Universität des Saarlandes
Fachrichtung 6.2 Informatik

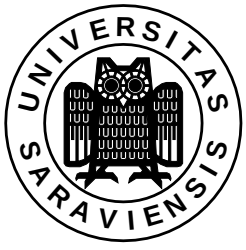
Fuzz Testing

Zufallstesten von Programmen, Diensten und Handys

Grobentwurf

Phase	Phasenverantwortlicher	E-Mail
Pflichtenheft	Stephan Schlicker	stephanschlicker@web.de
Grobentwurf	Holger Hewener	holger@hewener.net
Spezifikation	Andreas Schlicker	andreasschlicker@web.de
Implementierung	Dominik Gummel	bofh@krypt1.cs.uni-sb.de
Testbericht	Sascha Kiefer	sk@intertivity.com
Software	Markus Uhl	markus.uhl@gmx.de

21. Juli 2003



Inhaltsverzeichnis

1 Vorwort

Zu diesem Grobenturf des Softwareprojektes „wmlgen“ existiert ein Prototyp im Gruppen - CVS, welcher die Machbarkeit dieses präsentierten Designs zeigen soll.
Für weitere Informationen zur Funktionalität des Prototyps lesen Sie bitte Kapitel **??**: „Informationen zu Prototyp 1“.
Alle in diesem Dokument vorgestellten Diagramme finden sie auch unter http://www.hewener.net/st.

2 Einleitung

In den Abbildungen **??** bis **??** erkennt man die grobe Aufteilung des Systems in seine einzelne Komponenten:

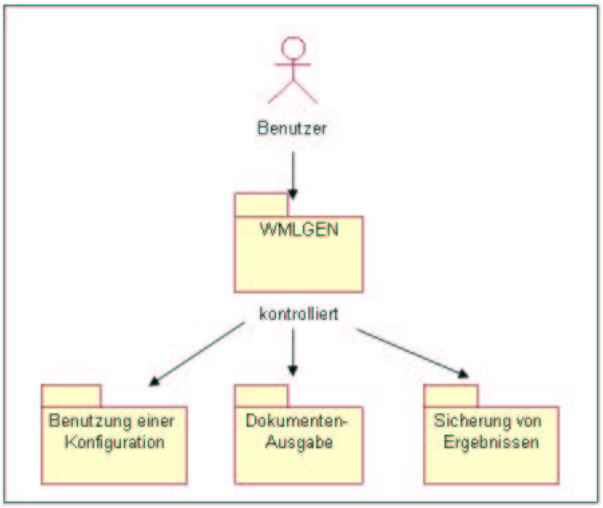


Abbildung 1: Grobgliederung bei lokaler Anwendung

In Abbildung **??** nutzt der Benutzer lokal auf seinem Rechner das Softwaresystem direkt. Das System erzeugt bei einer solchen lokalen Anwendung durch die Ausführung der Datei „wmlgen“ die Ausgabe eines WML - Dokumentes auf der Standardausgabe. Diese Ausgabe kann dann benutzt werden, um Browser- oder Mikrobrowsersoftware durch dieses erzeugte Dokument auf Kompatibilität und Konformität mit dem WML - Standard 1.3 zu testen.
Hierbei besteht dieses Szenario einzig aus dem hier vorgestellte System, indem das Programm die Benutzung einer Konfiguration, die Dokumentenausgabe und die Sicherung von Ergebnissen organisiert und steuert.

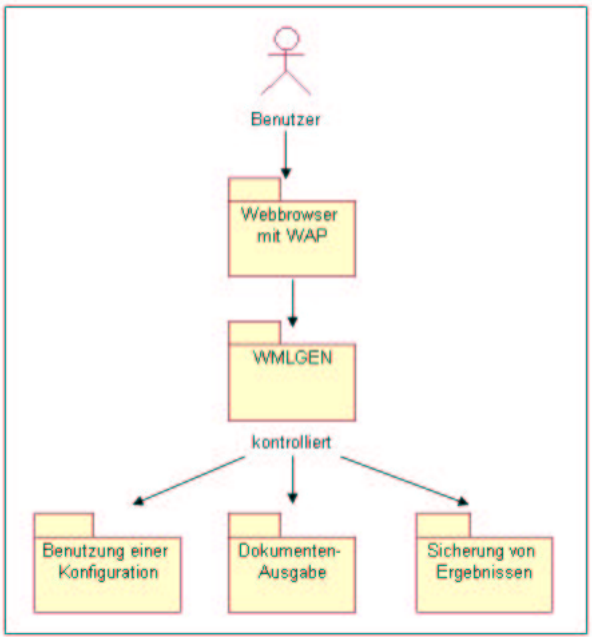


Abbildung 2: Grobgliederung bei Anwendung im Emulator

Alternativ lässt sich die Software in einen Webserver integrieren (Abbildung **??**), so dass die Erzeugung dieser Dokumente mit deren Abruf vom Webserver durch eine Browsersoftware erfolgt. Diese Testmöglichkeit eignet sich, um Fehlimplementierungen von Mikrobrowser im Massentest herauszufinden.
Eine solche Implementierung ist z.B. mittels CGI - Script in Perl zu realisieren.
Die Schnittstelle zwischen Benutzer und Programm (hier: Webbrowser mit WAP) muss anderweitig bezogen werden, da dies nicht Bestandteil dieses Softwaresystems ist.

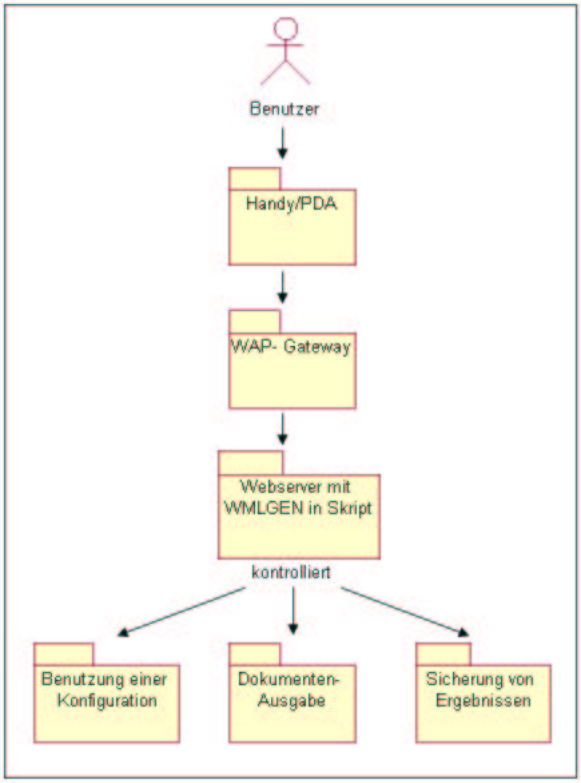


Abbildung 3: Grobgliederung bei Anwendung im Endgerät

Die gleiche Funktionalität bietet die Testmöglichkeit direkt im Endgerät (Handy, PDA,..) zu testen (Abbildung **??**), ist aber nur noch durch die Instanz des Gateways erweitert, welches die Dienste von öffentlichen Rechnern im Mobilfunknetz bereitstellt.

Auch hier wird die Schnittstelle zwischen Benutzer und System (hier: Handy/PDA und WAP - Gateway) nicht weiter betrachtet, da sie nicht Bestandteil dieses Systems sind.

Von allen drei Szenarien wird im folgenden der untere Teil, „wmlgen“ und die Punkte „Benutzung einer Konfiguration“, „Dokumentenausgabe“ und „Sicherung von Ergebnissen“, beschrieben. Für alle weiteren Informationen bezüglich der Szenarien lesen sie bitte die Spezifikation und deren Testfälle.

3 Funktionale Analyse des Systems

In Abbildung **??** erkennt man die grobe Aufteilung des Systems in Funktionen. Das Programm stellt drei grundlegende Funktionsbereiche zur Verfügung:

- die Verwaltung von Ein- und Ausgabedaten
- die Verarbeitung einer Konfiguration
- das Erstellen des Dokumentes zur Ausgabe

Diese drei Funktionsbereiche werden im folgenden feiner unterteilt:

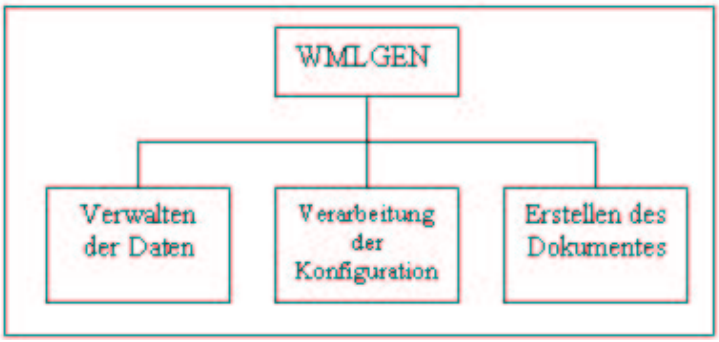


Abbildung 4: Funktionsbaum - Übersicht

In den Abbildungen **??**, **??** und **??** werden die einzelnen Teile des Funktionsbaums erklärt. Dabei beziehen sich die Angegebenen Verweise (/Fxxx/) auf die im Pflichtenheft dokumentierten Funktionen der Software.

