	**Bedeutung**
id	Attribut	widget	Identifikator für das Widget
version	Attribut	widget	Version des Widgets
name	Element	widget	Name des Widgets
short	Attribut	name	Kurze Variante des Namens des Widget
description	Element	widget	Beschreibung des Widgets
author	Element	widget	Person oder Organisation, welche das Widget erstellt hat
href	Attribut	author	IRI, welche den Autor assoziiert
email	Attribut	author	E-Mail-Adresse, welche den Autor assoziiert
license	Element	widget	Software-Lizenz unter welcher das Widget angeboten wird
href	Attribut	license	IRI oder relativer Pfad, welche eine Software-Lizenz referenziert
icon	Element	widget	Spezifiziert ein nicht Standard-Icon für das Widget
href	Attribut	icon	Relativer Pfad zu einer Datei die das Icon darstellt
width	Attribut	icon	Vom Autor präferierte Darstellungsbreite des Icons
height	Attribut	icon	Vom Autor präferierte Darstellungshöhe des Icons

Name	Typ	Eltern-element	Bedeutung
span	Element	name, author, description, license	Hülle für Text ohne eigene Funktion, benötigt für die Lokalisierung

Tabelle A.2.: Auflistung aller Elemente und Attribute aus dem Widget-Namensraum, welche Metainformationen tragen

Bei der ID eines W3C-Widgets muss es sich stets um einen validen Internationalized Resource Identifier (IRI) handeln. Der Identifikator ermöglicht das Unterscheiden von Widgets mit gleichem Name. Des Weiteren verhindert er eine doppelte Aufnahme eines Widgets in einen Widget-Store. Das Namenselement enthält den vollständigen, menschenlesbaren Namen des Widgets als Textknoten. Mittels dem „short“-Attribut kann zusätzlich eine Kurzversion des Namens für Kontexte, in welchen der Platz für diesen beschränkt ist, angegeben werden. Die Beschreibung des Widgets hinterlegt der Autor als Kind des „description“-Elementes als Textknoten. Auch beim „author“-Element wird ein Textknoten als Kind genutzt. Dieser enthält den Namen der Person oder Organisation, welche das Widget erstellt hat. Beim „license“-Element kann der Textknoten die Lizenzinformationen direkt als Klartext enthalten. Die Alternative ist das Setzen eines Verweises auf eine Datei mittels dem „href“-Attribut, die eine Repräsentation der Lizenzinformationen enthält. Durch das Verwenden des „icon“-Elementes kann ein Autor dem Widget Icons ohne Einschränkung des Speicherorts oder Dateinamens hinzufügen. Dabei ist zu beachten, dass bei Verwendung eines „icon“-Elementes stets das „href“-Attribut vorhanden sein muss. Das „width“- und das „height“-Attribut ermöglichen dem Autor das Angeben einer präferierten Anzeigegröße des Icons. Die Werte stellen CSS-Pixel dar. Der Autor muss Zahlen, die größer als null sind, hinterlegen. Die Größenangaben dienen vor allem der Widget-Umgebung zur Auswahl des am besten passenden Icons, wenn mehrere im Archiv vorhanden sind. Die Funktion des „span“-Elementes ist im Abschnitt A.5 erläutert. [5]

A.2. Konfigurationsdaten

Die Tabelle A.3 enthält alle im W3C-Format vorhandenen XML-Elemente und XML-Attribute für die Konfiguration des Widgets. Bei Elementen zeigt die Spalte „Elternelement“ das Element an, welches das angegebene XML-Element als Kind besitzen kann. Bei Attributen gibt die Spalte „Elternelement“ das Element an, welchem sie zugeordnet werden können. Die Elemente „feature“ und „param“ dürfen jeweils mehrfach als Kind des gleichen Elternelementes auftreten.

Name	**Typ**	**Eltern-element**	**Bedeutung**
height	Attribut	widget	Präferierte Darstellungshöhe des Widgets
width	Attribut	widget	Präferierte Darstellungsbreite des Widgets
viewmodes	Attribut	widget	Präferierter Anzeigemodus für das Widget
content	Element	widget	Spezifiziert die Startdatei des Widgets
src	Attribut	content	Relativer Pfad zur Startdatei
type	Attribut	content	Medientyp der Startdatei
encoding	Attribut	content	Kodierung der Startdatei
feature	Element	widget	Anfordern von Features
name	Attribut	feature	IRI, welche das Feature identifiziert
required	Attribut	feature	Angabe, ob ein Feature für das Widget unbedingt benötigt wird
param	Element	feature	Spezifiziert einen Parameter für ein Feature
name	Attribut	param	Name des Parameters
value	Attribut	param	Wert des Parameters

Tabelle A.3.: Auflistung aller Elemente und Attribute aus dem Widget-Namensraum, welche Konfigurationsdaten tragen

Die Attribute „width“ und „height“ spezifizieren die gewünschte Ansichtsfenstergröße für das Widget. Die Werte müssen bei Verwendung des entsprechenden Attributs in CSS-Pixeln angegeben werden. Es sind nur Zahlen größer null gestattet. Das „viewmodes“-Attribut dient der Angabe der bevorzugten „view-mode“-Werte für das Widget. Die „view-mode“-Werte entsprechen den Werten des „view-mode“-Medien-Features. Die fünf möglichen Zustände sind „windowed“, „floating“, „fullscreen“, „maximized“ und „minimized“. Der Wert „windowed“ bedeutet, dass das Widget nicht die volle Anzeigefläche des UA ausnutzt und einen Anzeigerahmen besitzt. Der Modus „floating“ unterscheidet sich von „windowed“ nur dadurch, dass er keinen Anzeigerahmen besitzt und der Hintergrund des Widgets transparent angezeigt wird. Bei „fullscreen“ handelt es sich um einen Modus, bei welchem kein Anzeigerahmen vorhanden ist und das Widget außer der gesamten Anzeigefläche zusätzlich noch den Platz um die Anzeigefläche für das Anzeigen des Inhalts nutzen kann. Ist der Wert „maximized“ gesetzt, nutzt das Widget die komplette Anzeigefläche und erhält zusätzlich einen Anzeigerahmen. Beim letzten Modus ist vom Widget im Anzeigebereich nur eine kleine dynamische grafische Anzeige vorhanden. Der Autor kann für den Wert des „viewmodes“-Attributs nur einen Modus, aber auch eine komma-separierte Liste an unterschiedlichen Modi angeben. Bei der Verwendung einer Liste ist der erste Wert der am meisten präferierte, der zweite Wert der zweitmeist präferierte usw. [3, 5]

Wenn die Startdatei nicht den Standard-Konfigurationen entspricht, muss ein Verweis auf diese mit dem „content“-Element erfolgen. Das Attribut „src“ des Elements bekommt dazu den relativen Pfad zur Startdatei im Widget-Paket zugewiesen. Durch die Benutzung das „content“-Elementes ist der Autor nicht gezwungen die Datei im Wurzelverzeichnis des Widgets abzulegen. Zusätzlich hat er die Möglichkeit den Medientyp mittels dem „type“-Attribut in Form eines MIME-Wertes anzugeben. Er ist dadurch nicht auf die drei Werte „text/html“, „image/svg+xml“ und „application/xhtml+xml“ beschränkt. Allerdings muss der Autor stets beachten, dass der entsprechende MIME-Typ auch von der gewünschten Widget-Umgebung unterstützt wird. Bei Absenz des „type“-Attributs geht die Widget-Umgebung vom Standardwert „text/html“ aus. Durch das „encoding“-Attribut kann der Autor die Kodierung der Startdatei in den Konfigurationsdaten angeben. Der Wert der Kodierung muss dem Wert einer „name“- oder „alias“-Angabe eines Zeichenvorrates aus der IANA-Zeichensatz-Spezifikation[2] entsprechen. Wenn das Attribut nicht vorhanden ist, nimmt die Widget-Umgebung den Standardwert „UTF-8“ an. Andere Kodierungen, außer „UTF-8“, müssen von den UAs nicht unterstützt werden. [5]

Das „feature“-Element dient dem Anfordern zusätzlicher Funktionalitäten von der Widget-Umgebung. Beim W3C-Widget-Format sind alle Features mit einem IRI identifizierbar. Durch das Anfordern eines Features kann die Widget-Umgebung, vorausgesetzt sie unterstützt das entsprechende Feature, dem Widget zur Laufzeit die entsprechenden Extra-Funktionalitäten zur Verfügung stellen. Ein UA könnte zum Beispiel das OAuth-Authentifizierungsverfahren als Feature anbieten. Das „name“-Attribut des Elementes dient der Angabe der IRI des Features. Es muss bei Verwendung eines Feature-Elementes vorhanden sein und als Wert eine valide IRI zugewiesen haben. Durch das „required“-Attribut ist es dem Autor möglich, die Wichtigkeit des Features für das Widget anzugeben. Setzt er den Wert des Attributs auf „true“, ist die erweiterte Funktionalität für das Widget unbedingt notwendig. Wenn die Widget-Umgebung das entsprechende Feature nicht unterstützt, weist sie das Widget ab. Ist der Wert hingegen auf „false“ gesetzt, so kommt das Widget auch bei Absenz des Features zur Ausführung. Sollte das „required“-Attribut nicht vorhanden sein, geht der UA von dem Wert „true“ aus. Jedes „feature“-Element kann eine beliebige Anzahl an „param“-Elementen enthalten. Diese stellen die Parameter für das entsprechende Feature dar. Bei den Parametern handelt es sich jeweils um Schlüssel-Wert-Paare. Das „name“-Attribut entspricht dem Schlüssel und das „value“-Attribut dem Wert des Paares. Bei Verwendung eines „param“-Elementes müssen stets beide Attribute vorhanden sein. [5]

[2] http://www.iana.org/assignments/character-sets

A.3. Inhalt

Der Inhalt eines W3C-Widgets teilt sich, wie bei einer normalen Webanwendung, in HTML-, CSS und JavaScript-Bestandteile auf. Es gibt keine zusätzlichen Beschränkungen für den Autor. Allerdings bietet die W3C-Umgebung eine kleine JavaScript-API mit erweiternder Funktionalität an. Zum Zeitpunkt des Schreibens befand sich das W3C-Dokument zur Widget-API noch in der Phase der „geplanten Empfehlung“. Auf die zusätzlich abrufbaren JavaScript-Attribute kann ein Autor über das Widget-Interface zugreifen. Den Namen des Widgets kann der Autor zum Beispiel mittels „widget.name“ abrufen. Alle vorhandenen Attribute und deren Rückgabewerte sind in der Tabelle A.4 aufgeführt. Bei jedem Wert ist nur der Lesezugriff erlaubt. [6]

Name des Attributes	**Datentyp**	**Wert**
author	readonly attribute DOMString	Name des Autors
description	readonly attribute DOMString	Beschreibung des Widgets
name	readonly attribute DOMString	Name des Widgets
shortName	readonly attribute DOMString	Kurzform des Widgetnamens
version	readonly attribute DOMString	Version des Widgets
id	readonly attribute DOMString	ID des Widgets
authorEmail	readonly attribute DOMString	E-Mail-Adresse des Autors
authorHref	readonly attribute DOMString	URI, die den Autor assoziiert
height	readonly attribute unsigned long	Aktuelle Höhe des Ansichtsfensters der Widget-Instanz
width	readonly attribute unsigned long	Aktuelle Breite des Ansichtsfensters der Widget-Instanz

Tabelle A.4.: Auflistung aller JavaScript-Attribute der Widget-API

A.4. Nutzereinstellungen

Die Nutzereinstellungen werden beim W3C-Widget-Format mit in der Konfigurationsdatei „config.xml“ eingerichtet. Für sie existiert das „preference“-Element. Dieses kann beliebig oft als Kind des „widget“-Elementes auftreten. Jedes „preference“-Element spezifiziert genau einen Parameter. Dafür hat es die drei Attribute „name“, „value“ und „readonly“, wovon „value“ und „readonly“ optionale Angaben sind. Der „name“ definiert den Namen der Einstellung und dient als Identifikator dieser. Mit dem „value“-Attribut kann der Autor einer Präferenz einen Startwert zuweisen. Durch das „readonly“-Attribut ist eine Beschränkung der Interaktion mit einer

