Gerrit Meixner und Robbie Schäfer

Modellbasierte Entwicklung von Benutzungsschnittstellen mit UIML

Model-Based Development of User Interfaces with UIML

UIML_MBUID_Aufgabenmodell_Dialogmodell_Präsentationsmodell

Zusammenfassung. Dieser Beitrag beschäftigt sich mit der modellbasierten Entwicklung von Benutzungsschnittstellen auf Basis der User Interface Markup Language (UIML). Aufbauend auf einer Einleitung wird in das Gebiet der modellbasierten Entwicklung von Benutzungsschnittstellen eingeführt. Insbesondere werden die Kernmodelle charakterisiert, die in eine kohärente modellbasierte Architektur eingebettet sein sollten. Im Anschluss werden die wesentlichen Merkmale und Neuerungen von UIML erläutert, aktuelle Werkzeuge zur Bearbeitung von UIML aufgezählt und die Sprache UIML in Zusammenhang mit einem modellbasierten Entwicklungsprozess bewertet.

Summary. This article provides information about the possibilities of model-based development of user interfaces with the User Interface Markup Language (UIML). Next to a brief introduction in the field of model-based user interface development – together with its most relevant models – UIML is introduced by its main concepts and latest developments. A specific focus is set on the question whether UIML is able to support a complete modeling process from a task model to a final user interface. The practical value of UIML is assessed towards the availability of actual development tools.

1. Einleitung

Schon seit langem wird versucht die Kommunikation zwischen Menschen und Computern zu vereinfachen und zu optimieren. Heute erfolgt die Interaktion in den meisten Fällen noch über grafische Benutzungsschnittstellen (GUI). Die Interaktion des Menschen mit einem Computer über multimodale Benutzungsschnittstellen (z. B. visuell, akustisch, haptisch) gewinnt allerdings zunehmend an Bedeutung bei der Bedienung (Zühlke 2004).

Auch die stetig wachsende Diversifizierung der Plattformen (PC, Smartphone, PDA, etc.) macht es erforderlich, dass Benutzungsschnittstellen für eine Vielzahl von Zielplattformen konsistent entwickelt werden, um deren intuitive Benutzung und dadurch implizit die Zufriedenheit der Nutzer gewährleisten zu können (Luyten 2004). Um diesen Anforderungen gerecht zu werden, spielen Aspekte wie Wiederverwendbarkeit, Flexibilität und Plattformunabhängigkeit eine ganz entscheidende Rolle bei der Entwicklung aktueller Benutzungsschnittstellen (Calvary et al. 2003). Da der immer wiederkehrende Entwicklungsaufwand für Einzellösungen für eine spezifische Plattform, Modalität oder gar einen besonderen Nutzungskontext zu hoch ist, bietet sich ein modellbasierter Ansatz für die Entwicklung von Benutzungsschnittstellen an (Puerta 1997).

Entwicklungen im Bereich der modellbasierten Entwicklung von Software beispielsweise. durch den Einsatz der Unified Modeling Language (UML) und der Model Driven Architecture (MDA) zeigen, wie anhand von Modellen und darauf aufbauenden Generatoren lauffähige Software erstellt werden kann. Das Ziel der MDA ist neben der Verbesserung der Portierbarkeit und Wiederverwendung von Modellen auch die Reduktion der Kosten bei der Entwicklung von Software (Gruhn, Pieper und Röttgers 2006).

Dieses modellbasierte Entwicklungskonzept kann auf die Entwicklung von Benutzungsschnittstellen übertragen werden. Modellbasiert bedeutet in diesem Kontext, dass eine abstrakte Darstellung von Benutzungsschnittstellen verwendet wird, welche dann (semi-)automatisch in eine konkrete Repräsentation der jeweiligen Plattform umgesetzt wird. Der Entwicklungsaufwand für unterschiedliche Plattformen wird dabei deutlich reduziert. Neben dieser abstrakten Beschreibung von Benutzungsschnittstellen, die losgelöst von jeglicher Repräsentation Benutzungsschnittstellen definiert, ist eine strukturierte Entwicklung von Benutzungsschnittstellen möglich, da verschiedene Modelle mit unterschiedlichen Abstraktionsstufen zur Entwicklung vorliegen und somit gezielt in der jeweiligen Phase eines modellbasierten Entwicklungsprozesses verwendet werden können (Mori, Paternò und Santoro 2004).

Ein weiterer wichtiger Punkt der modellbasierten Entwicklung liegt in der strikten Trennung zwischen der Darstel-

DOI 10.1524/icom.2009.0010

lung der Benutzungsschnittstelle und der Applikationslogik, wie zum Beispiel im Model-View-Controller-Paradigma (MVC) realisiert. Eine Benutzungsschnittstelle (View) kann somit von einem Designer entwickelt und von einem Informatiker mit Funktionen (Model, Controller) angereichert werden. Dadurch können sich Designer und Informatiker auf ihre jeweiligen Spezialgebiete beschränken.