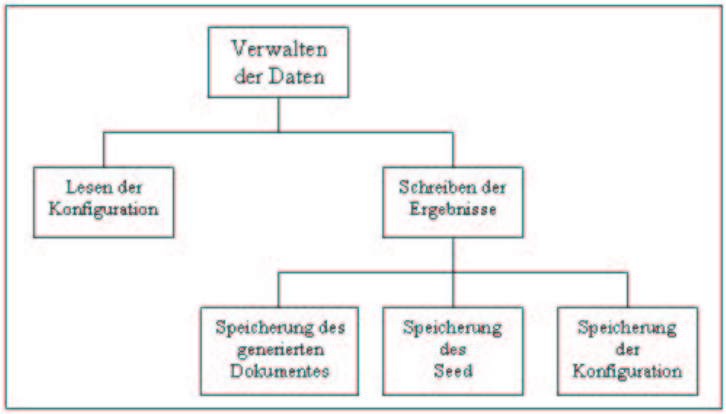


Abbildung 5: Funktionsbaum - Verwalten der Daten

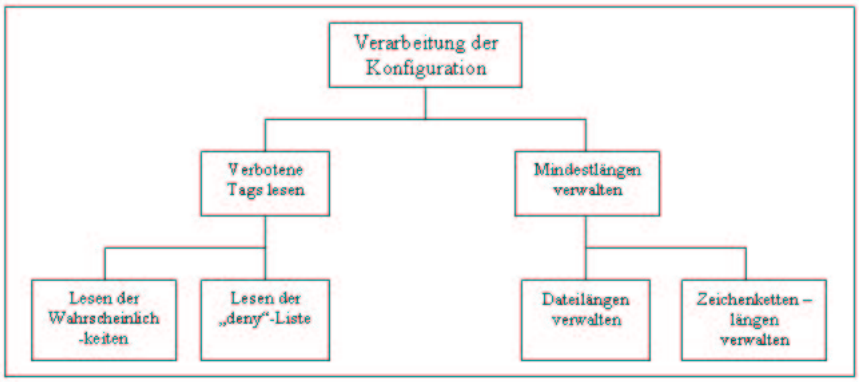


Abbildung 6: Funktionsbaum - Verarbeitung der Konfiguration

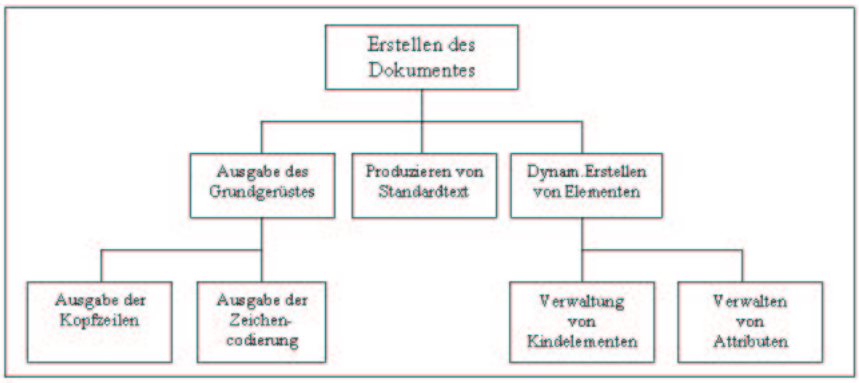


Abbildung 7: Funktionsbaum - Erstellen des Dokumentes

4 Klassenbeschreibungen

Zur Darstellung der Klassen wird die UML - Notation verwendet.
Alle Abbildungen befinden sich zudem auch im Gruppen CVS zum Download und zur Ansicht in einer größeren Auflösung. Außerdem befinden sich in den dort vorliegenden Dia-Diagrammen alle Attribute und Methoden aller Klassen, die hier z.T. weggelassen wurden, um die Diagramme darstellen zu können.

4.1 Kontrollstrukturen

In Abbildung **??** wird der Aufbau der Kontrollstrukturen des Systems beschrieben.

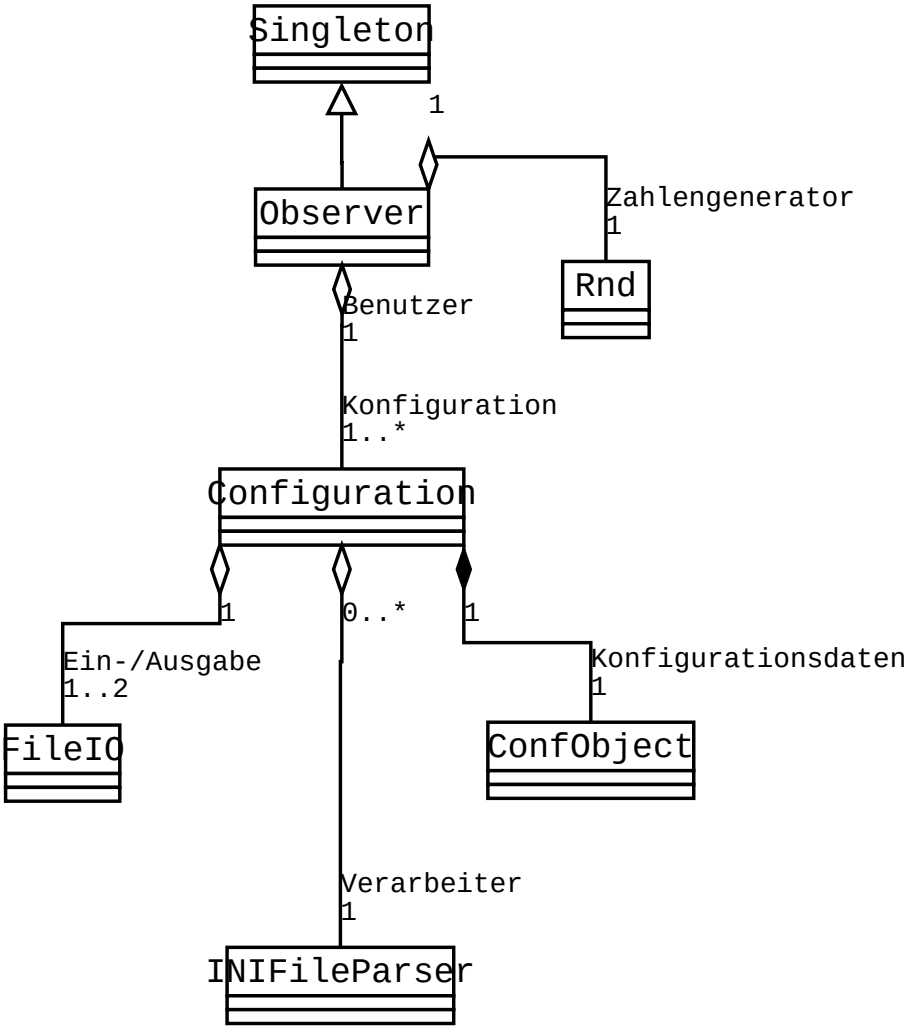


Abbildung 8: UML-Diagramm der Kontrollstrukturen

Die Zentrale des Softwaresystems ist die Klasse Observer, welche ähnlich dem gleichnamigen Design Pattern funktioniert. Der Observer ist die Struktur, die zwischen den verschiedenen Klassen vermittelt und als zentrale Anlaufstelle für allgemeine Operationen dient. Somit brauchen sich die Objekte, welche die Dienste des Observers nutzen, nicht untereinander zu kennen.
Während der Programmausführung werden die Kontrollstrukturen hauptsächlich am Anfang verwendet, um Konfigurationen zu laden und zu verarbeiten, und um in späteren Programmzuständen Konfigurationsinformationen zur Verfügung stellen zu können.

4.1.1 Die Klasse Singleton

Die Singleton - Klasse bietet die gleiche Funktionalität wie in der Fachliteratur beschrieben:
Durch nichtöffentlichen Konstruktor und statische Methoden und Attribute sichert diese Klasse, dass zu jeder Zeit die Anzahl an Instanzen einer Klasse, die direkt vom Pattern Singleton abgeleitet wurde, höchstens 1 ist.

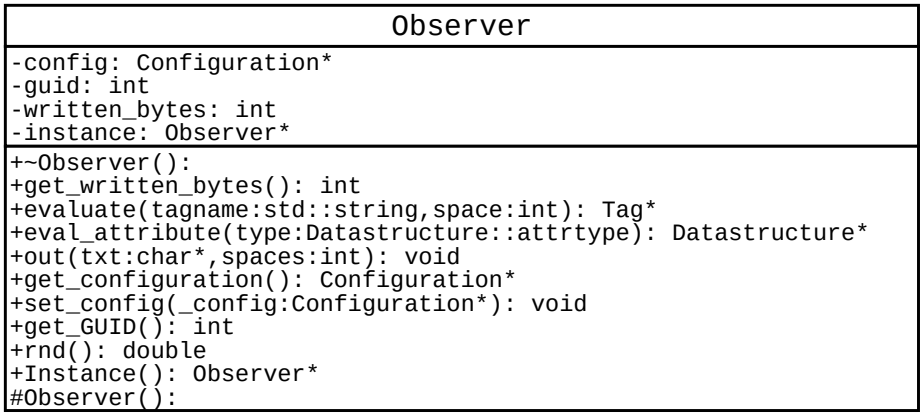


Abbildung 9: UML-Diagramm des Oberservers

4.1.2 Die Klasse Observer

Der Observer ist als Singleton implementiert und bietet Funktionalitäten, welche von annähernd jeder Klasse genutzt werden kann. Dazu haben fast alle Klassen direkt (über einen gespeicherten Pointer) oder indirekt (über einen Pointer in den Memberattributen) Zugriff auf die Instanz des Observers.
Er bietet allgemeine Funktionen wie das Generieren von im Programm einzigartigen Zahlen mit der statischen Funktion get_GUID() und der privaten statischen Ganzzahlvariable guid.
Ebenso kann er Zufallszahlen generieren, welche in diesem System öfter benötigt werden. Aus diesem Grund gibt es hierfür eine eigene Klasse RND mit statischen Zugriffsfunktionen.
Der Observer besitzt einen Verweis auf die Konfiguration, so dass auch jedes Element mit Zugriff auf den Observer auch Informationen aus der Konfiguration lesen kann.

Hierzu dienen die Funktionen get_configuration() und set_configuration (Configuration*).
Alle Programmausgaben bezüglich des erzeugten Programms laufen über die Funktion out(char* text, int spaces), welche die Ausgabe auf die Standardausgabe und gegebenenfalls auch in die Sicherungsdatei (mittels der Klasse FileIO) bietet. Dabei dient die Ganzzahlangabe lediglich der Formatierung der Ausgabe zur optisch besseren Lesbarkeit.
Mit dieser Funktion werden auch die Anzahl der ausgegebenen Bytes gezählt, so dass durch die Variable written_bytes die Angabe von Mindestausgabegrössen eingehalten werden kann.
Der Observer bietet durch die Funktion evaluate die Möglichkeit, eine Instanz eines WML - Elementes zu erzeugen, indem man der Funktion den Namen des Elementes als Parameter vom Typ string übergibt. Diese Instanz wird als Tag - Pointer zurückgegeben.
Ebenso kann mittels eval_attribute und Übergabe des Attributesnamens und -typs (mittels Datastructure::attrtype) eine Instanz eines Attributes erzeugt werden.
Beide Funktionen erzeugen die Daten durch Nachschlagen in einer Art Lookuptable.