Nutzereinstellung auf das Lesen möglich. Ein Schreibzugriff auf die Präferenz wird bei Setzen des Wertes auf „true“ verwehrt. Sollte der Autor das „readonly“-Attribut nicht benutzen, geht die Widget-Umgebung vom Wert „false“ aus. Der Zugriff auf eine Nutzereinstellung zur Laufzeit findet durch die Benutzung der JavaScript-API statt. Für das Zuweisen eines Wertes zu einer Präferenz gibt es die Methode „widget.preferences.setItem(name, value)“. Der Parameter „name“ bestimmt die auszuwählende Einstellung und „value“ gibt den neu zu setzenden Wert an. Das Abfragen eines Wertes funktioniert analog mit der „widget.preferences.getItem(name)“-Methode. Dabei bestimmt „name“ wieder die entsprechende Präferenz und deren Wert wird als Rückgabewert zurückgegeben. Als Alternative gibt es noch den Zugriff auf eine Nutzereinstellung durch das Anwenden eines Datenfeldzugriffs („widget.preferences[name]“). [5, 6]

A.5. Lokalisierung

Für die Lokalisierung eines W3C-Widgets werden zwei Verfahren zugleich eingesetzt. Die erste genutzte Möglichkeit ist die Verwendung des „xml:lang“-Attributs. Dieses Verfahren findet in der Konfigurationsdatei seine Verwendung. Die Elemente, die ein Autor durch „xml:lang“ lokalisieren kann, sind „name“, „description“ und „license“. Zu beachten ist, dass jeweils nur ein Element mit gleichem Namen pro „xml:lang“-Wert erlaubt ist. Das „xml:lang“-Attribut muss nicht direkt im Element gesetzt sein, denn auch das Vererbungsverhalten eines XML-Dokuments findet Beachtung. Der Wert des Attributs muss ein valides Datum für eine Sprache aus der IANA language subtag registry[3] sein, wobei überholte und redundante Einträge zu vermeiden sind. Um nicht nur sprachspezifisch, sondern auch regionsspezifisch zu lokalisieren, kann der Autor zum Sprach-Tag ein Regions-Tag durch Verbindung mit einem Bindestrich hinzufügen, wie zum Beispiel „en-us“. Alle mit „xml:lang“ beeinflussbaren Elemente können zudem mit dem „dir“-Attribut versehen werden. Mittels diesem kann der Autor die Textrichtung, in welcher der menschenlesbare Text beim UA zur Repräsentation kommt, steuern. Die validen Werte dafür sind in der Tabelle A.5 aufgelistet. Bei Absenz des Attributs geht der UA von „ltr“ aus. Das „span“-Element kann der Autor des Widgets nutzen, um einen Textteil eines Elementes einem anderen „xml:lang“-Wert oder einer anderen Textrichtung zuzuordnen. Dazu legt er ein neues „span“-Kind für das entsprechende Element an, setzt für dieses die „xml:lang“- und/oder „dir“-Attribute und fügt den Textteil als neuen Textknoten an das „span“-Element an. [5]

Die zweite genutzte Möglichkeit ist die ordnerbasierte Lokalisierung, wofür es den reservierten „locales“-Ordner gibt. Dieser bildet den Container für die ordnerbasierte

[3] http://www.iana.org/assignments/language-subtag-registry

Wert	**Ausgeschriebene Schreibweise**	**Bedeutung**
ltr	Left-to-right	Text von links nach rechts geschrieben (nur schwache Zeichen)
rtl	Right-to-left	Text von rechts nach links geschrieben (nur schwache Zeichen)
lro	Left-to-right override	Wie ltr, nur dass auch die stark bidirektionalen Zeichen beeinflusst sind
rlo	Right-to-left override	Wie rtl, nur dass auch die stark bidirektionalen Zeichen beeinflusst sind

Tabelle A.5.: Auflistung aller Werte und deren Bedeutung für das „dir“-Attribut

Lokalisierung. Im „locales“-Ordner legt der Autor für jede Lokalisierung, die er mit dem Widget unterstützen möchte, einen neuen Ordner an. Diesen benennt er nach dem gleichen Verfahren, welches schon für den Wert eines „xml:lang“-Attributwertes eingesetzt wird. Das heißt er nutzt die Werte aus dem IANA language subtag registry und kann mittels Bindestrich zur Sprache noch eine Region anfügen. Der Name des Ordners darf nur Kleinbuchstaben enthalten. Um Dateien zu lokalisieren legt der Autor die für eine bestimmte Kombination aus Land und Region angepasste Datei in das entsprechende Verzeichnis im „locales“-Ordner ab. Er muss dabei allerdings beachten, dass sich die Datei im lokalisierten Ordner im gleichen Kontext wie die unlokalisierte Datei befindet, das heißt die lokalisierte Datei muss den gleichen Dateinamen tragen und sich im gleichen Verzeichnis befinden. Zur Laufzeit tauscht der UA intern die für die aktuelle Lokalisierung vorhandenen Dateien mit den unlokalisierten Varianten aus. Das Verhalten der ordnerbasierten Lokalisierung ist für alle Dateien außer der Konfigurationsdatei im Widget-Archiv nutzbar und somit auch für die Lokalisierung der Icons verwendbar. [5]

Die Auswahl der anzuzeigenden Lokalisierung findet beim UA statt. Ein Zurückfallverhalten ist beim W3C-Format vorhanden. Die Standardlokalisierung gibt der Autor dem Widget durch die Dateien im Wurzelverzeichnis und dessen Unterverzeichnissen außer dem „locales“-Ordner. In der Konfigurationsdatei setzt der Autor die Standardlokalisierung, indem er das „xml:lang“-Attribut nicht verwendet. Er hat jedoch auch die Möglichkeit keine unlokalisierte Edition, sondern nur lokalisierte Versionen des Widgets anzubieten und dann eine Lokalisierung als Standard zu setzen. Dafür verwendet der Autor das „defaultlocale“-Attribut vom „widget“-Element der Konfigurationsdatei. Dieses erhält den Namen einer vorhandenen Lokalisierung des Widgets und wird daraufhin beim Zurückfallverhalten als Standardlokalisierung mit einbezogen. [5]

Anhang B.

Aufbau des Opera-Widget-Formates

Ein Opera-Widget besteht wie auch ein W3C-Widget aus einem Zip-Archiv, welches die Dateiendung „.wgt“ erhalten sollte. In diesem müssen sich mindestens ein Konfigurationsdokument mit dem Namen „config.xml“ und eine Startdatei befinden. Der reservierte Name für die Startdatei ist „index.html“. Außer „config.xml“ und „index.html“ gibt es keine reservierten Dateinamen. Für die Ordnerstruktur im Archiv gibt es beim Opera-Format zwei unterschiedliche Ansätze. Strategie eins entspricht dem Aufbau eines W3C-Widgets. Bei der zweiten Strategie müssen sich hingegen alle Dateien, welche zum Widget gehören, in einem einzigen Ordner befinden, welcher selbst in der Wurzel des Archivs platziert ist. Es sind keine weiteren Ordner in der gesamten Zip-Datei erlaubt. Das einzige Verzeichnis sollte vom Autor am besten den gleichen Namen wie das Archiv erhalten. Das Konfigurationsdokument muss wie beim W3C-Format wohlgeformten XML-Code enthalten und sollte UTF-8 kodiert sein. Das Wurzelelement muss auch den Namen „widget“ tragen. Allerdings ist für das Opera-Konfigurationsdokument kein Namensraum definiert. Die Konfigurationsdatei enthält, wie beim W3C-Widget, alle nötigen Metainformationen und Konfigurationsdaten. Im Anhang F befindet sich ein Beispiel für diese. [23]

B.1. Metainformationen

In der Tabelle B.1 sind alle Metainformationen aus der vierten Version der Opera-Widget-Spezifikation aufgelistet und kurz erklärt. Bei jedem Eintrag des Typs Element ist der entsprechende String mit den Metadaten als Kind-Textknoten an das Element angehangen. Die Elemente „author“ und „id“ sind die einzigen Metainformationsträger, welche keine Textknoten als eines ihrer Kinder besitzen dürfen. [23]

Jedes der Metaelemente ist grundsätzlich optional, außer dem „widgetname“-Element, welches dem Widget einen menschenlesbaren Namen für den Widget-Store gibt. Das „icon“-Element ist das einzige, welches der Autor mehrfach in einer Konfigurationsdatei benutzen darf. Wenn es genutzt wird, muss der Textknoten auf eine Datei zeigen, welche einem der drei unterstützten Icon-Formate angehört. Die drei Formate sind

Name	**Typ**	**Eltern-element**	**Bedeutung**
widgetname	Element	widget	Name des Widgets
author	Element	widget	Person, welche das Widget erstellt hat
name	Element	author	Name des Autors
organization	Element	author	Organisation, welcher der Autor angehört
email	Element	author	E-Mail-Adresse des Autors
link	Element	author	IRI, welche den Autor assoziiert
description	Element	widget	Beschreibung des Widgets
icon	Element	widget	Spezifiziert ein Icon für das Widget
width	Attribut	icon	Präferierte Darstellungsbreite des Icons
height	Attribut	icon	Präferierte Darstellungshöhe des Icons
id	Element	widget	Bildet die Identität eines Widgets
host	Element	id	Host, von welchem das Widget heruntergeladen werden kann
name	Element	id	String, welcher einzigartig an der in „host“ angegebenen Domain ist
revised	Element	id	String in einem der W3C Datums- und Zeitformaten[1], welcher mindestens Monat und Jahr des Datums der Revidierung darstellt

Tabelle B.1.: Auflistung aller Elemente und Attribute des Opera-Widget-Formates für Metadaten

PNG, GIF und SVG. Die Attribute „width“ und „height“ dienen, wie bei der W3C-Widget-Spezifikation, zum Angeben einer präferierten Anzeigegröße für das Icon. Die Werte müssen als positive ganze Zahlen gegeben sein. Bei Benutzung des optionalen „id“-Elementes müssen stets die drei möglichen Kinder „host“, „name“ und „revised“ jeweils genau einmal vorhanden sein. Die Reihenfolge der Kinder ist dabei egal. [23]

B.2. Konfigurationsdaten

Alle für die Konfiguration des Widgets zur Verfügung stehenden Elemente und Attribute sind in der Tabelle B.2 aufgelistet. Die Elemente „width“, „height“, „widgetfile“, „protocol“, „host“, „port“ und „path“ können jeweils einen Textknoten zum Ablegen der Daten besitzen. Alle Elemente und Attribute sind, sofern nicht anders beschrieben, optional. [23]

[1] http://www.w3.org/TR/NOTE-datetime

Name	Typ	Eltern-element	Bedeutung
defaultmode	Attribut	widget	Präferierter Anzeigemodus
dockable	Attribut	widget	Definiert, ob das Widget den „Mikrowidget“-Modus unterstützt
transparent	Attribut	widget	Spezifiziert, ob der Hintergrund des Widgets transparent dargestellt werden darf
network	Attribut	widget	Fordert den Zugriff auf das private und/oder öffentliche Netzwerk an
width	Element	widget	Präferierte Darstellungsweite des Widgets
height	Element	widget	Präferierte Darstellungshöhe des Widgets
widgetfile	Element	widget	Spezifiziert die Startdatei des Widgets
feature	Element	widget	Anfordern von Features
name	Attribut	feature	URI, welche das Feature identifiziert
required	Attribut	feature	Angabe, ob ein Feature für das Widget unbedingt benötigt wird
param	Element	feature	Spezifiziert einen Parameter für ein Feature
name	Attribut	param	Name des Parameters
value	Attribut	param	Wert des Parameters
security	Element	widget	Spezifiziert eine Sicherheitsdeklaration
access	Element	security	Fordert den Zugriff auf eine Ressource an
protocol	Element	access	Protokoll für den Zugriff auf die Ressource
host	Element	access	Host, unter welchem die Ressource liegt
port	Element	access	Port, welchen das Widget für den Ressourcenzugriff nutzt
path	Element	access	Pfad, welcher die Ressource im Host adressiert
content	Element	security	Ermöglicht das Verbieten von Plugins für das Widget
plugin	Attribut	content	Attribut zum Verbieten der Plugins