Viele Sprachen zur Beschreibung von Benutzungsschnittstellen bzw. deren Modelle basieren auf XML – der eXtensible Markup Language (Souchon und Vanderdonckt 2003). Die Vorteile des Einsatzes von XML-basierten Sprachen liegen insbesondere in der Plattformunabhängigkeit, der klaren hierarchischen Struktur, der Lesbarkeit und nicht zuletzt der einfachen Transformierbarkeit, was dadurch auch eine inhärente Modelltransformation ermöglicht.

In diesem Artikel wird die User Interface Markup Language (UIML) beschrieben und wie sie für die modellbasierte Entwicklung von Benutzungsschnittstellen eingesetzt werden kann. Im folgenden Abschnitt werden die unterschiedlichen Modelle charakterisiert, welche für die Entwicklung von Benutzungsschnittstellen von Bedeutung sind. Anschließend wird UIML im Detail vorgestellt und eine Bewertung bezüglich der Verwendbarkeit und Werkzeugunterstützung für die modellbasierte Entwicklung von Benutzungsschnittstellen vorgenommen.

2. Modellbasierte Entwicklung von Benutzungsschnittstellen

Ein Modell hat das Ziel, Zusammenhänge und Zusammenwirken in der Realität durchschaubar und Entwicklungen bzw. Verhaltensweisen vorhersehbar zu machen. Insbesondere werden dabei Abstraktionen oder Vereinfachungen der Realität ohne Übernahme der Komplexität durchgeführt. Zur Komplexitätsbeherrschung bei der Entwicklung von Benutzungsschnittstellen wird der modellbasierte Ansatz des Model-Based User Interface Development (MBUID) verwendet. MBUID stellt einen allgemeinen Rahmen zur Verfügung, um verschiedene Aspekte einer Benutzungsschnittstelle modellieren zu können. Das Design einer Benutzungsschnittstelle wird im Wesentlichen durch drei verschiedene Kernmodelle definiert: Aufgaben-, Dialog- und Präsentationsmodell. Diese Modelle stellen eine Konkretisierung des MVC-Paradigmas dar (Luyten 2004). Um die unterschiedlichen Nutzungskontexte bei der Modellierung zu berücksichtigen, hat sich in den letzten Jahren das Cameleon Reference Framework (Calvary et al. 2003) etabliert. Vereinfacht gesehen, basiert dieses Framework auf vier verschiedenen Schichten (vgl. Bild 1), wobei die oberen drei Schichten (Aufgaben und Konzepte, Abstrakte Benutzungsschnittstelle, Konkrete Benutzungsschnittstelle) den o.g. Modellen entsprechen. Die unterste Schicht des Frameworks ist die finale Benutzungsschnittstelle, die letztlich für ein bestimmtes Gerät erzeugt wurde und meist in Quellcode der Zielplattform ausgedrückt wird. Neben der vertikalen Ebene, die den Weg von abstrakten zu immer konkreteren Modellen beschreibt, gibt es auch eine horizontale Ebene, die abhängig vom gegenwärtigen Kontext ein Modell anpassen kann. So könnte sich beispielsweise für ein Aufgabenmodell die Reihenfolge der Unteraufgaben ändern, wenn der Nutzer sich an unterschiedlichen Orten befindet.

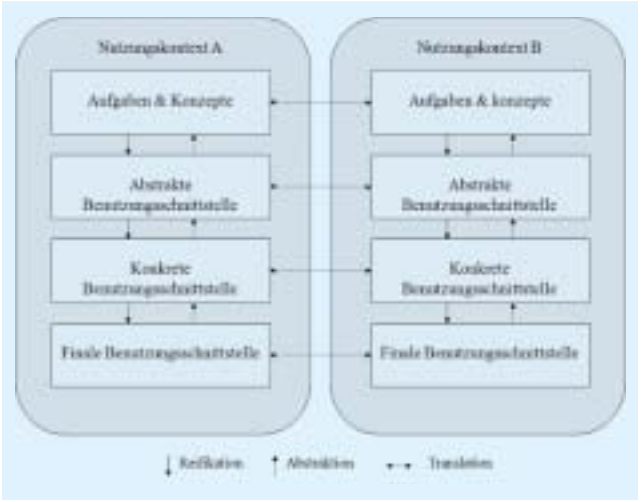


Bild 1: Vereinfachtes Cameleon Reference Framework nach (Limbourg et al. 2004)

Im Folgenden werden die drei Kernmodelle (Luyten 2004) beschrieben.