4.1.3 Die Klasse Rnd

Die Klasse Rnd dient dem Erstellen von Zufallszahlen.
Da „wmlgen“ gewährleistet, auf jedem unterstützten Betriebssystem (siehe Pflichtenheft) durch Angabe einer Konfiguration und eines SEED - Patterns, identische WML - Dokumente zu Erzeugung, ist es nicht möglich die Zufallsmethoden der Standard C Library (stdlib) zu nutzen.
Rnd stellt dem Programmierer zwei statische Funktionen zur Verfügung, die im Grunde die selben Funktionalitäten besitzen, wie ihre Synonyme der stdlib:

- void Rnd::Srand(unsigned long seed) initialisiert den Zufallsgenerator
- unsigned long Rnd::Rand(void) erzeugt eine 32-bit Pseudo - Zufallszahl

Der Zufallsgenerator basiert auf dem Tiny Encryption Algorithm (TEA).

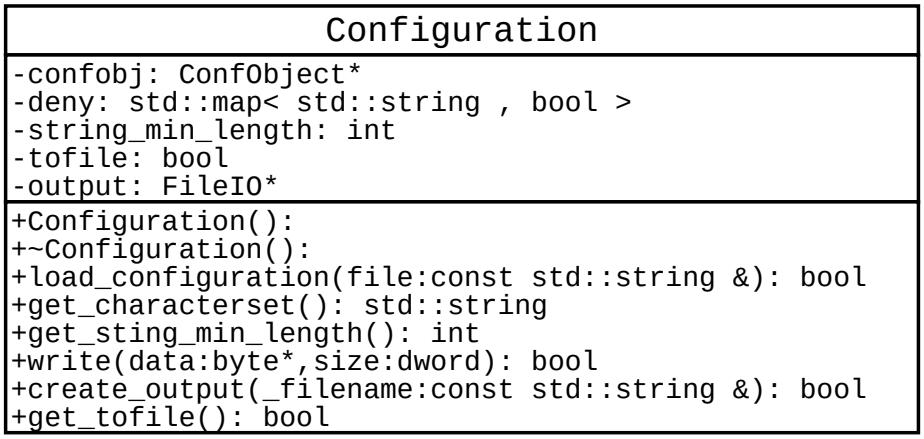


Abbildung 10: UML-Diagramm der Klasse Configuration

4.1.4 Die Klasse Configuration

Die Konfiguration speichert Informationen, welche aus der Parameterübergabe und der somit spezifizierten Konfigurationsdatei gelesen werden können.
Solche Informationen wären die minimalen Datei- und Zeichenkettenlängen, der für diese Zeichenketten verwendete Zeichensatz und die Tatsache, ob Ausgaben auch in Dateien geschrieben werden sollen.
Für weitere Werte, welche aus einzelnen Sektionen der Konfiguration gelesen wurden, existiert in der Konfiguration eine Instanz eines Konfigurationsobjektes (ConfObject).

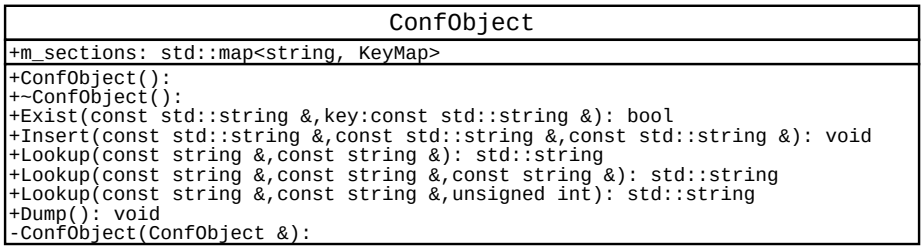


Abbildung 11: UML-Diagramm des ConfObjects

4.1.5 Die Klasse ConfObject

Die Klasse ConfObject speichert mit Hilfe von einer Nachschlagetabelle Werte für die Konfiguration eines WML - Dokumentes. Dabei ist die Basistabelle in Sektionen eingeteilt. Eine neue Sektion wird in der Konfigurationsdatei mit in eckigen Klammern ([. . .]) eingefasste Sektionsnamen eingeleitet. Jede Sektion hat wiederum verschiedene Schlüssel - Wert Paare, die später als Parameter in die WML - Dokumentenerzeugung mit einfließen. Mit Hilfe verschieden parametisierter lookup Funktionen kann der Programmierer zu folgenden Ergebnisse gelangen:

- Nachschlagen aller Schlüssel - Wert Paare in einer Sektion
- Nachschlagen eines bestimmten Wertes in einer bestimmten Sektion unter einem festgelegten Schlüssel
- Enumerierung von mehreren Werten zu einem Schlüssel in einer Sektion, falls mehr als ein Wert zu einem Schlüssel passt.

Die Funktion insert dient zum Einfügen von Schlüssel - Wert Paaren und zum Anlegen zusätzlicher Sektionen.

4.1.6 Die Klasse FileIO

Mit Hilfe von FileIO lassen sich standardisierte Filesystemoperationen durchführen. Somit ist es möglich Dateien entweder zum lesen oder zum schreiben zu öffnen (Open), Daten in zum Schreiben geöffnete Dateien zu speichern (Write) oder Daten aus zum Lesen geöffneten Dateien zu erhalten(Read) und geöffnete Dateien wieder zu schließen (Close).
Daten werden mit Hilfe des Typs unsigned char ausgetauscht.

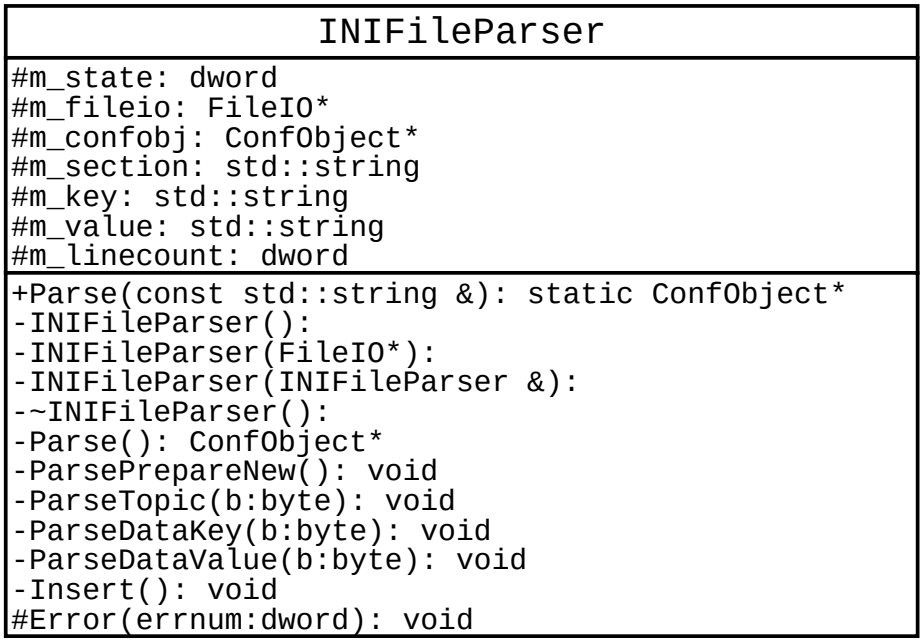


Abbildung 12: UML-Diagramm des INIFileParsers

4.1.7 Die Klasse INIFileParser

Die Klasse INIFileParser besitzt im wesentlichen nur eine statische Funktion Parse, welche als Parameter eine Zeichenkette erhält, die die Pfad- und Datennamenbeschreibung der Konfigurationsdatei enthält. Diese Funktion öffnet die Datei, parst den Inhalt und liefert ein daraus resultierendes ConfObject zurück.
Bei eventuellen syntaktischen Fehlern in der Konfigurationsdatei, gibt der Parser entsprechende Fehlermeldungen auf dem standard error stream aus.
Die Ausgabe wird im folgenden Format sein:

Fehlertext[Line x]

Wobei x die Fehlerzeilenummer der Datei sein wird.
Genaue Fehlertext Formulierungen liegen momentan noch nicht vor.

4.2 Elemente (Tags)

In Abbildung **??** wird der Aufbau der Tag-Elemente des Systems beschrieben.

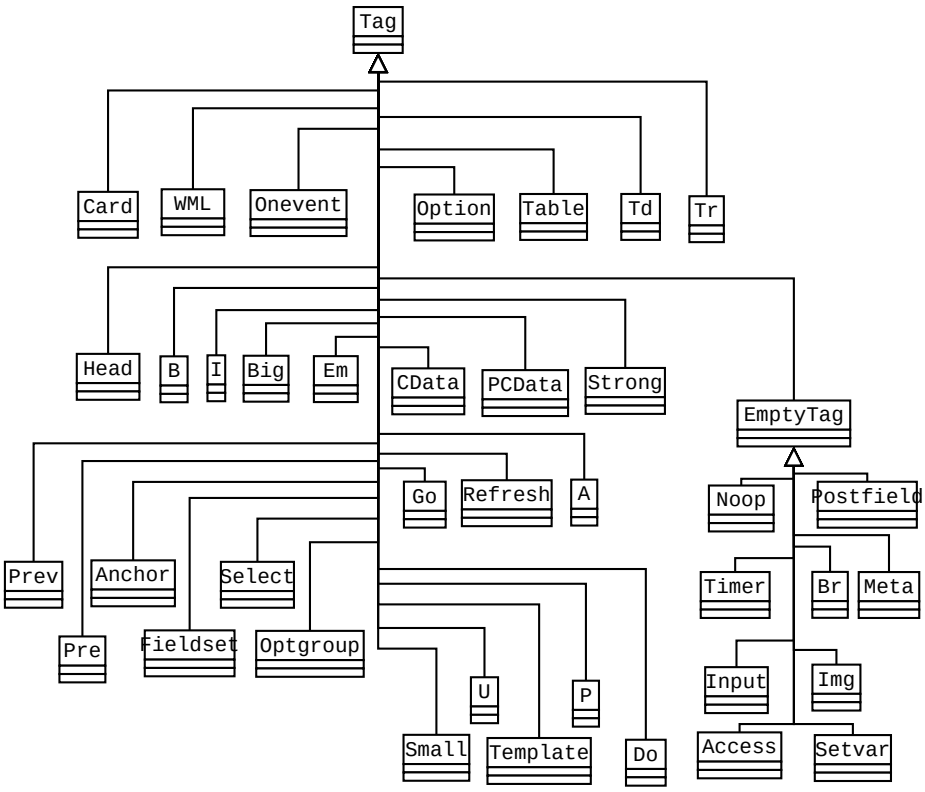


Abbildung 13: UML-Diagramm Elemente (Tags)

Da Abbildung **??** viel zu umfangreich ist, um im akzeptabler Grösse dargestellt werden zu können, verweise ich hier nochmals auf die Originalbilddatei, welche für den Leser im Gruppen - CVS gespeichert ist und fasse dieses Konzept nochmals zusammen:
Die einzelnen Tags des WML Dokumentes werden durch jeweils eine Klasse repräsentiert, wobei alle Tag-Klassen von einer virtuellen Klasse „Tag“ abgeleitet sind. Die einzigen Ausnahmen sind die Tags, welche im WML - Standard keine weiter untergeordneten Tags besitzen. Diese werden auch leere Tags genannt und sind direkt von der abstrakten Klasse „EmptyTag“ abgeleitet, welche wiederum von „Tag“ abgeleitet ist.
Diese EmptyTags bieten bisher keine weitere oder eingeschränkte Funktionalität, einzig und allein die Implementierung ihrer Ausgabefunktion ist minimal abgeändert. So bietet die Klasse aber eine bessere Übersicht.