Tabelle B.2.: Auflistung aller Elemente und Attribute des Opera-Widget-Formates für die Konfiguration des Widgets

Der „defaultmode“ spezifiziert analog zum „viewmodes“-Attribut bei W3C-Widgets den präferierten Anzeigemodus. Valide Werte für dieses Attribut sind „widget“, „application“ und „fullscreen“. Der Wert „widget“ spezifiziert die Darstellung des Widgets ohne einen Anzeigerahmen. Das Widget hat in diesem Modus selbst die Kontrolle über die eigene Darstellungsgröße. Beim Wert „application“ wird die Größe hingegen vom UA bestimmt. Außerdem sollte der UA das Widget mit einem Anzeigerahmen darstellen. Der Anzeigerahmen bei Opera-Widgets entspricht einer Titelleiste bei ei-

nem normalen Programm und ermöglicht die Größenänderung und das Verschieben. Im „fullscreen“-Modus stellt der UA das Widget auf der gesamten zur Verfügung stehenden Anzeigefläche dar und kann bei Bedarf zusätzlich einen Rahmen verwenden. Für den Fall, dass die Darstellung eines Widgets in einem angegeben Modus nicht möglich ist, existiert beim Opera-Format ein Zurückfallverhalten. Die Reihenfolge entspricht dabei „application“, „fullscreen“ und „widget“. Der Standardwert ist „widget“. Mit dem „dockable“-Attribut kann der Autor dem UA den Wechsel in einen vierten Anzeigemodus mit dem Namen „docked“ erlauben bzw. verbieten. Bei diesem beschränkt sich die Darstellung des Widgets auf ein Minimum. Der UA kann beispielsweise eine Titelleiste mit Namen und Status anzeigen. Valide Werte zum Erlauben des Modus sind „yes“, „true“ und „dockable“. Bei allen anderen Werten und bei Abwesenheit ermöglicht der UA keinen Wechsel in den „docked“-Modus. Ein weiteres Attribut zur Beeinflussung der Anzeige des Widgets ist „transparent“. Ist der Wert auf „yes“, „true“ oder „transparent“ gesetzt, stellt der UA den Hintergrund standardmäßig transparent dar. Bei allen anderen Werten erfolgt keine durchsichtige Darstellung des Hintergrundes. Wenn das Attribut nicht vorhanden ist, ist das Verhalten vom Anzeigemodus des Widgets abhängig. Beim „widget“-Modus erfolgt eine transparente Darstellung des Hintergrundes, bei allen anderen Modi nicht. Der UA kann den Wert jedoch jederzeit bedingt durch Anforderungen der Plattform überschreiben. [23]

Mittels der Elemente „width“ und „height“ kann der Autor die präferierte initiale Anzeigegröße des Widgets angeben. Die Werte des Textknotens stellen jeweils einen CSS-Pixel-Wert dar. Der Standardwert für beide Elemente ist 300 und wird vom UA bei einem invalidem Wert oder fehlendem Element verwendet. Mit dem „widgetfile“-Element kann der Autor in Analogie zum „content“-Element des W3C-Widgets eine Startdatei für das Widget auswählen, welches nicht den reservierten Namen benutzen muss. Bei Verwendung muss der Kindknoten einen validen relativen Pfad auf die Datei enthalten. Der Wert darf nicht URL-kodiert sein. Das „feature“-Element und seine „param“-Kindelemente mit ihren Attributen „name“, „required“, „name“ und „value“ haben das gleiche Verhalten wie die entsprechenden Elemente aus der W3C-Spezifikation. [23]

Mithilfe des „network“-Attributes und „security“-Elementes inklusive aller Elemente und Attribute unterhalb von „security“ kann der Autor das Sicherheitsmodell für das Widget konfigurieren. Das Sicherheitsmodell beinhaltet das Anfordern von Zugriff auf Ressourcen an bestimmten Adressen und das Verbieten von Plugins für das Widget. Standardmäßig sind Plugins für den eingebetteten Code, wie zum Beispiel eine Flash-Datei, erlaubt. Bei Verwendung des „plugin“-Attributs innerhalb des „content“-Elementes kann der Autor mit allen Werten, außer „yes“, „true“ und „plugin“, den UA anweisen die Plugins zu deaktivieren. Alle anderen Elemente und Attribute für das Sicherheitsmodell dienen zur Konfiguration des Zugriffs. Mit dem „network“-

Attribut kann der Autor grundlegend spezifizieren, ob das Widget auf Ressourcen des lokalen und/oder öffentlichen Netzwerks zugreifen darf. Valide Werte sind „private“ für das lokale und „public“ für das öffentliche Netz. Die beiden Werte dürfen auch durch ein Komma separiert als Liste verwendet werden, um Zugriff auf beide Netzwerke anzufordern. Bei Abwesenheit des Attributes oder einem invaliden Wert erlaubt der UA keinen Zugriff auf eine Ressource aus irgendeinem Netzwerk. Der Wert „private“ entspricht allen IP (Internetprotokoll)-Adressen, die in den lokalen Rechner aufgelöst werden. Das öffentliche Netzwerk entspricht allen anderen Adressen. Den Wert des „network“-Attributs kann der UA allerdings abhängig von den eigenen Sicherheitseinstellungen lokal überschrieben. Damit ein Widget Zugriff auf eine entfernte Ressource erlangen kann, reicht es nicht aus, dass der Autor nur das „network“-Attribut spezifiziert, da dieses nur einen Netzwerkbereich bestimmt. Für die genauere Angabe der Zugriffsziele existiert das „access“-Element. Dieses arbeitet stets in Verbindung mit dem „network“-Attribut. Das „access“-Element enthält selbst keine Daten, außer einer beliebigen Anzahl der vier möglichen Kinderelemente. [23]

Das Element „protocol“ bestimmt die Protokolle, die der UA für den Zugriff auf entfernte Ressourcen erlauben soll. Der Wert eines jeden „protocol“-Elementes entspricht jeweils dem Namen des Schemas, welches das Widget nutzen möchte. Der Standardwert bei Abwesenheit einer Protokollspezifikation entspricht den drei URI-Schemata „http“, „https“ und „widget“. Sobald mindestens ein „protocol“-Element gesetzt ist, darf der UA nur noch den Zugriff über die spezifizierten Protokolle zulassen. Eine Ausnahme bildet das „widget“-Uri-Schema, auf welches ein Zugriff auch ohne Definition möglich sein sollte. Das „file“-Protokoll ist das einzige nicht erlaubte Protokoll, selbst wenn es der Autor anfordert. Mittels dem „host“-Element ist eine Spezifikation der Hostnamen und IP-Adressen der Zugriffe möglich. Bei Absenz des Elementes geht der UA von einer Anfrage auf alle Hosts aus. Wenn das „host“-Element hingegen vorhanden ist, erlaubt die Widget-Umgebung nur noch den Zugriff auf die spezifizierten Hosts. Dabei ist stets auf die exakte Übereinstimmung zu achten. Der Wert „www.example.org“ entspricht zum Beispiel nicht „example.org“. [23]

Das „port“-Element gibt an, welche Ports das Widget für den Zugriff verwenden möchte. Es gibt drei Möglichkeiten zur Angabe des Wertes des Textknotens. Der Autor kann die gewünschten Ports einzeln, als komma-separierte Liste oder als Bereich definieren. Einen Bereich gibt er durch Angabe des kleinsten Wertes verbunden durch einen Gedankenstrich mit dem größten Wert an. Falls kein „port“-Element vorhanden ist, nimmt der UA die Anfrage nach allen möglichen Ports an. Es ist allerdings zu beachten, dass durch die Standard-Sicherheitsregeln der Widget-Umgebung nie alle Ports zugelassen sind. Mit dem Element „path“ kann der Autor die genutzten Pfade für die Zugriffe spezifizieren. Falls kein „path“-Element vorhanden ist, erlaubt der UA die Nutzung aller Pfade. [23]

Jegliche durch die sicherheitsbezogenen Elemente erstellte Regel kann eingeschränkt oder missachtet werden, falls sie nicht konform mit den Sicherheitseinstellungen des UA ist [23].

B.3. Inhalt

Der Inhalt besteht in Analogie zur W3C-Spezifikation aus HTML, CSS und JavaScript. Im Gegensatz zum W3C-Widget bietet Opera allerdings nicht nur eine Erweiterung des JavaScripts mittels einer API, sondern auch eine Erweiterung von CSS an. Die CSS-Erweiterung ermöglicht dem Autor das Gestalten des Widgets abhängig vom Anzeigemodus. Dafür hat Opera in die Widget-Umgebung das Medien-Feature „-o-widget-mode“ eingeführt. Das Verhalten ist analog zum „view-mode“-Medien-Feature. Die einzigen Unterschiede zwischen den beiden sind der Name und die Werte. Die möglichen Werte beim „-o-widget-mode“-Medien-Feature entsprechen den Namen der vier Anzeigemodi („widget“, „application“, „fullscreen“ und „docked“). [23]

Die JavaScript-API der Opera-Widget-Umgebung ist im Gegensatz zu der API der W3C-Spezifikation etwas umfangreicher. Sie teilt sich in die vier Schnittstellen „widget“, „widgetWindow“, „WidgetModeChangeEvent“ und „ResolutionEvent“ auf. Die „widget“-Schnittstelle stellt die wichtigsten und meistgenutzten Funktionen zur Verfügung. Deshalb sind alle ihre Attribute in der Tabelle B.3 und alle Methoden in der Tabelle B.4 aufgelistet. Die drei anderen Schnittstellen finden in den Opera-Widgets kaum Verwendung und sollen deshalb nicht genauer betrachtet werden. Die „widgetWindow“-Schnittstelle bietet zwei Attribute zum Anzeigen einer Statusnachricht auf einer Managementseite und vier Methoden an. Mithilfe der vier Methoden kann das Widget selbst die eigene Größe und Position verändern. Die „WidgetModeChangeEvent“- und „ResolutionEvent“-Schnittstelle definieren jeweils ein Event. Das „WidgetModeChangeEvent“ wird bei Veränderung des Anzeigemodus der Widgets ausgelöst. Eine Veränderung der Größe des Darstellungsbereichs des Widgets führt zur Auslösung des „ResolutionEvents“. [23]

Name	**Datentyp**	**Wert**
identifier	readonly attribute DOMString	Einzigartiger Identifikator der Widget-Instanz
originURL	readonly attribute DOMString	URL-kodierter Wert des Identifikators des Widgets (URL, unter welcher das Widget heruntergeladen werden kann)
widgetMode	readonly attribute DOMString	Name des aktuellen Anzeigemodus

Name	Datentyp	Wert
onshow	attribute Function	Rückruffunktion, die beim Anzeigen des Widgets ausgelöst wird
onhide	attribute Function	Rückruffunktion, die beim Verstecken des Widgets ausgelöst wird

Tabelle B.3.: Auflistung aller Attribute der „widget“-Schnittstelle des Opera-Widget-Formats

Name	Datentyp Rückgabewert	Datentyp der Übergabeparameter	Kurzbeschreibung
openURL	void	(DOMString url)	Aufruf der gegebenen „url“ mit einem entsprechendem Schema-Handler
preferenceForKey	string	(DOMString key)	Holt den Wert für eine durch einen Schlüssel „key“ gegebene Präferenz
setPreferenceForKey	void	(DOMString value, DOMString key)	Setzt für die durch den Schlüssel gegebene Präferenz den Wert „value“
getAttention	void	()	Lenkt die Aufmerksamkeit des Nutzers auf das Widget (plattformabhängig)
showNotification	void	(DOMString msg, Function callback)	Zeigt ein Fenster mit der „msg“ an und ruft bei Bestätigung die „callback“-Funktion
show	void	()	Zeigt ein verstecktes Widget wieder an
hide	void	()	Versteckt ein Widget

Tabelle B.4.: Auflistung aller Methoden der „widget“-Schnittstelle des Opera-Widget-Formates

B.4. Nutzereinstellungen

Bei Opera-Widgets gibt es keine Möglichkeit Nutzereinstellungen mittels Konfigurationsdaten vorzudefinieren. Es existieren lediglich die zwei Methoden „preferenceForKey (key)“ und „setPreferenceForKey (value, key)“ des „widget“-Objektes zum Laden und Abspeichern einer Präferenz.