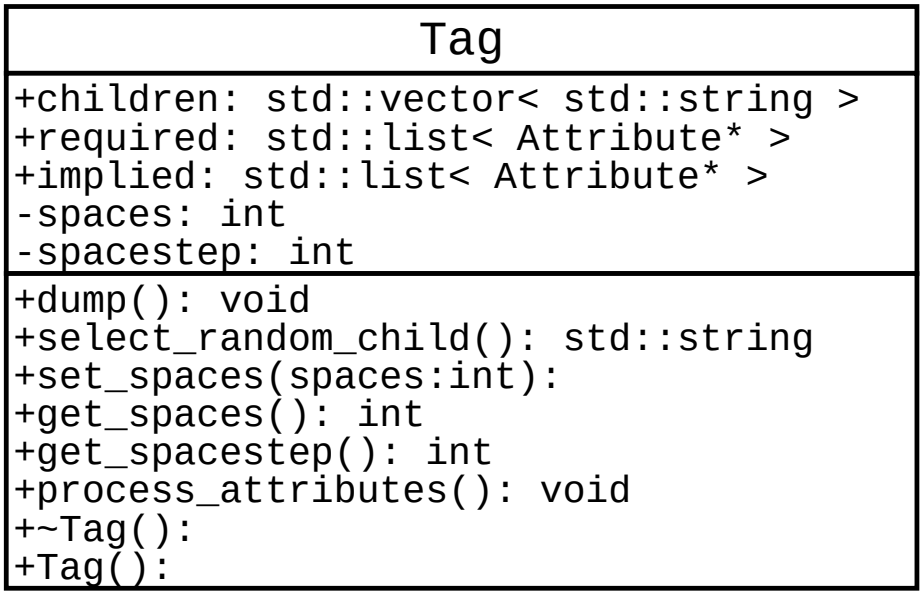


Abbildung 14: UML-Diagramm der Klasse Tag

4.2.1 Die Klasse Tag

Die virtuelle Klasse Tag ist die Vorlage der einzelnen WML - Elemente. Sie schreibt durch die virtuelle Funktion dump vor, dass jedes Tag seine eigene Art und Weise der Ausgabe bestimmt.
In den Listen (std::list< Attribute* >) required und implied werden die von einem Tag benötigten und änderbaren Attribute gespeichert, welche bei Aufruf der dump - Funktion abgearbeitet werden.
Ebenso speichert der Vektor (std::vector< std::string >) children die möglichen Unterelemente dieses Tags (mögliche child-Tags).
Aus diesem Vektor ausgewählt wird ein Element per Zufall mittels std::string select_random_child(), so dass man diesen String mittels des Observer in einen Tag* evaluieren kann.
Die Funktionen get_spaces und get_spacestep dienen der Formatierung, so dass jedes Tag seine Einrücktiefe im Dokument kennt. Dadurch ist auch jedem Tag bekannt, wie tief die Ausgabe schon vorgedrungen ist.
Die Ausgabe der Tag - eigenen Attribute geschieht mittels der Funktion process_attributes, welche die Attribute der required - Liste komplett abarbeitet und eventuell Elemente aus der implied - Liste ausgibt.
Analog ist process_children für die Abarbeitung/Erstellung von möglichen Kindern verantwortlich.

4.2.2 Die Klasse EmptyTag

Diese Klasse bietet keine erweiterte Funktionalität und erbt somit die Eigenschaften von „Tag“.
Von EmptyTag abgeleitete Klassen besitzen nur eine gering geänderte dump - Funk-

tion, da bei diesen Tags die Ausgabe des Abschlusstags fehlt und in das Anfangstag integriert wird.

4.2.3 Die Klasse WML

Diese Klasse repräsentiert das WML - Tag, welches direkt nach der Einleitung durch XML Version und Doctype Definition beginnen muss und das restliche Dokument komplett umspannt.
Es kann nur ein einziges Tag dieser Art in einem Deck geben, und es trägt neben den Standardattributen eine Sprachdefinition.
DTD-Definition:

```
<!ELEMENT wml ( head?, template?, card+ )>
<!ATTLIST wml
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.4 Die Klasse Card

Diese Klasse repräsentiert eine einzelne Karte innerhalb des WML - Decks. Jedes Deck muss mindestens eine Instanz einer solchen Karte besitzen.
Nach diesem Tag folgt der jeweilige Inhalt, welcher eine Karte auszeichnet.
DTD-Definition:

```
<!ELEMENT card (onevent*, timer?, (do | p | pre)*)>
<!ATTLIST card
          title %vdata; #IMPLIED
          newcontext %boolean; "false"
          ordered %boolean; "true"
          xml:lang NMTOKEN #IMPLIED
          %cardev;
          %coreattrs; >
```

4.2.5 Die Klasse Do

Diese Klasse repräsentiert die Bindung einer Reaktion an ein Steuerelement in einer Karte. Somit lassen sich Knöpfe mit speziellen Funktion belegen wie z.B. mit einer Menuführung.
DTD-Definition:

```
<!ELEMENT do (%task;)>
<!ATTLIST do
          type CDATA #REQUIRED
          label %vdata; #IMPLIED
          name NMTOKEN #IMPLIED
          optional %boolean; "false"
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.6 Die Klasse Onevent

Diese Klasse repräsentiert die Reaktion auf Ereignisse in einem WML - Stapel.
DTD-Definition:

```
<!ELEMENT onevent (%task;)>
<!ATTLIST onevent
          type CDATA #REQUIRED
          %coreattrs; >
```

4.2.7 Die Klasse Head

Diese Klasse repräsentiert die Dokumenteninformationen. Es darf nur einmal pro WML - Dokument auftreten und muss das erste Element nach dem öffnenden WML - Tag sein.
In den Kinderelementen werden sich die Metainformationen befinden.
Diese Klasse besitzt nur die Standardattribute.
DTD-Definition:

```
<!ELEMENT head ( access | meta )+>
<!ATTLIST head
          %coreattrs; >
```

4.2.8 Die Klasse Template

Diese Klasse repräsentiert eine Aktionsschablone mit der man Einheitlichkeit in der Nutzungsoberfläche eines WML - Dokumentes erzeugen kann.
DTD-Definition:

```
<!ELEMENT template (%navelmts;)*>
<!ATTLIST template
          %cardev;
          %coreattrs; >
```

4.2.9 Die Klasse Access

Diese Klasse repräsentiert eine Zugriffsbeschränkung, welche den Zugriff von nicht explizit angegebenen Seiten sperren kann.
Dies ist ein Tag ohne weitere Kinder und somit direkt von „EmptyTag“ abgeleitet.
DTD-Definition:

```
<!ELEMENT access EMPTY>
<!ATTLIST access
          domain CDATA #IMPLIED
          path CDATA #IMPLIED
          %coreattrs; >
```

4.2.10 Die Klasse Meta

Diese Klasse repräsentiert die Markierung von Metainformationen für das Dokument. Neben den Standardattributen gibt es Attribute für das verwendete Metadatenschema, den Namen der Metainformation und den Inhalt des Metadatums.
Dies ist ein Tag ohne weitere Kinder und somit direkt von „EmptyTag“ abgeleitet.
DTD-Definition:

```
<!ELEMENT meta EMPTY>
<!ATTLIST meta
          http-equiv CDATA #IMPLIED
          name CDATA #IMPLIED
          forua %boolean; "false"
          content CDATA #REQUIRED
          scheme CDATA #IMPLIED
          %coreattrs; >
```

4.2.11 Die Klasse Go

Diese Klasse repräsentiert ein Navigationselement. So navigiert das WAP - Gerät zu einer neuen Karte und evtl. zu einem neuen Stapel.
DTD-Definition:

```
<!ELEMENT go (postfield | setvar)*>
<!ATTLIST go
          href %HREF; #REQUIRED
          sendreferer %boolean; "false"
          method (post|get) "get"
          enctype %ContentType; "application/x-www-form-urlencoded"
          cache-control %cache-control; #IMPLIED
          accept-charset CDATA #IMPLIED
          %coreattrs; >
```

4.2.12 Die Klasse Prev

Diese Klasse repräsentiert eine Navigation entlang der Nutzungsgeschichte. Dies entspricht der vom Webbrowser bekannten „Zurück“ Navigation.
DTD-Definition:

```
<!ELEMENT prev (setvar)*>
<!ATTLIST prev
           %coreattrs; >
```

4.2.13 Die Klasse Refresh

Diese Klasse repräsentiert ein erneutes Anzeigen der aktuellen Karte mit veränderten Variablen und somit anderen Seiteninhalten.
DTD-Definition:

```
<!ELEMENT refresh (setvar)*>
<!ATTLIST refresh
          %coreattrs; >
```

4.2.14 Die Klasse Noop

Diese Klasse repräsentiert ein Element, welches einfach nichts tut. Es dient dazu Bindungen auf Stapelebene individuell auf Kartenebene wieder auszuschalten.
Dies ist ein Tag ohne weitere Kinder und somit direkt von „EmptyTag“ abgeleitet.
DTD-Definition:

```
<!ELEMENT noop EMPTY>
<!ATTLIST noop
          %coreattrs; >
```

4.2.15 Die Klasse Postfield

Diese Klasse repräsentiert Daten, die mit einer Seitenanforderung automatisch mit an den Server gesendet werden soll. So lassen sich Variablen durch ihren Namen spezifizieren und mit einem Wert belegen.
Dies ist ein Tag ohne weitere Kinder und somit direkt von „EmptyTag“ abgeleitet.
DTD-Definition:

```
<!ELEMENT postfield EMPTY>
<!ATTLIST postfield
          name %vdata; #REQUIRED
          value %vdata; #REQUIRED
          %coreattrs; >
```

4.2.16 Die Klasse Setvar

Diese Klasse repräsentiert eine Änderung von Vaiablen als Nebeneffekt einer Ereignisbehandlung.
Dies ist ein Tag ohne weitere Kinder und somit direkt von „EmptyTag“ abgeleitet.
DTD-Definition:

```
<!ELEMENT setvar EMPTY>
<!ATTLIST setvar
          name %vdata; #REQUIRED
          value %vdata; #REQUIRED
          %coreattrs; >
```

4.2.17 Die Klasse Select

Diese Klasse repräsentiert eine Auswahlliste, bei der der Benutzer einen vorgegebenen Wert selektiert.
Das Kind-Tag Option repräsentiert hierbei eine Auswahlmöglichkeit.
DTD-Definition:

```
<!ELEMENT select (optgroup|option)+>
<!ATTLIST select
          title %vdata; #IMPLIED
          name NMTOKEN #IMPLIED
          value %vdata; #IMPLIED
          iname NMTOKEN #IMPLIED
          ivalue %vdata; #IMPLIED
          multiple %boolean; "false"
          tabindex %number; #IMPLIED
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.18 Die Klasse Optgroup

Diese Klasse repräsentiert eine logische Gruppierung von Auswahlmöglichkeiten. Sie dient als Zusammenfassung mehrerer Optionen, die vom Microbrowser geeignet bei der Darstellung genutzt werden können.
DTD-Definition:

```
<!ELEMENT optgroup (optgroup|option)+ >
<!ATTLIST optgroup
          title %vdata; #IMPLIED
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.19 Die Klasse Option

Diese Klasse repräsentiert eine Auswahlmöglichkeit, wobei das Element jeweils die darzustellende Beschreibung der Option umfasst.
DTD-Definition:

```
<!ELEMENT option (#PCDATA | onevent)*>
<!ATTLIST option
          value %vdata; #IMPLIED
          title %vdata; #IMPLIED
          onpick %HREF; #IMPLIED
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.20 Die Klasse Input

Diese Klasse repräsentiert die Eingabe von Text oder anderen Zeichenfolgen. Der Benutzer kann es auswählen und dann mit einer geeigneten Eingabemöglichkeit benötigte Daten eingeben.
DTD-Definition:

```
<!ELEMENT input EMPTY>
<!ATTLIST input
          name NMTOKEN #REQUIRED
          type (text|password) "text"
          value %vdata; #IMPLIED
          format CDATA #IMPLIED
          emptyok %boolean; #IMPLIED
          size %number; #IMPLIED
          maxlength %number; #IMPLIED
          tabindex %number; #IMPLIED
          title %vdata; #IMPLIED
          accesskey %vdata; #IMPLIED
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.21 Die Klasse Fieldset

Diese Klasse repräsentiert logisch zusammengehörende Felder und Text. Durch das Fieldset können Daten gruppiert werden.
DTD-Definition:

```
<!ELEMENT fieldset (%fields; | do)* >
<!ATTLIST fieldset
          title %vdata; #IMPLIED
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.22 Die Klasse Timer

Diese Klasse repräsentiert eine Alarmuhr, welche so definiert auch auf einen Wert gestellt werden kann. Es darf nur innerhalb von <card> verwendet werden und dort höchstens einmal auftreten.
DTD-Definition:

```
<!ELEMENT timer EMPTY>
<!ATTLIST timer
          name NMTOKEN #IMPLIED
          value %vdata; #REQUIRED
          %coreattrs; >
```

4.2.23 Die Klasse Img

Diese Klasse repräsentiert die Einbindung von WBMP - Grafiken in eine WML - Seite. Die Attribute steuern dabei die Details der Darstellung.
Dies ist ein Tag ohne weitere Kinder und somit direkt von „EmptyTag“ abgeleitet.
DTD - Definition:

```
<!ELEMENT img EMPTY>
<!ATTLIST img
          alt %vdata; #REQUIRED
          src %HREF; #REQUIRED
          localsrc %vdata; #IMPLIED
          vspace %length; "0"
          hspace %length; "0"
          align %IAlign; "bottom"
          height %length; #IMPLIED
          width %length; #IMPLIED
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.24 Die Klasse Anchor

Diese Klasse repräsentiert eine weitergehende Bindung von Aktionen an einen Anker. Somit können Kurzwahltasten oder komplexere Aktionen mit einen Anker verbunden werden.
DTD - Definition:

```
<!ELEMENT anchor ( #PCDATA | br | img | go | prev | refresh )*>
<!ATTLIST anchor
          title %vdata; #IMPLIED
          accesskey %vdata; #IMPLIED
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.25 Die Klasse A

Diese Klasse repräsentiert einen Quellanker für Links innerhalb eines Kartenstapels (Deck) oder auf eine externe Ressource. Er kann aus normalen Text mit Zeilenumbrüchen oder Bildern bestehen.
DTD-Definition:

```
<!ELEMENT a ( #PCDATA | br | img )*>
<!ATTLIST a
          href %HREF; #REQUIRED
          title %vdata; #IMPLIED
          accesskey %vdata; #IMPLIED
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.26 Die Klasse Table

Diese Klasse repräsentiert einen logischen Rahmen um eine Tabelle mit einer beliebigen Anzahl von Tabellenzeilen.
DTD-Definition:

```
<!ELEMENT table (tr)+>
<!ATTLIST table
          title %vdata; #IMPLIED
          align CDATA #IMPLIED
          columns %number; #REQUIRED
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.27 Die Klasse Tr

Diese Klasse repräsentiert eine Tabellenzeile.
DTD-Definition:

```
<!ELEMENT tr (td)+>
<!ATTLIST tr
          %coreattrs; >
```

4.2.28 Die Klasse Td

Diese Klasse repräsentiert eine Zelle in einer Tabellenzeile. Sie beinhaltet die eigentlichen Daten in der Tabelle.
DTD-Definition:

```
<!ELEMENT td ( %text; | %layout; | img | anchor |a )*>
<!ATTLIST td
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.29 Die Klasse Em

Diese Klasse repräsentiert einen hervorgehobenen Textabschnitt.
DTD-Definition:

```
<!ELEMENT em (%flow;)*>
<!ATTLIST em
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.30 Die Klasse Strong

Diese Klasse repräsentiert einen Textabschnitt, welcher noch mehr hervorgehoben ist als mit dem Tag .
DTD - Definition:

```
<!ELEMENT strong (%flow;)*>
<!ATTLIST strong
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.31 Die Klasse B

Diese Klasse repräsentiert eine Schriftart, welcher mit der aktuellen identisch ist, jedoch durch Fettschrift gekennzeichnet ist.
DTD - Definition:

```
<!ELEMENT b (%flow;)*>
<!ATTLIST b
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.32 Die Klasse I

Diese Klasse repräsentiert eine Schriftart, welche mit der aktuellen identisch ist, jedoch kursiv (schräg) dargestellt ist.
DTD - Definition:

```
<!ELEMENT i (%flow;)*>
<!ATTLIST i
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.33 Die Klasse U

Diese Klasse repräsentiert eine Schriftart, welcher mit der aktuellen identisch ist, jedoch unterstrichen ist.
DTD - Definition:

```
<!ELEMENT u (%flow;)*>
<!ATTLIST u
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.34 Die Klasse Big

Diese Klasse repräsentiert eine Schriftart, welche grösser als die Standardschriftart ist.
DTD - Definition:

```
<!ELEMENT big (%flow;)*>
<!ATTLIST big
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.35 Die Klasse Small

Diese Klasse repräsentiert eine Schriftart, welche kleiner als die Standardschriftart ist.
DTD - Definition:

```
<!ELEMENT small (%flow;)*>
<!ATTLIST small
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.36 Die Klasse P

Diese Klasse repräsentiert eine Absatzformatierung innerhalb einer Karte.
DTD - Definition:

```
<!ELEMENT p (%fields; | do)*>
<!ATTLIST p
          align %TAlign; "left"
          mode %WrapMode; #IMPLIED
          xml:lang NMTOKEN #IMPLIED
          %coreattrs; >
```

4.2.37 Die Klasse Br

Diese Klasse repräsentiert einen Zeilenumbruch und trägt nur Standardattribute.
DTD - Definition:

```
<!ELEMENT br EMPTY>
<!ATTLIST br
          %coreattrs; >
```

4.2.38 Die Klasse Pre

Diese Klasse repräsentiert vorformatierten Text, der ohne Interpretation darin vorhandener Tags dargestellt wird. Somit lässt sich auch Text mit mehreren Leerzeichen korrekt ausgeben.
DTD - Definition:

```
<!ELEMENT pre (#PCDATA | a | anchor | do | u | br | i | b
               | em | strong | input | select )*>
<!ATTLIST pre
          xml:space CDATA #FIXED "preserve"
          %coreattrs; >
```

4.2.39 Die Klasse CData

Diese Klasse repräsentiert Text, welcher vom Microbrowser direkt auf dem Bildschirm ausgegeben werden kann.
Mit dieser Klasse ist es möglich von der WML-Sprache reservierte Zeichenketten auszugeben.

4.2.40 Die Klasse PCData

Diese Klasse repräsentiert Text, welcher vor Ausgabe im Microbrowser nach Tag, Kommentare oder weiteren Markups durchsucht und dementsprechend interpretiert wird.

4.3 Attribute

In Abbildung **??** wird der Aufbau der Attributstrukturen des Systems beschrieben.

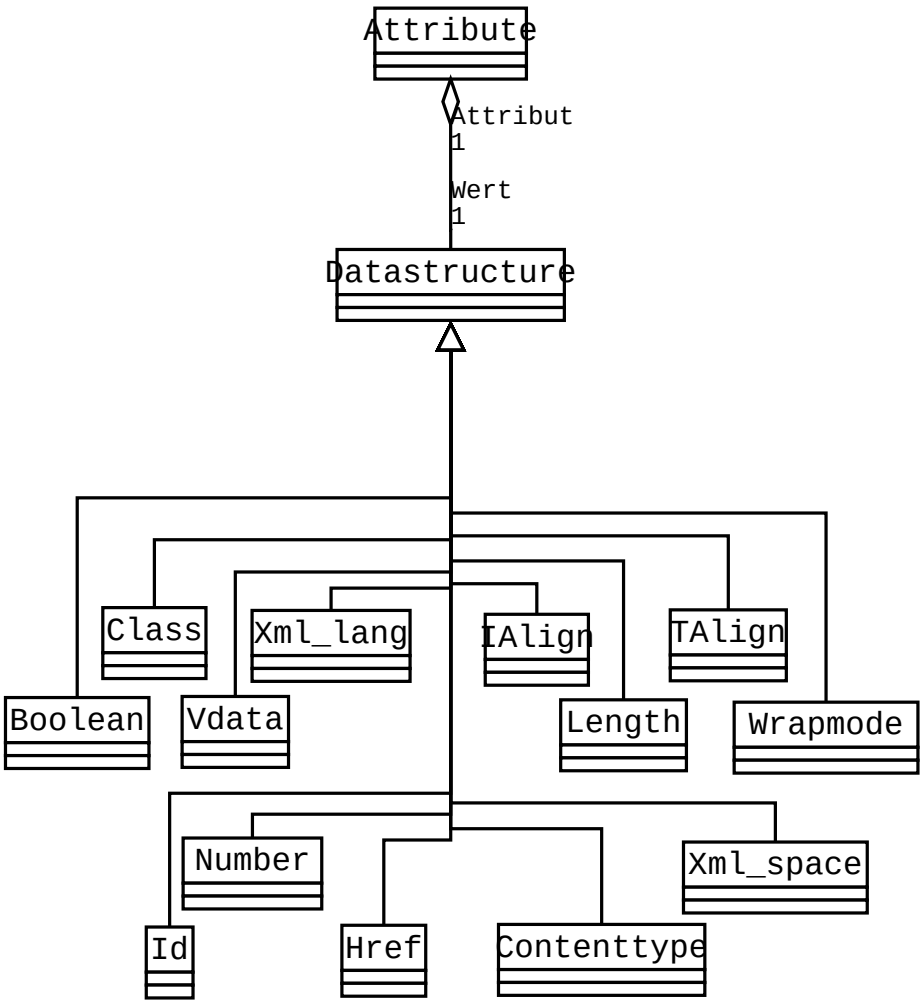


Abbildung 15: UML-Diagramm Attribute

Attribute
+observer: Observer* +att_name: std::string +value: Datastructure*
+Attribute(observer:Observer*,att_name:std::string,type:Datastructure::attrtype): +~Attribute(): +dump(): void

Abbildung 16: UML-Diagramm der Klasse Attribute

4.3.1 Die Klasse Attribute

Die Klasse Attribute stellt ein Attribut eines einzelnen WML - Tags dar. Daraus folgt, dass ein WML Tag in der Regel mehrere Attribute haben kann.
Jedes Attribut besitzt neben dem Pointer zum Observer seinen Attributnamen als Zeichenkette (std::string attr_name) und seinen Wert, welcher durch eine Instanz von Datastructure repräsentiert wird. Somit hat jede Attributsklasse genau eine von Datastructure abgeleitete Klasse.
Die Funktion dump, welche zur Ausgabe des Attributes dient, benutzt neben seinem Namen die dump - Funktion ihres Wertes, repräsentiert durch den Datastructure Typ.

4.3.2 Die Klasse Datastructure

Die Klasse Datastructure ist die abstrakte Oberklasse aller Datentypen, welche in Attributen von WML - Tags vorkommen.
Die Klasse besitzt einen enumerierten Typ, welcher den Attributen übergeben wird, so dass Attribute eine der von Datastructure abgeleiteten Unterklassen erzeugen können.
Datastructure bietet eine zu überladene Funktion dump an, welche eine Zeichenkette mit einer möglichen Ausgabe der Datenstruktur zurückliefert.

4.3.3 Die Klasse Class

Die Klasse Class repräsentiert die Ausgabe eines Klassennamens als Attribut. Sie ist von Datastructure abgeleitet und bietet in der dump - Funktion die Rückgabe eines gültigen Klassennamens.

4.3.4 Die Klasse XML_lang

Die Klasse XML_lang repräsentiert die Ausgabe einer gültigen Sprachkodierung im WML - Stil als Attribut. Sie ist von Datastructure abgeleitet und bietet in der dump - Funktion die Rückgabe einer gültigen Sprachkodierung aus den Werten der RFC.

4.3.5 Die Klasse XML_space

Die Klasse XML_space repräsentiert die Auswahl der Leerzeichenbehandlung. Sie ist von Datastructure abgeleitet und bietet in der dump - Funktion die Rückgabe von „preserve“ oder „default“.

4.3.6 Die Klasse Boolean

Die Klasse Boolean repräsentiert den boole’schen Datentyp. Sie ist von Datastructure abgeleitet und bietet in der dump - Funktion die Rückgabe von „true“ oder „false“.

4.3.7 Die Klasse Vdata

Die Klasse Vdata repräsentiert die Ausgabe eines Variablennamens als Attribut. Sie ist von Datastructure abgeleitet und bietet in der dump - Funktion die Rückgabe eines gültigen Variablennamens nach einer typischen Grammatik, wie sie in allen Programmiersprachen vorhanden ist.

4.3.8 Die Klasse Id

Die Klasse Id repräsentiert die Ausgabe eines repräsentativen Bezeichners als Attribut. Sie ist von Datastructure abgeleitet und bietet in der dump - Funktion die Rückgabe eines eindeutigen repräsentativen Bezeichners. Hierzu wird die Funktion get_GUID des Observers benutzt.