B.5. Lokalisierung

Das Widget-Format von Opera enthält keine Möglichkeit zur Lokalisierung der Widgets. Eine Lokalisierung kann nur mit herkömmlichen Mitteln basierend auf JavaScript, welche auch für Webseiten bzw. Webapplikationen genutzt werden, erfolgen.

Anhang C.

Aufbau des iGoogle-Gadget-Formates

Der Aufbau eines iGoogle-Gadgets besteht aus lediglich einer validen XML-Datei. Diese enthält alle fünf Grundbestandteile eines Widgets. Das Wurzelelement des XML-Dokuments ist das „Module“-Element. Dieses kann die folgenden drei Kinderelemente besitzen: „ModulePrefs“, „UserPref“ und „Content“. Im „ModulePrefs“-Element befinden sich sämtliche Metainformationen, einige Konfigurationsdaten und die Konfigurationen für die Lokalisierung des Widgets. Es ist optional und darf maximal einmal pro Gadget vorkommen. Das Elemente „UserPref“ ist ebenso optional, darf allerdings beliebig oft im Dokument auftreten. Es dient zur Konfiguration der Nutzereinstellungen. Das „Content“-Element enthält den Inhalt des Widgets und kann ein paar Konfigurationsdaten enthalten. Es muss in jeder Gadget-Datei mindestens einmal vorhanden sein und kann allerdings auch mehrfach auftreten. Jedes iGoogle-Gadget wird immer von einem Server vorverarbeitet. Der Anhang G enthält eine Beispieldatei. [14]

C.1. Metainformationen

Alle Metainformationen eines iGoogle-Gadgets sind optionale Angaben. Bis auf eine Ausnahme bilden alle Metadaten jeweils ein Attribut des „ModulePrefs“-Elementes. Alle Attribute sind mit ihrer Bedeutung in der Tabelle C.1 verzeichnet. Die einzige Metainformation, die ein eigenes Element als Kind von „ModulePrefs“ besitzt, ist für das Icon des Widgets gedacht. Das „Icon“-Element nimmt die Ausnahmestellung ein, da es zwei verschiedene Möglichkeiten für diese Metainformation gibt. Das Icon kann der Autor entweder mittels einer Base64-Kodierung direkt in das XML-Dokument einbinden oder als externe Datei mit einer Adresse im Internet veröffentlichen und dann einen Verweis auf diese angeben. Die Auflösung der Bilddatei sollte am besten 16*16 Pixel betragen. Das Icon ist vor allem für die Verwendung in der Widget-Umgebung zur Anzeige in der Titelleiste des Gadgets gedacht. Den Modus wählt der Autor durch das „mode“-Attribut aus. Ist dieses nicht vorhanden, interpretiert der Server den Inhalt des Textknotens unterhalb des „Icon“-Elementes als Verweis auf eine externe Bildressource. Bei Verwendung des Attributs muss dessen Wert stets

„base64“ sein. Dadurch weiß der Server, dass der Textknoten ein base64-kodiertes Bild enthält. Bei Verwendung des Base64-Modus muss der Autor zudem das Attribut „type“ des „Icon“-Elementes benutzen. Der Wert dieses Attributs muss dem Namen des MIME-Typs des eingebetteten Bildes entsprechen. [14, 15]

Name	**Bedeutung**
title	Titel des Gadgets
title_url	Angabe einer externen Adresse, mit welcher der Server dem Titel einen Link hinzufügen kann
directory_title	Titel für das Widget, wenn dieses in einem Widget Store liegt
description	Beschreibung der Funktionalität
author	Name des Autors bzw. Auflistung der Namen der Autoren
author_email	E-Mail-Adresse zur Kontaktaufnahme zum Autor
author_location	Geographische Position des Autors in Textform
author_affiliation	Organisation, welcher der Autor angehört
author_photo	Absolute URL zu einem Foto des Autors im PNG-, JPG- oder GIF-Format (PNG präferiert)
author_aboutme	Kurzbeschreibung des Authors über sich selbst (möglichst in ca. 500 Zeichen)
author_link	Link zur Webseite des Authors
author_quote	Zitat oder Spruch, den ein Autor gerne angeben möchte (möglichst in ca. 300 Zeichen)
screenshot	Absolute URL zu einem Screenshot des Gadgets im PNG-, JPG- oder GIF-Format (PNG präferiert)
thumbnail	Absolute URL zu einem Bild im PNG-, JPG- oder GIF-Format (PNG präferiert), welches die Hauptfunktionalität des Gadgets demonstriert

Tabelle C.1.: Auflistung aller Attribute des „ModulePrefs“-Elementes für die Metainformationen des iGoogle-Gadgets

C.2. Konfigurationsdaten

Die Konfigurationsdaten verteilen sich auf das „ModulePrefs“- und die „Content“-Elemente. Sämtliche Konfigurationsinformationen sind grundsätzlich optionale Angaben. Die Tabelle C.2 zeigt alle Elemente und Attribute des „ModulePrefs“-Elements.

Name	**Typ**	**Eltern-element**	**Bedeutung**
width	Attribut	ModulePrefs	Präferierte Anzeigebreite

Name	**Typ**	**Eltern-element**	**Bedeutung**
height	Attribut	ModulePrefs	Präferierte Anzeigehöhe
Require	Element	ModulePrefs	Anfordern eines unbedingt benötigten Features
Optional	Element	ModulePrefs	Anfordern eines optionalen Features
feature	Attribut	Require oder Optional	Name des anzufordernden Features
Param	Element	Require oder Optional	Spezifiziert einen Parameter für ein Feature
name	Attribut	Param	Name des Parameters
Preload	Element	ModulePrefs	Vorladen einer Ressource durch den Server anfordern
href	Attribut	Preload	URL der vorzuladenden Ressource
view	Attribut	Preload	Auflistung der Anzeigemodi, für welche das Vorladen getätigt werden soll
authz	Attribut	Preload	Typ der Authentifizierung
sign_owner	Attribut	Preload	Gibt an, ob die Authentifizierung mit den Daten des Besitzer getätigt wird
sign_viewer	Attribut	Preload	Gibt an, ob die Authentifizierung mit den Daten des Betrachters getätigt wird
oauth_service_name	Attribut	Preload	Name, welcher einen OAuth-Service identifiziert, der in einem OAuth-Service-Element spezifiziert ist
oauth_token_name	Attribut	Preload	Name, der ein spezielles OAuth-Token für eine bestimmte Ressource identifiziert
oauth_request_token	Attribut	Preload	Request-Token, falls ein Service Provider die automatische Bereitstellung eines im Voraus genehmigten Tokens unterstützt
oauth_request_token_secret	Attribut	Preload	Das zugehörige Geheimnis zum unter oauth_request_token angegebenen Token
OAuth	Element	ModulePrefs	Container-Element für OAuth-Konfigurationen
Service	Element	OAuth	Spezifikation eines OAuth-Services
name	Attribut	Service	Name zum Referenzieren eines spezifizierten Services zur Laufzeit

Name	**Typ**	**Eltern-element**	**Bedeutung**
Request	Element	Service	Spezifikation der URL des OAuth-Request-Token-Servers
Access	Element	Service	Spezifikation der URL des OAuth-Access-Token-Servers
url	Attribut	Request oder Access	URL des Endpunktes des Servers
method	Attribut	Request oder Access	Zugriffsmethode
param_location	Attribut	Request oder Access	Position, in der die OAuth-Parameter übergeben werden
Authorization	Element	Service	Spezifikation der URL zu der Seite, auf welcher der Nutzer zur Authentifizierung seine Anmeldedaten eingeben kann
url	Attribut	Authorization	URL der Seite, welche in einem neuen Fenster geöffnet wird

Tabelle C.2.: Auflistung aller Elemente und Attribute des „ModulePrefs“-Elementes zum Konfigurieren des iGoogle-Gadgets

Durch die Attribute „width“ und „height“ kann ein Autor die präferierte initiale Anzeigegröße unabhängig von einem speziellen Inhalt des Gadgets definieren. Als Werte sind nur positive ganze Zahlen gültig. Mit dem „Require“- und dem „Optional“-Element kann der Autor jeweils ein Feature mit Zusatzfunktionalität für das Widget anfordern. Bei Verwendung muss stets das „feature“-Attribut vorhanden sein und den Namen eines Features als Wert besitzen. Der Entwickler kann beliebig viele „Require“- und „Optional“-Elemente für ein Gadget definieren. Für Features die mit Parametern konfigurierbar sind, existiert das „Param“-Element. Es muss bei Verwendung stets ein „name“-Attribut mit dem Namen besitzen. Den Wert des Parameters hinterlegt der Autor als Kind von „Param“ als Textknoten. Jedes Feature kann durch mehrere Parameter spezifiziert sein. [14]

Das Verwenden eines „Preload“-Elementes erfordert stets ein „href“-Attribut. Durch das Element erhält der Server, welcher auch der Proxy-Server für das Nachladen von Daten ist, die Aufgabe die Ressource an der angegebenen Adresse schon während des Renderprozesses herunterzuladen und zu cachen. Ein „Preload“ tritt stets in Verbindung mit einem Aufruf der JavaScript-Funktion „makeRequest“ auf. Diese ist durch die API von iGoogle gegeben und dient dem dynamischen Nachladen als Ajax-Ersatz mit Verwendung eines Proxys. Bei Aufruf der Funktion stehen die Daten für das

Gadget somit schneller zur Verfügung. Für das „Preload“-Element stehen noch einige weitere Parameter zur Verfügung. Durch das optionale „view“-Attribut kann ein Entwickler angeben, welche Anzeigemodi die vorgeladene Ressource verwenden. Die Angabe der Modi erfolgt als komma-separierte Liste der einzelnen Werte. Die restlichen Parameter sind alle für das Laden von Ressourcen nötig, bei welchen ein Zugriff nur mit vorheriger Authentifizierung gewährt wird. Mit dem „authz“-Attribut gibt der Autor das benötigte Authentifizierungsverfahren an. Valide Werte sind „none“, „signed“ und „oauth“. Der Wert „none“ entspricht dem Standardverhalten bei Abwesenheit des Attributs. Ist der „oauth“-Modus ausgewählt, wendet der Server OAuth zur Authentifizierung an. Damit OAuth verwendet werden kann, muss es zuvor mittels einem „OAuth“-Element und dessen Kindern und Attributen konfiguriert sein. Die Attribute „sign_owner“ und „sign_viewer“ sind jeweils vom booleschen Typ. Der Standardwert für beide ist „true“, d.h. der Server hängt an die Ressourcen-Anfrage sowohl die Authentifizierungsdaten des Besitzers als auch die des Betrachters. Mit den anderen vier Attributen kann der Autor Konfigurationsdaten für die OAuth-Authentifizierung angeben. Ist das „oauth_service_name“-Attribut nicht vorhanden, nutzt der Server den ersten spezifizierten OAuth-Service ohne Namen. [14]

Mit dem „OAuth“-Element ist es dem Entwickler möglich OAuth-Dienste für die Authentifizierung für das dynamische Nachladen von Daten zu konfigurieren. Das Element darf maximal einmal in einer Gadget-Datei vorhanden sein. Es besitzt eine beliebige Anzahl an „Service“-Elementen, mindestens jedoch eines. Mittels diesen spezifiziert der Autor jeweils einen OAuth-Service für das Widget. Jeder Service kann durch das „name“-Attribut einen Namen zum späteren Referenzieren erhalten. Falls der Nutzer zur Authentifizierung bei einem OAuth-Dienst Anmeldedaten eingeben muss, sollte der Entwickler das „Authorization“-Element verwenden. Mit diesem referenziert er eine Webseite, die der Benutzer als Pop-Up zur Eingabe seiner Daten gezeigt bekommt. Das „Authorization“-Element darf nie ohne sein Attribut „url“ auftreten, da es die URL zur anzuzeigenden Seite enthält. Die Elemente „Request“ und „Access“ definieren jeweils die URL für den Request-Token-Server bzw. Access-Token-Server. Beide benötigen stets ein „url“-Attribut. Die anderen beiden Attribute sind jeweils optional. Mit dem „method“-Attribut wird die Request-Methode festgelegt. Die möglichen Werte sind „GET“ und „POST“, wobei „GET“ den Standardwert darstellt. Die Auswahl der Übertragungsposition der Authentifizierungsparameter geschieht über das „param_location“-Attribut. Mögliche Positionen sind der „header“ oder der „body“ der HTTP (Hypertext Transfer Protocol)-Anfrage oder die Übertragung als Parameter in der „url“. Der Standardwert bei Abwesenheit des Attributs ist „header“. Mehr Informationen über das Erstellen von Gadgets, die OAuth benutzen, können über die Seite „Writing OAuth Gadgets“[1] der Google Developers Webseite abgerufen werden. [14, 17]

[1] https://developers.google.com/gadgets/docs/oauth

Das „Content“-Element kann die Attribute „type“, „href“, „view“, „preferred_height“ und „preferred_width“ besitzen. Mit dem „type“-Attribut wird der Typ des Inhalts des Widgets konfiguriert. Die validen Werte sind „html“ und „url“, wobei „html“ auch den Standardwert repräsentiert. Details über die unterschiedlichen Inhaltstypen sind im Kapitel C.3 erläutert. Die Angabe eines „href“-Attributs mit einem validen URI ist bei Verwendung des „url“-Typs unbedingt notwendig. Nähere Informationen befinden sich im Abschnitt zum Inhalt eines iGoogle-Gadgets. Mittels dem „view“-Attribut kann der Autor den Anzeigemodus oder eine Liste von Anzeigemodi, für welche der folgende Inhalt des Gadgets gedacht ist, angeben. Eine Liste kann er durch Kommaseparierung mehrerer Werte erstellen. Valide Werte sind „canvas“, „home“, „preview“, und „profile“, welche jeweils einem Anzeigemodus mit dem entsprechenden Namen repräsentieren. Beim „home“-Modus stellt der UA das Gadget in einem kleinen Bereich dar. Meist befindet es sich in dieser Ansicht mit mehreren Widgets auf einer Seite. Die Darstellung von „profile“ entspricht der von „home“. Der Anzeigemodus „canvas“ existiert für eine großflächige Präsentation des Gadgets. Es ist typischerweise der Hauptinhalt der Seite. Sowohl beim „home“- als auch beim „canvas“-Modus ist der Nutzer des Widgets bekannt. Beim „preview“-Anzeigemodus ist der Benutzer hingegen typischerweise nicht bekannt, da es sich lediglich um eine Demonstrationsansicht des Gadgets handelt. Wenn ein Widget eine Darstellung für den „canvas“-Modus besitzt, muss es auch eine für den „home“-Modus enthalten. Ein weiterer valider Wert für das „view“-Attribut ist „default“. Dieser entspricht keinem Anzeigemodus, sondern tritt als Wildcard für alle Modi ein, für die sonst kein Inhalt definiert ist. Das gleiche Verhalten tritt bei Abwesenheit des „view“-Attributs ein. Der Wert „default“ kann natürlich auch in Form einer Liste auftreten. Die beiden Attribute „preferred_height“ und „preferred_width“ kann ein Autor zur Angabe der gewünschten initialen Darstellungsgröße für den darauf folgenden Widget-Inhalt verwenden. Der Wert muss jeweils eine positive ganze Zahl repräsentieren. [13, 14]

C.3. Inhalt

Der Inhalt für das Gadget wird durch „Module“-Elemente in die XML-Datei integriert. Bei iGoogle-Gadgets können mehrere unterschiedliche Inhalte für ein Widget existieren, wobei jedes „Module“-Element exakt einen definiert. Mit dem „view“-Attribut spezifiziert der Autor für welche Anzeigemodi der entsprechende Inhalt zuständig ist. Ein Autor sollte jeden Modus für maximal ein „Module“-Element verwenden. Für das Definieren eines Inhaltes gibt es die zwei Möglichkeiten HTML- und URL-Modus. Zur Angabe des genutzten Verfahrens existiert das „type“-Attribut. Wenn dieses nicht vorhanden ist oder der Wert „html“ entspricht, handelt es sich um den HTML-Modus. Bei diesem befindet sich im Normalfall sämtlicher HTML-Code in der XML-Datei. Der Inhalt befindet sich in Form eines Textknotens als

Kind am „Content“-Element. Der gesamte HTML-Code des Gadgets muss in einem CDATA-Abschnitt („<![CDATA[HTML-Inhalt]]>“) geschachtelt sein, damit ihn der XML-Prozessor nicht verarbeitet. Im CDATA-Abschnitt kann der Autor normales HTML für den Aufbau des Gadgets verwenden. Innerhalb des HTML-Codes kann er „script“- und „style“-Tags verwenden, um dem Widget die gewünschte Funktionalität und Gestaltung zu geben. Diese dürfen natürlich auch mittels Referenzen auf externe Dateien mit dem entsprechenden Code verweisen. Allerdings darf der Entwickler die Tags „html“, „head“ und „body“ nicht benutzen. Der Inhalt beginnt sofort mit dem Teil eines HTML-Codes, der sich normalerweise innerhalb des „body“-Elementes befindet. Das Integrieren eines HTML-Kopfteils ist nicht möglich. Das „html“-, „head“- und „body“-Tag erstellt der Server für jedes Widget automatisch. [15]

Der HTML-Modus ist der empfohlene Modus, da der Entwickler mit ihm die vollen Möglichkeiten eines iGoogle-Gadgets ausnutzen kann. Beim URL-Modus handelt es sich eher um eine Art Kompatibilitätsmodus. Der Autor gibt mit dem „src“-Attribut eine Referenz auf eine Webseite bzw. Webanwendung an. Den Wert von „src“ nutzt der Server für das neue „iframe“-Tag, welches er zum Darstellen des Inhalts des Gadgets verwendet. Die vorhandene Webseite wird somit unverändert mittels einem „iframe“ in die Widget-Umgebung eingebettet. Wenn sich ein HTML-Dokument an der URL-Adresse befindet, muss dieses wie eine normale HTML-Datei die Tags „html“, „head“ und „body“ besitzen, da eine unveränderte Darstellung stattfindet. Der Vorteil dieses Verfahrens ist, dass ein Entwickler seine vorhandene Webanwendung sehr schnell in ein Gadget umwandeln kann. Der Nachteil ist vor allem, dass die API nicht vollständig zur Verfügung steht. Beim URL-Modus darf sich kein Inhalt im „Content“-Element selbst befinden. Der Server ignoriert jeglichen Inhalt, wie HTML- oder JavaScript-Code, im gesamten iGoogle-Gadget-XML-Dokument. [15]

Die JavaScript API ist im Gegensatz zu den zwei bisherig betrachteten Formaten sehr umfangreich. Die Untersuchung sämtlicher Funktionen würde den Umfang der Arbeit übersteigen. Viele der angebotenen Funktionen befinden sich in jeweils einer der zahlreich angebotenen Features. Eine Beschreibung des Großteils der Funktionen befindet sich in der Gadget-API-Reference[2] auf der Google Entwickler-Webseite. Die wichtigsten API-Bibliotheken sind in der Tabelle C.3 aufgelistet. Die Spalte „Typ“ gibt jeweils an, ob die Funktionen zur Kern-API gehören oder nur durch das Anfordern mit einem „Require“- bzw. „Optional“-Element zur Verfügung stehen. Die angegebenen Namen der Bibliotheken entsprechen den jeweiligen Namen der Features. Eine der wichtigsten Funktionen ist „registerOnLoadHandler(callback)“ aus der „util“-Bibliothek. Sie dient als Ersatz für das „onload“-Attribut des „body“-Tags eines HTML-Dokumentes, da der Autor im „Content“-Element kein „body“-Tag verwenden darf. Der Parameter „callback“ steht für die Funktion, welche nach dem Laden des Widgets zur Ausführung kommt. [13]

[2] https://developers.google.com/gadgets/docs/reference/

Wert	Typ	Bedeutung
dynamic-height	Feature	Gadget kann seine Höhe dynamisch ändern
flash	Feature	Einbetten von Flash-Inhalten in das Gadget
io	Kern	Funktionen für das dynamische Nachladen und Senden von Daten; beachtet die Konfigurationen zur Authentifizierung
json	Kern	Funktionen zum Konvertieren von JSON in einen String und umgekehrt
locked-domain	Feature	Isoliert das Gadget von anderen auf der gleichen Seite zum Schutz sensitiver Daten
minimessage	Feature	Erstellen von Nachrichten für den Benutzer und Anzeigen dieser
prefs	Kern	Lesender Zugriff auf Nutzereinstellungen und einige Parameter der Widget-Instanz
pubsub	Feature	Funktionen zur Kommunikation über Publisher-/Subscriber-Nachrichtenkanäle
rpc	Feature	Funktionen, um entfernte Prozeduraufrufe zu tätigen für Widget-zu-Umgebung-, Umgebung-zu-Widget- und Inter-Widget-Kommunikation
setprefs	Feature	Schreibender Zugriff auf Nutzereinstellungen
settitle	Feature	Schreibender Zugriff auf den Titel des Gadgets
skins	Feature	Informationen über den vom Betrachter genutzten Skin
tab	Feature	Erstellen von Gadgets, die über mehrere Tabs verfügen
util	Kern	Allgemeine Funktionen: Kodieren/Dekodieren von Strings, Hinzufügen von „onload“-Funktionen, Prüfen auf Feature-Unterstützung, Holen der Parameter eines Features
views	Feature	Operationen mit den Ansichtsmodi
window	Feature	Operationen mit Element in dem das Gadget geschachtelt ist

Tabelle C.3.: Auflistung der Bibliotheken der JavaScript-API von iGoogle-Gadgets

C.4. Nutzereinstellungen

Für das Definieren von Nutzereinstellungen existiert bei den iGoogle-Gadgets das „UserPref“-Element. Es darf im XML-Dokument beliebig oft auftreten. Jedes Vor-

kommen spezifiziert exakt eine Präferenz. Die Nutzereinstellungen haben bei iGoogle-Gadgets, außer dem persistenten Speichern von Daten zu einem Widget für einen speziellen Benutzer, noch eine weitere Funktion. Aus allen spezifizierten „UserPref“-Elementen erstellt der Server für jedes Gadget ein „div“-Element, welches dem Nutzer die Möglichkeit gibt Werte für die Einstellungen zu setzen. Normalerweise ist der „div“-Bereich inklusive dessen Inhalt nicht sichtbar. Wenn der Benutzer allerdings in der Titelleiste auswählt, dass er die Einstellungen bearbeiten möchte, bekommt er das Element und den zugehörigen Inhalt angezeigt. Er sieht die Werte aller Präferenzen und kann diese abändern. Anschließend kann er seine Änderungen speichern oder verwerfen. Der Entwickler muss sich um eine Möglichkeit zum Modifizieren der Nutzereinstellungen durch den Nutzer nicht mehr kümmern.