4.3.9 Die Klasse Number

Die Klasse Number repräsentiert die Ausgabe einer Zahl als Attribut. Sie ist von Datastructure abgeleitet und bietet in der dump - Funktion die Rückgabe einer zufällig erzeugten Zahl.

4.3.10 Die Klasse Href

Die Klasse Href repräsentiert die Ausgabe einer Hyperlinkadresse als Attribut. Sie ist von Datastructure abgeleitet und bietet in der dump - Funktion die Rückgabe einer konform erzeugten Hyperlinkadresse.

4.3.11 Die Klasse Contenttype

Die Klasse Contenttype repräsentiert die Ausgabe eines Medientyps. Diese darf Variablendaten (vdata) enthalten.

4.3.12 Die Klasse Wrapmode

Die Klasse Wrapmode repräsentiert die Auswahl eines Zeilenumbruchsmodus. Sie ist von Datastructure abgeleitet und bietet in der dump - Funktion die Rückgabe von „wrap“ oder „nowrap“.

4.3.13 Die Klasse Length

Die Klasse Length repräsentiert die Ausgabe einer Längenangabe als Attribut. Sie ist von Datastructure abgeleitet und bietet in der dump - Funktion die Rückgabe einer zufällig erzeugten Zahl als Längenangabe.

4.3.14 Die Klasse IAlign

Die Klasse IAlign repräsentiert die Ausgabe einer vertikalen Anordnung als Attribut. Sie ist von Datastructure abgeleitet und bietet in der dump - Funktion die Rückgabe von „top“, „middle“ oder „bottom“.

4.3.15 Die Klasse TAlign

Die Klasse TAlign repräsentiert die Ausgabe einer vertikalen Anordnung als Attribut. Sie ist von Datastructure abgeleitet und bietet in der dump - Funktion die Rückgabe von „left“, „right“ oder „center“.

5 Klassenbeziehungen

Das Zentrum des Softwaresystems ist der Observer. Er erstellt auf Anfrage von Tagelementen weitere Tags und erstellt auch Attributwerte mit ihrem entsprechenden Datentyp für die Attribute. Daher besitzen alle Klasseninstanzen - abgesehen von den Konfigurationsklassen - eine Referenz auf das Observerobjekt. Der Observer kennt dazu noch die Konfiguration, welche somit auch allen Klassen zur Verfügung steht, welche den Observer kennen.
Die Zufallszahlenerzeugung durch die Klasse Rnd wird auch vom Observer übernommen.

Die Konfiguration besitzt ein Konfigurationsobjekt, welches mit Hilfe von INIFileParser und FileIO erstellt wurde.

Zu Tags gehören mehrere Instanzen von Attributen, welche wiederum das Objekt Datastructure nutzen. Eine solche Instanz wird dann durch einen abgeleiteten Datentyp ausgedrückt.

In der folgenden Darstellung stehen die „Wolken“ stellvertretend für die vielen von Tag abgeleiteten Klassen bzw. für die Datentypen, welche von Datastructure abgeleitet wurden.

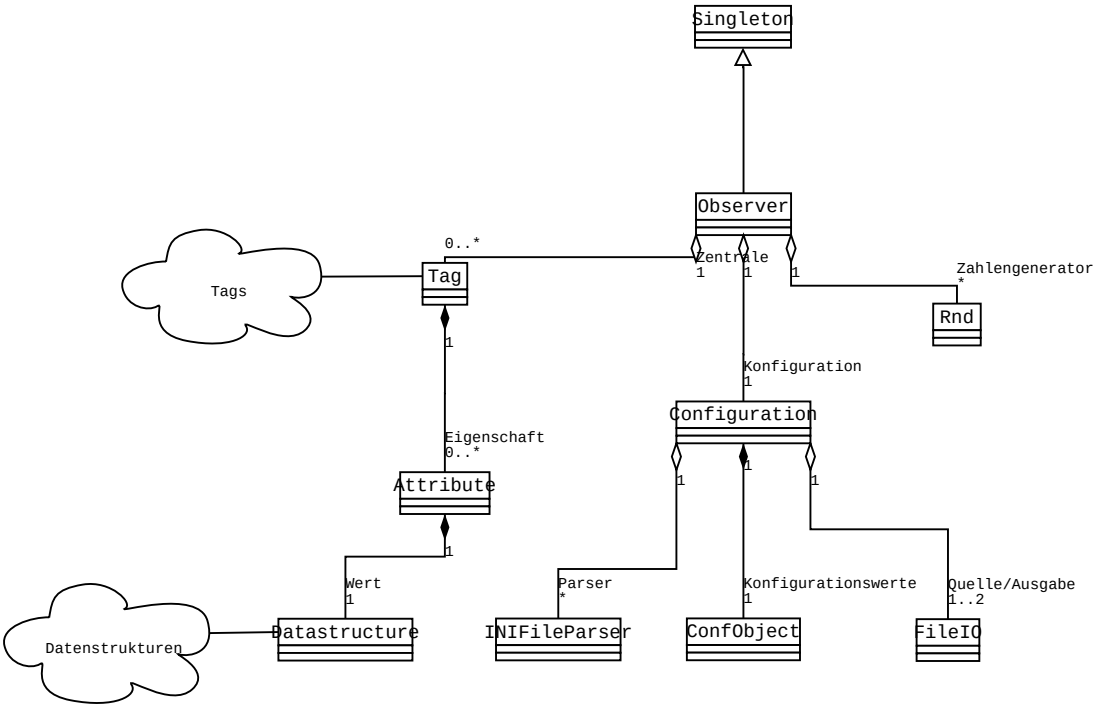


Abbildung 17: UML-Diagramm gesamt

6 Datenfluss

Folgende Sequenz- und Kolaborationsdiagramme visualisieren den Datenfluss an wichtigen Stellen des Softwaresystems:
Abbildung **??** zeigt die Instanziierung des Hauptprogramms mit der Erstellung der Kontrollstrukturen als Sequenzdiagramm. Abbildung **??** stellt das Kolaborationsdiagramm für diesen Vorgang dar.
Abbildung **??** zeigt im Detail die Erstellung eines neuen (Unter-)Tags mit allen erfolgten Funktionsaufrufen, die hierzu notwendig sind, als Sequenzdiagramm, Abbildung **??** zeigt hierfür das Kolaborationsdiagramm.
Die Abbildungen **??** und **??** visualisieren die Ausgabe der Tags während des Programmablaufs als grobe Übersicht.

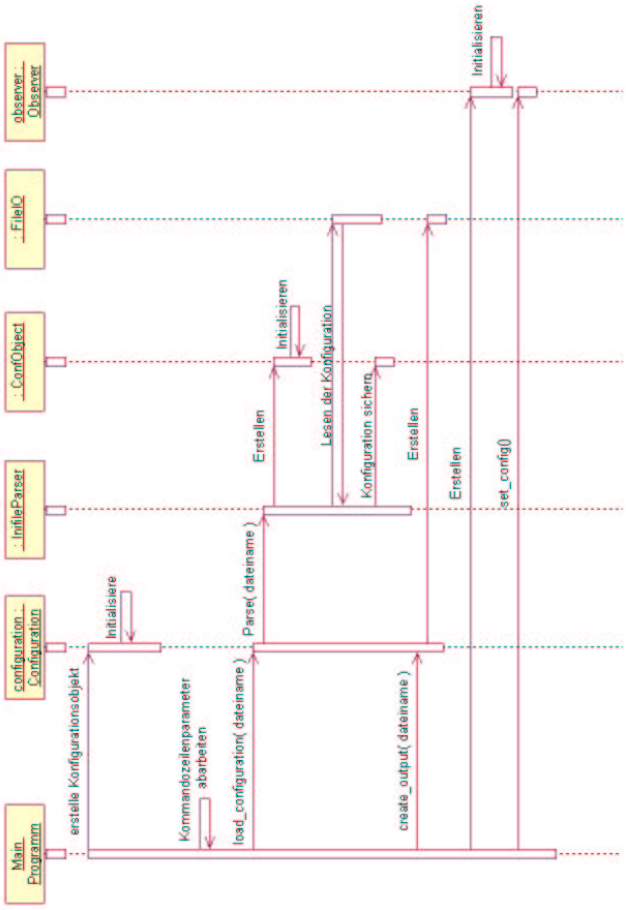


Abbildung 18: Instanziierung des Hauptprogramms

In der Programminstanziierung wird zuerst eine Instanz der Konfiguration erstellt bevor die Kommandozeilenparameter abgearbeitet werden. Aufgrund dieser Angaben wird dann eine Konfiguration aus einer Datei geladen, was durch die Konfiguration auf den IniFileParser ausgelöst wird. Dieser erstellt das in der Konfiguration gespeicherte

Konfigurationsobjekt, welches benutzt wird, um die mittels FileIO gelesen Daten zu speichern.
Nach diesen Initialisierungen wird im Hauptprogramm über die Konfiguration eine Ausgabe über FileIO erzeugt, bevor der Observer instanziiert wird und die Konfiguration übergeben bekommt.
Daraufhin wird das erste Tag (<wml>) erstellt und mittels dump() angestossen. Den weiteren Verlauf entnehmen sie bitte Abbildung **??**.

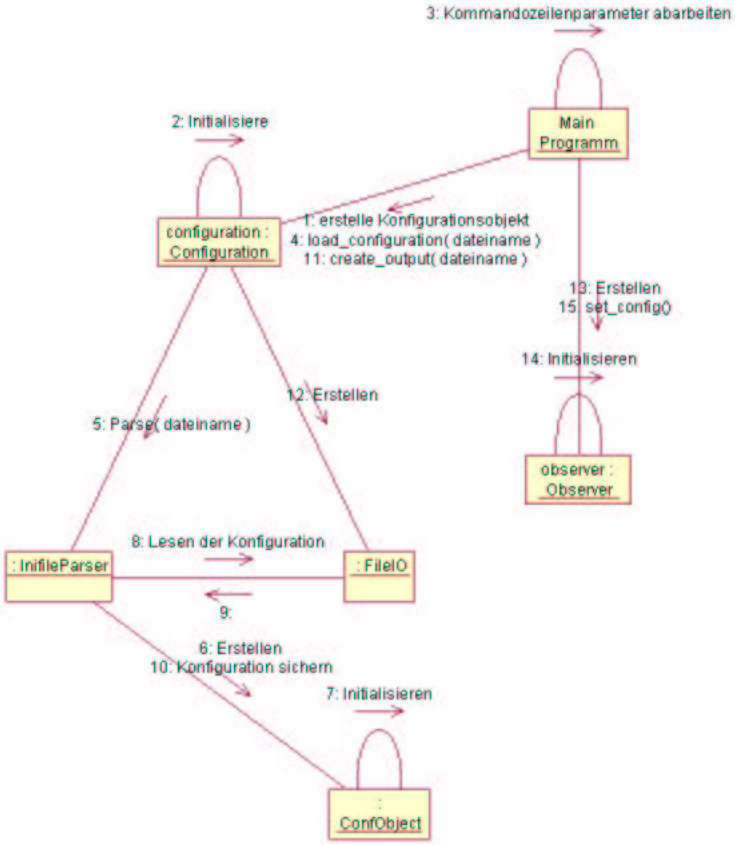


Abbildung 19: Kolaborationsdiagramm Instanziierung des Hauptprogramms

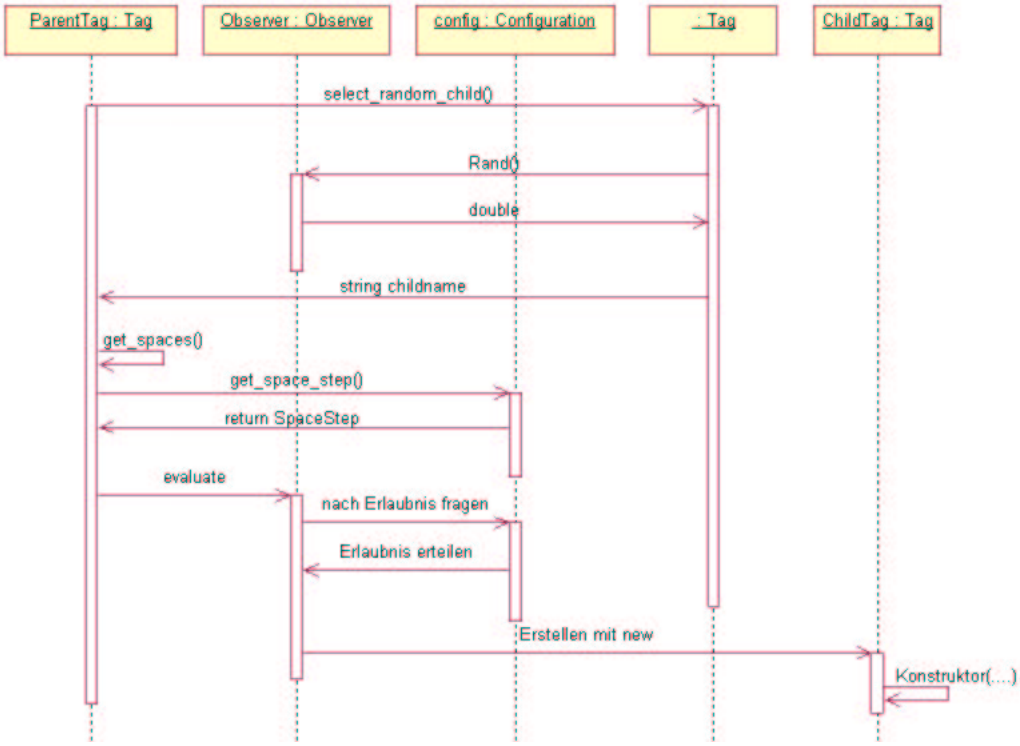


Abbildung 20: Detaillierte Erstellung eines Kind-Tags

In der Abarbeitung der Kinder können Tags ihre KinderTags erstellen. Um dies zu veranlassen, wird über die abstrakte Basisklasse Tag eine Zeichenkette, welche den Attributsnamen repräsentiert, ausgewählt. Dafür benutzt Tag den Zufallszahlengenerator des Observer. Nach der Bestimmung der Anzahl der Leerstellen vor dem Tagnamen zur Formatierung fordert das Tag vom Observer die Erstellung des Kindtags an. Dazu wird mittels „evaluate“ dem Observer die Zeichenkette übergeben, der daraufhin in der Konfiguration nachsieht, ob dieses Tag erstellt werden darf. Wenn kein Verbot über die Konfiguration vorliegt wird er das Tag erstellen, welches daraufhin instanziiert wird.

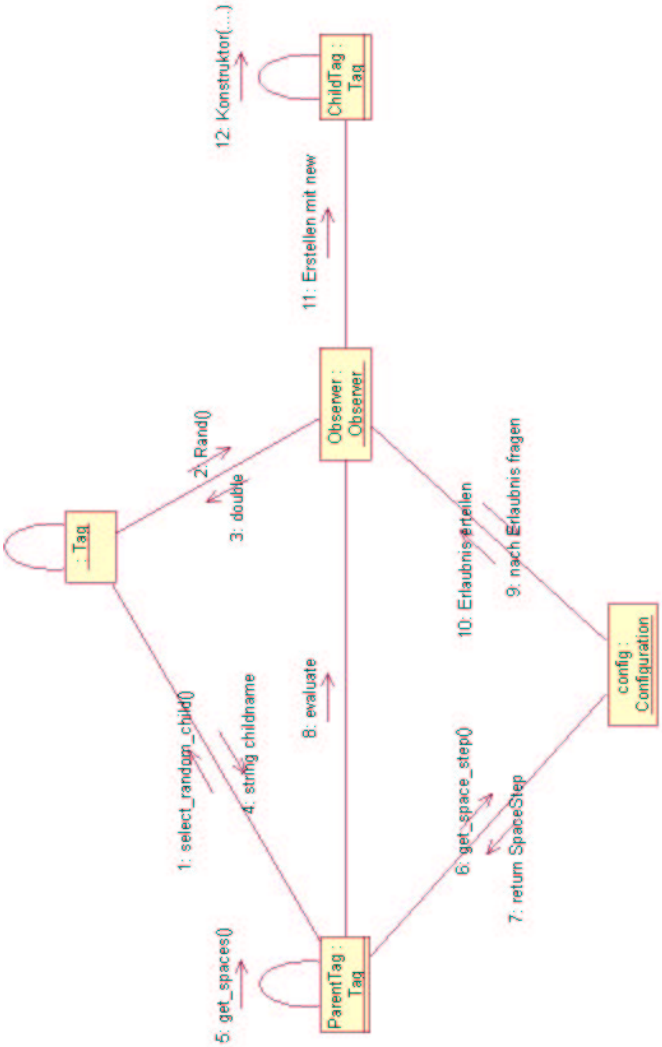


Abbildung 21: Kolaborationsdiagramm detaillierte Erstellung eines Kind-Tags

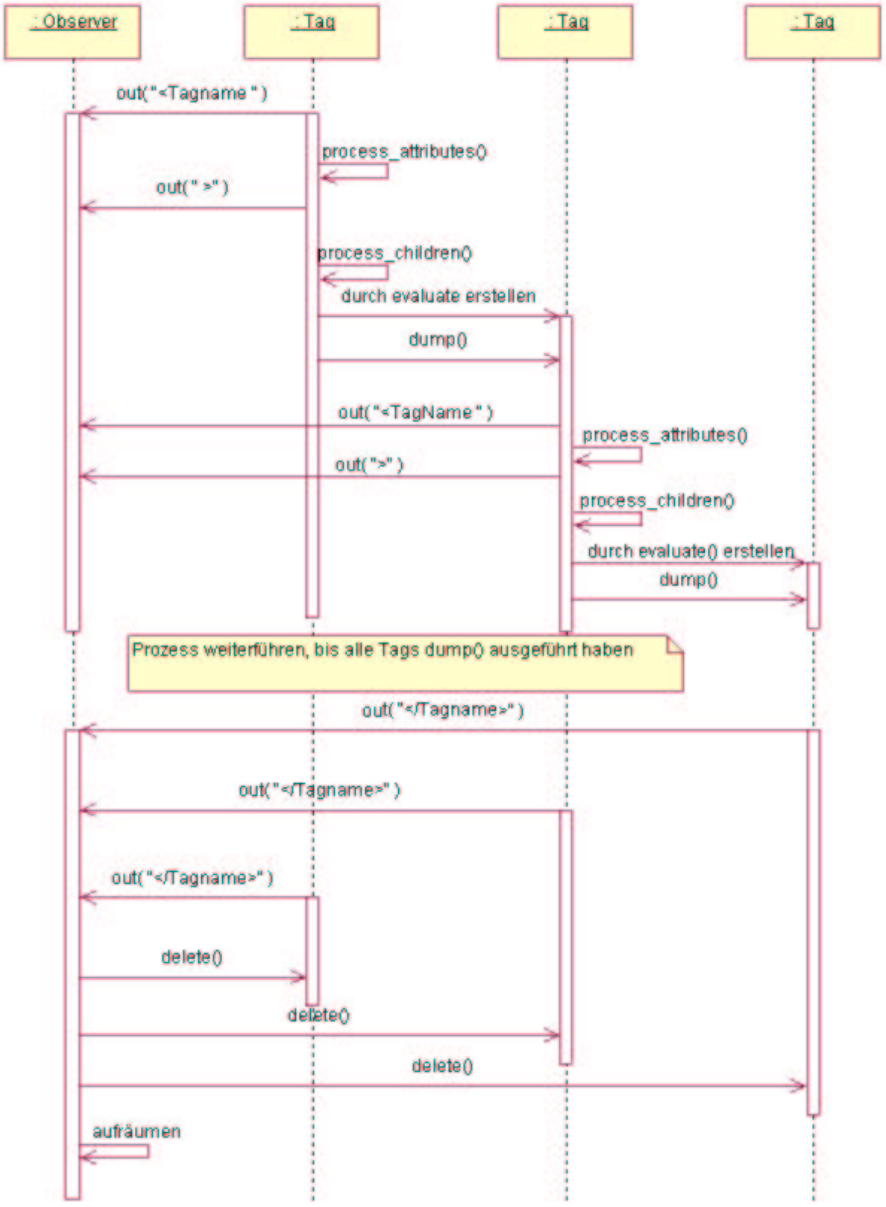


Abbildung 22: Dokumentausgabe

Die Dokumentausgabe erfolgt rekursiv durch die Tags an den Observer. Dieser ist mittles „out“ für eine passende Ausgabe in Datei, Webserver oder Konsole verantwortlich.
Ein Tag gibt zuerst seinen Tagnamen aus und arbeitet seine Attribute ab. Nach Ausgabe der schliessenden spitzen Klammer des Anfangstags werden die Kinder abgearbeitet, welche mittels evaluate erstellt und mittels dump() ausgegeben werden. So werden alle Tags ausgegeben bis die Abbruchkriterien erfüllt sind (keine Tags mehr möglich oder

Dateigrössen erreicht).
Wenn die Rekursion zurückkehrt wird das abschliessende Tag jeder Instanz noch ausgegeben bevor der Observer alle Tags zerstört.

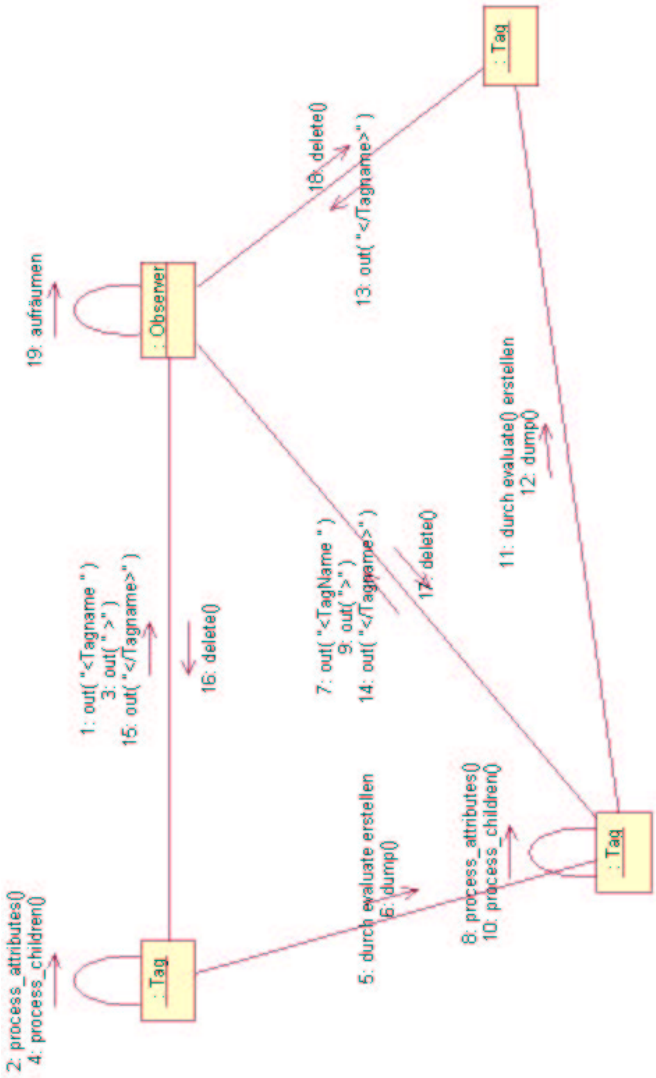


Abbildung 23: Kolaborationsdiagramm Dokumentausgabe

Desweiteren existiert ein Diagramm der DTD, welches für die Darstellung innerhalb dieses Dokumentes zu groß ist (> 2500x2500 Pixel). Dieses Dokument ist im Gruppen-CVS und auf der Homepage zu finden.

7 Informationen zu Prototyp 1

Zu diesem Entwurf existiert ein Prototyp, welcher als Machbarkeitsbeweis dieses Designs dienen soll.
Er umfasst schon annähernd alle Klassen und implementiert viele mit vorgeschriebener Funktionalität. So sind alle Tags, die generiert werden müssen enthalten und können generiert und ausgegeben werden. Hierbei werden jedoch bei den Tag - Attributen meist nur Standardwerte ausgegeben, da nicht alle Attributdatentypen komplett implementiert sind.
Konfigurationsdateien können auch gelesen und geparst werden, werden jedoch noch nicht während der Erzeugung des Dokumentes berücksichtigt. Textausgaben mittels CData und PCData existieren noch nicht.
Ein Perl - Skript existiert, so dass eine Version für den folgenden Online - Massentest im Emulator oder Endgerät bereits vorhanden ist.
Die Onlineversion befindet sich auf:
http://alan.cs.uni-sb.de/wml/cgi-bin/wmlgen.pl

Die Quellcodes befinden sich im Gruppen - CVS im Modul „code“ im Unterpfad „prototyp“. Zur Programmerstellung liegt ein Makefile vor.