Das „UserPref“-Element kann die fünf Attribute „name“, „display_name“, „urlparam“, „datatype“, „required“ und „default_value“ besitzen. Das „name“-Attribut ist die einzige stets benötigte Angabe. Es definiert den Identifikator für eine Nutzereinstellung und muss somit einen einzigartigen Wert erhalten. Dieser darf nur Buchstaben, Nummern und Unterstriche enthalten. Alle anderen Angaben sind optional. Mit dem „display_name“-Attribut kann der Autor das anzuzeigende Label der Präferenz im Einstellungsmenü spezifizieren. Sollte er keinen Wert für das Label definieren, erfolgt die Anzeige des Identifikators der Einstellung. Die Verwendung des Attributs „urlparam“ ergibt nur bei Nutzung des URL-Modus einen Sinn. Es ermöglicht das Erweitern der URL, die das „iframe“-Tag benutzt. Die Nutzereinstellungen finden in diesem Fall als Parameter der URL Verwendung. Der Wert von „urlparam“ ist dabei der Name des Parameters. Das „default_value“-Attribut bestimmt den Startwert einer Präferenz. Ob eine Nutzereinstellung optional ist oder einen Wert zur Grundlegenden Funktionalität des Gadgets besitzen muss, kann ein Autor durch das Attribut „required“ anzeigen. Der Wert dafür muss booleschen Typs sein. Bei Abwesenheit des Attributs geht die Widget-Umgebung von „false“ aus, was heißt, dass die Präferenz optional ist. Ist eine Nutzereinstellung nicht optional und hat keinen Startwert definiert, ist das Einstellungsmenü schon von Beginn an geöffnet. Es lässt sich erst schließen, wenn der Nutzer allen benötigten Einstellungen einen Wert zugewiesen hat. [11]

Mit dem „datatype“-Attribut definiert der Entwickler den Typ der Präferenz. Es gibt fünf unterschiedliche Typen: „string“, „bool“, „enum“, „hidden“ und „list“. Der Wert „string“ ist der Standardwert bei Abwesenheit des Attributs. Die Nutzereinstellung erhält den Datentyp „string“ und das Einstellmenü erhält eine Erweiterung durch ein „input“-Tag vom Typ „text“. Der Wert „bool“ führt zum Hinzufügen eines „input“-Tags vom Typ „checkbox“. Im JavaSkript-Code darf die entsprechende Präferenz nur „true“ oder „false“ zugewiesen bekommen. Der Typ „hidden“ führt zu keiner Darstellung im Einstellungsmenü. Die Präferenz ist vor dem Nutzer versteckt und erfährt nur durch den JavaScript-Code eine Änderung ihres Wertes. Ist der Typ

einer Nutzereinstellung auf „list“ eingestellt, bildet diese ein dynamisches Array. Im Einstellfenster präsentiert sich die Liste durch eine „input“-Box vom Typ „text“ mit einem zugehörigen Button und einer Auflistung aller Werte, welche sich in der Liste aktuell befinden. Hinter jedem dieser Elemente des Arrays befindet sich ein kleines „x“, welches beim Anklicken dieses aus der Liste entfernt. Durch das Texteingabefeld kann der Benutzer den Wert eines neuen Elementes für das Array festlegen und mit einem Klick auf den zugehörigen Button den Wert anhängen. Eine Präferenz vom Typ „enum“ erzeugt ein neues „select“-Tag im Menü. Die Werte für die Auswahlliste muss ein Autor durch „EnumValue“-Elemente definieren. Jeder „EnumValue“ muss ein Attribut „value“ besitzen, welches den Wert bei Auswahl dieser Option festlegt. Mit dem optionalen „display_value“-Attribut kann der Entwickler zusätzlich den angezeigten vom tatsächlichen Wert der Option abheben. Jedes „EnumValue“-Element bildet im Einstellungsmenü ein neues „option“-Tag für das „select“-Element. [13, 11]

Für das Lesen von Nutzereinstellungen im JavaScript-Code reicht die Kern-API aus. Für das Schreiben ist zuerst das Anfordern des „setprefs“-Features nötig. Um Präferenzen im JavaScript-Code zu benutzen, muss der Autor zunächst ein neues Präferenz-Objekt mittels „new gadgets.Prefs()“ definieren. Dieses hat für die Nutzereinstellungen die Methoden „getArray(key)“, „getBool(key)“, „getFloat(key)“, „getInt(key)“ und „getString(key)“. Bei Nutzung des „setprefs“-Features existieren zusätzlich die Methoden „set(key, value)“ und „setArray(key, value)“. Der Parameter „key“ ist jeweils der Name der Präferenz und „value“ stellt den neu zu setzenden Wert dar. Die Methoden zum Laden geben den Wert jeweils im Datentyp entsprechend zu ihrem Namen zurück. Die „set“-Methode ist dem gegenüber allgemeingültig. Nur für Arrays sollte „setArray“ anstatt „set“ verwendet werden. Eine Nutzereinstellung vom Typ „list“ kann der Entwickler sowohl als Array laden und speichern, als auch mittels einem String, bei welchem alle Werte der Liste jeweils durch ein „|“-Zeichen voneinander separiert sind. [13]

C.5. Lokalisierung

Um ein iGoogle-Gadget zu lokalisieren, muss ein Autor „Locale“-Elemente als Kind vom „ModulePrefs“-Element verwenden. Jedes „Locale“-Element repräsentiert genau eine Lokalisierung für eine Sprache bzw. eine bestimmte Kombination aus Sprache und Region. Es bietet die vier Attribute „lang“, „country“, „language_direction“ und „messages“ an. Die Textrichtung der Lokalisierung ist durch den Wert von „language_direction“ bestimmt. Das Attribut ist auf die Eigenschaften „ltr“ und „rtl“ beschränkt. Bei Abwesenheit nimmt der Server den Standardwert „ltr“ an. Die Attribute „lang“ und „country“ zeigen Sprache und Region an, für welche die Lokalisierung gedacht ist. Beide haben den Standardwert „ALL“. Dieser deutet an, dass die entspre-

chende Lokalisierung für alle Sprachen bzw. Regionen gedacht ist. Des Weiteren ist er für das Zurückfallverhalten nötig. Valide Werte für die Sprache sind zwei Zeichen lange Werte aus der ISO639-1 und für die Region ISO3166-1-alpha-2-Codes. Beide sind im IANA language subtag registry verzeichnet. [12]

Die Übersetzungen bestehen bei iGoogle-Gadgets immer aus Schlüssel-Wert-Paaren. Es gibt zwei Möglichkeiten diese Paare in das Widget zu integrieren. Die erste Möglichkeit nutzt das „messages“-Attribut. Diesem weist der Autor eine URL als Referenz auf eine externe XML-Datei zu. Das Wurzelelement der Datei muss den Namen „messagebundle“ tragen. Als Kinder der Wurzel sind beliebig viele „msg“-Elemente zugelassen. Jede „msg“ repräsentiert eine Übersetzung in Form eines Schlüssel-Wert-Paares. Den Schlüssel stellt das „name“-Attribut, welches nicht optional ist. Die Übersetzung zum entsprechenden Schlüssel enthält einen Textknoten als Kind des „msg“-Elementes. Bei der zweiten Möglichkeit fügt der Entwickler die Schlüssel-Wert-Paare direkt in die Gadget-XML-Datei ein. Dazu benutzt er wieder „msg“-Elemente, welche er als Kinder des entsprechenden „Locale“-Elementes anfügt. [12]

Um die definierten Übersetzungen für das Gadget zu verwenden, stellt iGoogle zwei Verfahren bereit. Das Erste basiert auf Substitutionsvariablen. Der Autor verwendet dazu in der XML-Datei die Zeichenkette „__MSG_key__“, wobei er die Zeichen „key“ durch den Schlüssel der gewünschten Übersetzung austauscht. Falls der Schlüssel Leerzeichen enthält, muss er diese in der Variable durch Unterstriche ersetzen. Das Substitutionsverfahren kann überall in der XML-Datei des Gadgets zum Einsatz kommen. Somit kann der Entwickler auch sämtliche Konfigurationsdaten und Metainformationen lokalisieren. Die Substitutionsvariablen tauscht der Server vor Darstellung des Widgets für den jeweiligen Betrachter dynamisch aus. Die zweite Möglichkeit beschränkt sich auf den JavaScript-Teil des Gadgets. Zunächst muss wie bei den Nutzereinstellungen ein Präferenzobjekt mit „new gadgets.Prefs()“ definiert sein. Zum Laden einer Übersetzung kann der Autor anschließend die „getMsg(key)“-Methode verwenden. Der Parameter „key“ stellt den Schlüssel des Paares dar. Die Rückgabe des Aufrufs ist der Wert der Übersetzung. Bei beiden Verfahren wird ein Zurückfallverhalten eingesetzt. Findet ein Server zur aktuellen Kombination von Sprache und Region des Nutzers für einen Schlüssel keine Übersetzung, sucht er in der nächstallgemeineren Lokalisierung. [12]

Für die Unterstützung bidirektionaler Gadgets bietet iGoogle zusätzlich vier weitere Substitutionsvariablen an. Diese entsprechen alle dem Muster „__BIDI_key__“, wobei die Zeichenkette „key“ jeweils abhängig von der gewünschten Funktionalität durch einen der in Tabelle C.4 aufgelisteten Werte zu ersetzen ist.

Name („key“)	**Rückgabewert bei „ltr“**	**Rückgabewert bei „rtl“**
START_EDGE	left	right
END_EDGE	right	left
DIR	ltr	rtl
REVERSE_DIR	rtl	ltr

Tabelle C.4.: Auflistung aller BIDI-Substitutionsvariablen von iGoogle-Gadgets

Anhang D.

Aufbau des UWA-Widget-Formates

Der grundlegende Aufbau eines UWA-Widgets befindet sich wie bei den iGoogle-Gadgets in einer Datei, wobei es sich bei UWA um eine XHTML-Datei handelt. Diese muss UTF-8-kodiert sein und den Netvibes-Namensraum inkludieren. Die Dateiendung sollte entweder „.html“ oder „.xml“ sein. Der Aufbau entspricht grundsätzlich dem einer normalen XHTML-Datei. Nach dem XML-Header und dem Verweis auf die XHTML-Dokumenttypdefinition durch ein „DOCTYPE“-Tag folgt das „html“-Element. Dieses sollte den XHTML-Namensraum „http://www.w3.org/1999/xhtml“ und den Netvibes-Namensraum „http://www.netvibes.com/ns/“, an welchen der Präfix „widget“ gebunden wird, als „xmlns“-Attribute besitzen. Im „head“-Element sind alle Metainformationen und Konfigurationsdaten enthalten. Um das Widget während der Entwicklung prüfen zu können, ohne jedes Mal einen Upload in das Netvibes Ecosystem tätigen zu müssen, gibt es den Standalone-Modus. Für dessen Nutzung integriert der Autor zusätzlich ein „script“- und ein „link“-Element in den Kopfteil der XHTML-Datei:
<link rel=„stylesheet“ type=„text/css“ href=„http://www.netvibes.com/themes/uwa/style.css“ /><script type=„text/javascript“ src=„http://www.netvibes.com/js/UWA/load.js.php?env=Standalone“></script>

Die beiden Dateien emulieren die normale Widget-Umgebung und stellen die gesamte API dieser zur Verfügung. Allerdings können kleine Unterschiede zwischen Standalone-Modus und normaler Darstellung auf der Netvibes-Seite auftreten, weshalb Netvibes einen finalen Test in der UWA-Widget-Umgebung empfiehlt. Ein Beispiel eines UWA-Widgets befindet sich im Anhang H. [28]

D.1. Metainformationen

Alle Metainformationen, mit Ausnahme von drei Angaben eines UWA-Widgets, sind jeweils mittels eines „meta“-Elements in die UWA-Datei integriert. Die Ausnahmen bilden der Titel und die Icons des Widgets. Es gibt zwei verschiedene Icons: Favicon und Rich-Icon. Das Favicon ist das normale Icon eines UWA-Widgets. Das Rich-Icon

ist vor allem für den Export in andere Formate nötig. Titel und Favicon müssen für das Bilden eines validen UWA-Widgets stets vorhanden sein. Für die Namensgebung muss der Autor das „title“-Element benutzen. Er geht dazu genauso vor, als wollte er den Titel einer XHTML-Datei setzen. Ein „link“-Element als Kind vom „head“-Tag des Dokuments referenziert das Favicon des Widgets. Das Icon sollte ein Favicon im klassischen Sinne darstellen, d.h. eine Datei im ICO-, PNG- oder GIF-Format (PNG präferiert) mit 16*16 Pixeln und 256 Farben. Das „rel“-Attribut dieses Elements muss dabei den Wert „icon“ besitzen. Des Weiteren sollte durch ein „type“-Attribut der Datentyp des Icons angegeben sein. Der Wert muss dem MIME-Typ des Bildes entsprechen. Das dritte benötigte Attribut ist „href“, welches die Referenz auf die Datei durch eine URL bereitstellt. Um ein Widget mit einem Rich-Icon zu erweitern, muss der Entwickler ein weiteres „link“-Element hinzufügen. Dieses besitzt die drei gleichen Attribute. Der Wert von „rel“ muss allerdings „rich-icon“ sein. Die Bilddatei sollte natürlich auch entsprechend größer als das Favicon sein. [28]

Alle anderen Metainformationen sind optional und bestehen aus einem „meta“-Element als Kind des „head“-Tags. Sie sind in Tabelle D.1 aufgeführt. Der Wert der Name-Spalte entspricht dem jeweiligen „name“-Attribut des entsprechenden „meta“-Elements. Das „content“-Attribut erhält jeweils den gewünschten Wert für das Metadatum. Ein Autor sollte stets so viele Metainformationen wie möglich angeben, um den Export für möglichst alle Formate anbieten zu können, da verschiedene Plattformen unterschiedliche Anforderungen an die Metaangaben haben.

Name	**Bedeutung**
author	Name des Autors
email	E-Mail-Adresse des Autors
website	URI, die den Autor assoziiert
description	Beschreibung der Funktionalität
version	Version des Widgets
screenshot	Absolute URL zum Screenshot des Gadgets (280 Pixel Breite empfohlen)
thumbnail	Absolute URL zum Thumbnail des Gadgets (120*60 Pixel empfohlen)
keywords	Schlüsselwörter für das Widget

Tabelle D.1.: Auflistung aller durch „meta“-Elemente gegebenen Metainformationen eines UWA-Widgets

D.2. Konfigurationsdaten

Die Konfiguration eines UWA-Widgets geschieht über weitere „meta“-Elemente. Das „name“-Attribut entspricht wieder dem Identifikator der einzelnen Parameter. Insgesamt stehen bei UWA-Widgets nur drei Möglichkeiten für die Konfiguration zur Verfügung. Mit dem äpiVersionParameter kann der Autor die Version der API angeben, auf welcher das Widget basiert. Der Parameter „autoRefresh“ darf nur eine positive ganze Zahl zugewiesen bekommen. Der Wert gibt an, aller wie vieler Minuten die Widget-Umgebung den Inhalt des Widgets aktualisieren muss. Bei Abwesenheit des entsprechenden „meta“-Elementes findet keine regelmäßige Aktualisierung statt. Der letzte Parameter gibt an, ob aufgetretene JavaScript-Fehler in der Konsole geloggt werden sollen. Er trägt den Name „debugMode“ und nimmt nur die Werte „true“ oder „false“ entgegen. Mit „true“ erfolgt das Aufzeichnen der Fehler. [28]

D.3. Inhalt

Wie bei allen bisher betrachteten Widget-Formaten teilt sich der Inhalt in die drei Bestandteile HTML, CSS und JavaScript auf. Sämtlicher Inhalt sollte sich immer nur in der Widget-Datei befinden. Ein Entwickler sollte für CSS und JavaScript keine „link“- bzw. „script“-Elemente mit Verweisen auf externe Dateien verwenden. Eine Ausnahme bilden die beiden Elemente für das Integrieren des Standalone-Modus, welche eine Referenz auf externe Dokumente besitzen. Die beste Vorgehensweise ist, sämtlichen CSS- und JavaScript-Code in einem „link“- und einem „script“-Tag im Kopfteil der XHTML-Datei abzulegen. [28]

Das UWA-Widget-Format beinhaltet wie auch das iGoogle-Format eine umfangreiche API für den JavaScript-Code. Allerdings erweitert die UWA-API die Funktionalitäten von JavaScript nicht nur, sondern führt zugleich eine Limitierung ein. Der Autor sollte bestimmte JavaScript-Funktionen in UWA-Widgets nicht verwenden. Dies betrifft die „document“- und „window“-Objekte und deren Methoden. Die Einschränkung des JavaScript-Codes findet vor allem statt, damit sich die Widgets untereinander nicht beeinträchtigen können. Die API bietet allerdings für die nicht zu verwendenden Funktionalitäten diverse Ersatz-Funktionen an. Diese stellt das „widget“-Objekt zur Verfügung. Das „widget“-Objekt besitzt allerdings noch einige weitere Methoden und Eigenschaften. Einige wichtige sind in der Tabelle D.2 aufgeführt. Die Methoden sind von den Eigenschaften unterscheidbar, da sie stets mit Klammern und ihren eventuellen Parametern angegeben sind. [27, 30]

Das „widget“-Objekt definiert weiterhin einige Event-Handler. Diesen sollte der Entwickler jeweils maximal einmal eine Rückruf-Funktion oder einen Codeblock zuwei-

Name	Bedeutung
setAutoRefresh(delay)	Setzt einen neuen Wert „delay“ für die Zeitabstände der Aktualisierung des Widget-Inhaltes in Minuten (Konfigurationswert wird überschrieben)
openURL(url)	Öffnet ein neues Browserfenster mit der gegeben URL
body	Referenz auf das „body“-Element des Widgets
createElement(tagName)	Erstellt ein neues DOM-Element mit dem gegebenen „tagName“
createElement(tagName, options)	Wie obere Funktion nur mit einem JSON-Objekt (options), welches die Attribute des neuen Elements enthält (z.B.: 'id':'my', 'href':'new.html')
addBody(content)	Fügt den „content“ an das „body“-Element des Widgets an
setBody(content)	Ersetzt den Inhalt des „body“-Elements mit dem „content“

Tabelle D.2.: Wichtige Eigenschaften und Methoden des „widget“-Objektes eines UWA-Widgets

sen. Die Event-Handler „widget.onRefresh“, „widget.onResize“ und „widget.onLoad“ sind ein paar Beispiele. Der „onLoad“-Handler ist nötig, da das normale „onload“-Event nicht ausgelöst wird. Zum „widget“-Objekt gibt es noch das „UWA“-Objekt. Dieses definiert vor allem einige Methoden zum dynamischen Nachladen von Daten, welche zusätzlich im „Data“-Objekt geschachtelt sind. Normale Ajax-Anfragen sollte ein Autor in seinem Widget nicht verwenden. Alle Methoden des „UWA.Data“-Objektes sind „getFeed“, „getJson“, „getFeed“, „getText“, „getXml“ und „request“. Die „request“-Methode bietet einen allgemeingültigen Ajax-Request an, welcher mit einigen Parametern konfigurierbar ist. Alle anderen genannten Methoden bauen auf der „request“-Funktion auf und dienen einer speziellen Datenanfrage. Die Parameter dieser sind jeweils eine URL, welche die nachzuladende Ressource adressiert und eine Rückruffunktion, welche bei erfolgreichem Laden der Zieldatei zur Ausführung kommt. Das bereits vorverarbeitete Ergebnis der Datenabfrage übergibt die genutzte Methode der Rückruffunktion als ersten Parameter. [30]

Außer den Methoden zum dynamischen Nachladen von Daten besitzt das UWA-Objekt die „extendElement(element)“-Methode. Der Parameter „element“ stellt ein DOM-Element (Document Object Model) des Widgets dar. Mit der Methode kann ein normales DOM-Element mit der UWA-Element-Erweiterung ergänzt werden. Alle mit „widget.createElement“ erstellten Elemente und das „body“-Element sind hingegen automatisch erweitert. Die UWA-Element-Erweiterung fügt dem jeweiligen DOM-Knoten einige Methoden hinzu. Diese sind in der Tabelle D.3 aufgeführt. Die

Bedeutung bezieht sich jeweils auf das Element, dessen Methode aufgerufen wird. [30]

Name	**Bedeutung**
addContent(content)	Fügt „content“ in ein „div“ am Ende des vorhandenen Inhalts an
appendText(text)	Fügt einen neuen Textknoten mit dem „text“ am Ende des vorhandenen Inhalts an
empty()	Leert den Inhalt
setContent(content)	Überschreibt den Inhalt mit einem „div“, welches „content“ enthält
setText(text)	Überschreibt den Inhalt mit einem Textknoten mit dem „text“
setHTML(html)	Überschreibt den Inhalt mit „html“-Inhalt (innerHTML)
setStyle(property, value)	Setzt die CSS-Eigenschaft „property“ mit dem Wert „value“
getDimensions()	Gibt ein JSON-Objekt mit Weite und Höhe des Elementes zurück
getElementsByClassName (className)	gibt eine Referenz auf alle Elemente zurück, welche mit dem gegebenen CSS-Klassennamen „className“ assoziiert sind
getElementsByTagName (tagName)	Gibt eine Referenz auf alle Elemente zurück, welche mit dem gegebenen „tagNamen“ assoziiert sind
getParent()	Gibt eine Referenz auf das Elternelement zurück
getChildren()	Gibt eine Referenz auf alle Kinderelemente zurück
hide()	Setzt die CSS-Anzeigeregel auf den Wert „none“ („display:none“)
show()	Setzt die CSS-Anzeigeregel auf einen leeren Wert (wie „display:inline“)
toggle()	Wechselt die CSS-Anzeigeregel zwischen „none“ und leeren Wert
remove()	Entfernt das Element aus dem DOM-Baum
hasClassName (className)	Gibt „true“ zurück, wenn das Element den „className“ besitzt, sonst „false“
addClassName (className)	Fügt den „className“ zu den bereits vorhandenen Klassennamen hinzu
removeClassName (className)	Entfernt den „className“ aus der Menge der Klassennamen
addEvent(event, callbackFunction)	Weißt dem Event-Handler „event“ eine neue „callbackFunction“ zu (z.B. für „click“)

Name	Bedeutung
removeEvent (event, callbackFunction)	Entfernt die „callbackFunction“ aus dem Event-Handler „event“

Tabelle D.3.: Auflistung aller Methoden eines DOM-Elementes, welches die UWA-Element-Erweiterung besitzt

Netvibes bietet zu den bisher genannten Funktionalitäten einige weitere an. Diese sind allerdings seltener genutzt und nicht zwingend benötigt. Das „UWA.Controls“-Objekt stellt dem Entwickler beispielsweise Bausteine, wie einen einfachen Audioplayer, zum schnelleren Entwickeln des Widgets bereit. Informationen zu vielen Funktionen der API stehen unter der UWA-JavaScript-Dokumentation[1] zum Abruf bereit. Detaillierte Informationen lassen sich über die UWA-Spezifikation[2] abrufen. Diese ist im Moment zwar etwas veraltet, aber dennoch sehr hilfreich.

Zusätzlich zur JavaScript-API existieren zwei internetdienstbasierte APIs von Netvibes. Über die REST-API sind Informationen über die aktuelle Nutzerumgebung, andere Widgets, Tabs, Zeitverlauf der Aktionen des Nutzers usw. abrufbar. Der grundlegende Endpunkt dieser ist „http://rest.netvibes.com“. Alle Daten lassen sich im XML- oder JSON-Format abrufen. Für den Zeitverlauf sind Daten zusätzlich in den Formaten RSS oder Atom anforderbar. Details sind über die Webseite zur Netvibes-Rest-API[3] und deren Unterseiten nachlesbar. Der zweite Internetdienst ist die Ecosystem-API. Er ermöglicht das Abrufen von Informationen über das sogenannte Netvibes Ecosystem. Einige Abrufe der API stehen frei zur Verfügung, allerdings sind einige auch privat und verlangen einen Schlüssel. Diesen bekommen nur Partner von Netvibes ausgestellt. Details stehen über die zugehörige Webseite[4] zum Abruf bereit. [29]

D.4. Nutzereinstellungen

Bei den Nutzereinstellungen wendet Netvibes ein ähnliches Verfahren wie Google an. Aus allen spezifizierten Präferenzen entsteht automatisch ein Einstellungsmenü. Dieses kann der Nutzer per Klick auf ein Symbol aufrufen und alle Parameter festlegen. Für das Definieren der Nutzereinstellungen existiert das „widget:preferences“-Element. Es stellt den Container für das Spezifizieren der Präferenzen und darf in jedem Widget maximal einmal auftreten. Die Kinder von „widget:preferences“ sind

[1] http://dev.netvibes.com/doc/uwa/documentation/javascript_framework
[2] http://documentation.netvibes.com/doku.php?id=uwa
[3] http://dev.netvibes.com/doc/api/rest
[4] http://dev.netvibes.com/doc/api/eco

die „preference“-Elemente. Von diesen darf der Autor beliebig viele verwenden. Jedes „preference“-Element definiert genau eine Präferenz. Das wichtigste Attribut des Elements ist „name“, dessen Wert stets einzigartig sein muss. Es stellt den Identifikator für die jeweilige Einstellung. Den Startwert kann der Entwickler mittels „defaultValue“-Attribut festlegen. Das „label“-Attribut ermöglicht ihm, jeder im Einstellungsmenü angezeigten Präferenz ein Label zu geben. Ist es nicht vorhanden, wird der Name als Label dargestellt. Mit dem „type“-Attribut legt der Autor den Typ der Präferenz fest. Die sechs möglichen Typen sind: „text“, „hidden“, „boolean“, „range“, „list“ und „password“. Der Standardtyp bei Abwesenheit des Attributs ist „text“. Eine Einstellung dieser Art repräsentiert ein „input“-Element des Typs „text“. Der Datentyp entspricht einer normalen Zeichenkette. Wenn der Wert des Attributs „hidden“ beträgt, handelt es sich um eine versteckte Einstellung. Eine solche Präferenz ist nicht nutzereditierbar. Deshalb findet auch keine Darstellung für diese im Menü statt. Alle „preference“-Elemente der Art „boolean“ präsentieren sich durch ein „input“-Tag vom Typ „checkbox“. Valide Werte für eine solche Nutzereinstellung sind „true“ oder „false“. Eine „password“-Präferenz verhält sich wie eine des Typs „string“ mit der Ausnahme, dass das Attribut „type“ den Wert „password“ erhält. Ein „preference“-Element der Art „list“ bildet eine Enumeration. Die möglichen Werte gibt der Entwickler der Aufzählung durch „option“-Elemente als Kind des entsprechenden „preference“-Elementes. Jede Option muss ein „value“-Attribut besitzen. Es spezifiziert einen möglichen Wert der Enumeration. Zusätzlich kann der Autor das „label“-Attribut verwenden, um dem Wert einen aussagekräftigen Namen zu geben. Die Darstellung einer Aufzählung erfolgt im Einstellungsmenü durch ein „select“-Element. Für jeden Enumerationswert erhält das „select“-Tag ein neues „option“-Kind. Der letzte Präferenz-Typ ist „range“. Er bildet wie „list“ eine Aufzählung. Allerdings müssen nicht alle Enumerationswerte von Hand definiert werden. Dafür eignet sich der „range“-Typ lediglich für Aufzählungen, welche aus ganzzahligen Zahlenfolgen mit jeweils gleichen Abständen der benachbarten Bestandteile zueinander bestehen. Zum Definieren der Werte besitzt das „preference“-Element die drei zusätzlichen Attribute „min“, „max“ und „step“. Das Attribut „min“ stellt die niederwertigste Zahl der Folge dar. Der Wert von „max“ repräsentiert die höchstwertige Nummer. Mit dem „step“-Attribut gibt der Autor die Abstände der Bestandteile der Zahlenfolge zueinander an. Die Werte der drei Attribute müssen jeweils ganzzahlig sein. Für „step“ gilt zusätzlich die Beschränkung auf positive Zahlen. [28]

Für den Zugriff auf die Präferenz mittels JavaScript bietet das „widget“-Objekt der JavaScript-API vier Methoden an. Drei davon ermöglichen das Laden des Wertes einer Einstellung. Sie unterscheiden sich jeweils im Datentyp des Rückgabeparameters. Die Methode „getValue“ gibt eine Zeichenkette zurück, „getBool“ führt zu einem booleschen Rückgabewert („true“ oder „false“) und „getInt“ liefert eine ganze Zahl zurück. Als Parameter erwarten alle drei genannten Funktionen jeweils den Identifikator einer Präferenz. Die letzte Methode ist für das Speichern eines neuen Wertes

zuständig. Ihr Aufruf-Name ist „setValue“. Sie erwartet zwei Parameter. Der Erste gibt den Namen der zu modifizierenden Nutzereinstellung an. Der zweite Parameter ist der neu zu setzende Wert. [30]

D.5. Lokalisierung

Für die Lokalisierung eines UWA-Widgets existiert kein vorgegebenes Verfahren. Die JavaScript-API bietet lediglich zwei Eigenschaften des „widget“-Objektes an, welche ein Autor für die Lokalisierung verwenden kann. Der Aufruf „widget.locale“ gibt das Land des Betrachters zurück. Der Rückgabewert ist eine Zeichenkette aus Kleinbuchstaben, die ein Land repräsentieren. Eine Auflistung aller möglichen Werte stellt eine Adresse[5] der Ecosystem-API bereit. Der Standardwert ist „us“. Die Eigenschaft „widget.lang“ gibt die vom Nutzer ausgewählte Sprache in Verbindung mit dem Land zurück. Der Teil für die Sprache ist mit einem Unterstrich vom Land getrennt. Zusätzlich ist die Abkürzung für das Land in Großbuchstaben geschrieben. Der erste Teil entspricht einem der Werte, die bei Browsern zur Angabe der Sprache in Verwendung sind. Der Standardwert ist „en_US“. Beide Eigenschaften sind in der Netvibes-Widget-Umgebung von den Plattformeinstellungen abhängig. [30]

[5] http://api.eco.netvibes.com/regions

Anhang E.

Beispiel für eine W3C-Konfigurationsdatei

```
<?xml version = "1.0" encoding = "UTF-8"?>
<widget xmlns = "http://www.w3.org/ns/widgets"
       xmlns:ex = "http://example.org/ex-ns"
       id = "http://example.org/exampleWidget"
       version = "2.0 Beta"
       height = "200"
       width = "200"
       viewmodes = "windowed floating"
       defaultlocale = "en-us">

  <access origin = "http://example.org:1337"/>

  <name short = "Example 2.0" xml:lang = "en-us">
    The example Widget!
  </name>

  <name short = "Example 2.0" xml:lang = "de">
    Das Beispiel-Widget!
  </name>

  <feature name = "http://example.com/camera">
    <param name = "autofocus" value = "true"/>
  </feature>

   <preference name = "skin"
              value = "alien"/>

  <preference name = "apikey"
            value = "ea31ad3a23fd2f"
            readonly = "true"/>

  <description>
    A sample widget to demonstrate some of the possibilities.
  </description>
```

```
35
36    <author href = "http://foo-bar.example.org/"
37            email = "foo-bar@example.org">Foo Bar Corp</author>
38
39    <icon width = "300"
40          height = "300"
41          src = "img/example.png"/>
42
43    <icon src = "img/big.png" ex:role = "big"/>
44
45    <content src = "myWidget.html"/>
46
47    <license>
48      Example license (based on MIT License)
49      Copyright (c) 2008 The Foo Bar Corp.
50      THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
        EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
        OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
        NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
        HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
        WHETHER IN AN ACTION OF CONTRACT, INSULT OR OTHERWISE, ARISING
        FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
        OTHER DEALINGS IN THE SOFTWARE.
51    </license>
52  </widget>
```

Listing E.1: Beispiel für eine Konfigurationsdatei eines W3C-Widgets in Anlehnung an das Beispiel aus der W3C-Widget-Spezifikation [5]

Anhang F.

Beispiel für eine Opera-Konfigurationsdatei

```
<?xml version = "1.0" encoding = "UTF-8"?>
<widget defaultmode = "widget"
        network = "public"
        dockable = "yes"
        version = "2.0 Beta">

  <widgetname>The example Widget!</widgetname>

  <feature name = "http://example.com/camera">
    <param name = "autofocus" value = "true"/>
  </feature>

  <description>
    A sample widget to demonstrate some of the possibilities.
  </description>

  <id>
    <host>http://example.org</host>
    <name>exampleWidget</name>
    <revised>2010-05</revised>
  </id>

  <width>200</width>
  <height>200</height>

  <icon width = "300"
        height = "300">img/example.png</icon>
  <icon>img/big.png</icon>

  <widgetfile>myWidget.html</widgetfile>

  <author>
    <name>Max Mustermann</name>
    <link>http://foo-bar.example.org/</link>
```

```
35      <email>foo-bar@example.org</email>
36      <organization>Foo Bar Corp</organization>
37    </author>
38
39    <security>
40      <access>
41        <host>example.com</host>
42        <host>example.org</host>
43        <path>/good</path>
44        <port>2048-4906</port>
45        <port>80,1337</port>
46      </access>
47    </security>
48  </widget>
```

Listing F.1: Beispiel für eine Konfigurationsdatei eines Opera-Widgets

Anhang G.

Beispiel für ein iGoogle-Gadget

```
<?xml version = "1.0" encoding = "UTF-8"?>
<Module>
  <ModulePrefs title = "__MSG_loc_title__"
               title_url = "http://example.org/exampleWidget"
               description = "A sample widget to demonstrate some of
               the possibilities."
               screenshot = "http://example.org/screenshot.jpg"
               thumbnail = "http://example.org/thumbnail.jpg"
               author = "Max Mustermann"
               author_affiliation = "Foo Bar Corp"
               author_link = "http://foo-bar.example.org/"
               author_email = "foo-bar@example.org">

    <Require feature = "http://example.com/camera">
      <Param name = "autofocus">true</Param>
    </Require>

    <Icon>http://example.org/example.png</Icon>
  </ModulePrefs>

  <UserPref name = "viewername"
            display_name = "your name"
            required = "true"/>

  <UserPref name = "apikey"
            datatype = "hidden"
            default_value = "ea31ad3a23fd2f"/>

  <UserPref name = "fontsize"
            display_name = "font size"
            datatype = "enum"
            default_value = "9">
    <EnumValue value = "9" display_value = "Small"/>
    <EnumValue value = "12" display_value = "Medium"/>
```

```
34      <EnumValue value = "18" display_value = "Big"/>
35    </UserPref>
36
37    <Locale lang="en" country="us" >
38      <msg name="loc_title">The example Widget!</msg>
39    </Locale>
40    <Locale lang="de" messages="http://example.org/de_ALL.xml" />
41
42    <Content type = "html" view = "home">
43      <![CDATA[
44      small content
45      ]]>
46    </Content>
47
48    <Content type = "html" view = "canvas">
49      <![CDATA[
50        big content
51      ]]>
52    </Content>
53  </Module>
```

Listing G.1: Beispiel für ein iGoogle-Gadget

Anhang H.

Beispiel für ein UWA-Widget

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns = "http://www.w3.org/1999/xhtml"
      xmlns:widget = "http://www.netvibes.com/ns/">
  <head>
    <meta name = "author" content = "Foo Bar Corp"/>
    <meta name = "website" content = "http://foo-bar.example.org/"/>
    <meta name = "email" content = "foo-bar@example.org"/>
    <meta name = "description"
          content = "A sample widget to demonstrate some of the
          possibilities."/>
    <meta name = "version" content = "2.0 Beta"/>
    <meta name = "apiVersion" content = "1.0"/>
    <meta name = "autoRefresh" content = "20"/>
    <meta name = "debugMode" content = "true"/>
    <meta name = "screenshot"
          content = "http://example.org/screenshot.jpg"/>
    <meta name = "thumbnail"
          content = "http://example.org/thumbnail.jpg"/>

    <link rel = "stylesheet"
          type = "text/css"
          href = "http://www.netvibes.com/themes/uwa/style.css"/>
    <script type = "text/javascript"
            src = "http://www.netvibes.com/js/UWA/load.js.php?
            env=Standalone"></script>

    <title>The example Widget!</title>

    <link rel = "icon"
          type = "image/png"
          href = "http://www.example.com/example.png"/>
    <link rel = "rich-icon"
```

```
32          type = "image/png"
33          href = "http://www.example.com/big.png"/>
34
35      <widget:preferences>
36        <preference type = "text"
37                    name = "viewername"
38                    label = "your name"/>
39
40        <preference type = "range"
41                    name = "fontsize"
42                    step = "1"
43                    min = "8"
44                    max = "18"
45                    defaultValue = "12"
46                    label = "font size"/>
47
48        <preference type = "hidden"
49                    name = "apikey"
50                    defaultValue = "ea31ad3a23fd2f"/>
51
52        <preference type = "list"
53                    name = "skin"
54                    label = "skin"
55                    defaultValue = "classic">
56          <option value = "alien" label = "alien"/>
57          <option value = "darkblue" label = "dark blue"/>
58          <option value = "classic" label = "classical"/>
59        </preference>
60      </widget:preferences>
61    </head>
62
63    <body>
64        ... normaler HTML-Inhalt ...
65    </body>
66  </html>
```

Listing H.1: Beispiel für ein UWA-Widget