2.1 Das Aufgabenmodell

Das Aufgabenmodell ist eine Struktur, die die Aufgaben, Tätigkeiten und Handlungen einzelner Nutzer der Benutzungsschnittstelle – in der Regel hierarchisch – beschreibt. Dabei gliedert das Aufgabenmodell einzelne Aufgaben in weitere Unteraufgaben, z. B. lässt sich der Vorgang „Kaffee kochen" an einer handelsüblichen Kaffeemaschine in die Unteraufgaben „Wasser einfüllen", „Filter einsetzen", „Kaffee einfüllen", „Maschine anschalten", etc. gliedern. Desweiteren wird die Abfolge der Aufgaben zeitlich mittels Temporaloperatoren in Relation gesetzt (Meixner, Görlich und Schäfer 2008). Temporaloperatoren sind zur Beschreibung notwendig, da das zeitliche Verhalten der Aufgabenausführung sonst nicht klar definiert werden kann. Am Beispiel der Aufgabe „Kaffee kochen" sind Temporaloperatoren zur Spezifizierung notwendig, damit nicht erst Kaffee eingefüllt wird, bevor der Filter eingesetzt wurde. Notationen, um Aufgaben strukturiert modellieren, darstellen und evaluieren zu können, sind u.a. die „Useware Markup Language" (useML) (Reuther 2003) oder „Concurrent Task Trees" (CTT) (Paternò 1999). Das Aufgabenmodell wird normalerweise im Rahmen entsprechender Methoden der Aufgabenanalyse (Interviews, Beobachtungen, Strukturlegetechniken, etc.) entwickelt. Die Integration eines Aufgabenmodells in einen modellbasierten Entwicklungsprozess, optimiert die aufgaben- und nutzerzentrierte Entwicklung der Benutzungsschnittstelle (Paris, Lu und Vander Linden 2003). Die Anwendung von Transformationsprozessen ermöglicht die Generierung des Dialog- und Präsentationsmodell auf Basis des Aufgabenmodells.

2.2 Das Dialogmodell

Das Dialogmodell beschreibt die Interaktionen des Nutzers mit der Benutzungsschnittstelle und deren Auswirkung ohne eine konkrete Technik bzw. „Look&Feel"

zu berücksichtigen. Das Dialogmodell verbindet das Aufgabenmodell mit dem Präsentationsmodell. Es stellt den dynamischen Teil bei der Ausführung von Aufgaben des Nutzers (Aufgabenmodell) dar und ist daher eng mit der (visuellen) Darstellung des Präsentationsmodells verknüpft. Für die Modellierung von Dialogen gibt es eine Vielzahl verschiedener Notationen, z. B. Endliche Automaten, Petri-Netze oder die Backus-Naur-Form (BNF). Meist wird das Dialogmodell in Sprachen zur Beschreibung der abstrakten und konkreten Benutzungsschnittstelle (vgl. Bild. 1) integriert.

2.3 Das Präsentationsmodell

Die eigentliche Abbildung der Benutzungsschnittstelle wird durch das Präsentationsmodell dargestellt. Die Art und Weise, wie Benutzungsschnittstellen repräsentiert werden, hängt von den Bedingungen des Ausgabemediums ab (z. B. Displaygröße, Farbdarstellung). Dabei kann die Interaktion bspw. über eine grafische Oberfläche eines PCs oder durch Sprachinteraktion mit einem Smartphone stattfinden. Das Präsentationsmodell setzt sich aus dem abstrakten und konkreten Präsentationsmodell zusammen. Typischerweise bestehen nach (Vanderdonckt und Bodart 1993) Benutzungsschnittstellen aus einer Menge abstrakter Interaktionsobjekte (AIO), die im nächsten Schritt auf korrespondierende konkrete Interaktionsobjekte (CIO) abgebildet werden. Die Gesamtheit der AIO's ergeben das abstrakte und die Gesamtheit der CIO's das konkrete Präsentationsmodell. Ein CIO ist demnach die konkrete Darstellung eines Objektes. Ein AIO „Helligkeitsregler für Bildschirme" kann sowohl als CIO „Schieberegler", aber auch als CIO „Zähler" dargestellt werden. Des Weiteren könnte ein AIO „Helligkeitsregler für Bildschirme" alternativ mit einer Spracheingabe bzw. -ausgabe realisiert werden. Beispiele für XML-basierte Sprachen zur Beschreibung von Präsentationsmodellen sind u.a. TeresaXML (Mori, Paternò und Santoro 2004), XIML, XAML, XUL oder UIML.

3. Die User Interface Markup Language (UIML)

Eine User Interface Description Language (UIDL) ist i.d.R. eine deklarative Beschreibung einer Benutzungsschnittstelle. Dabei wird die Benutzungsschnittstelle unabhängig von einer Zielsprache spezifiziert (Souchon und Vanderdonckt 2003). Die User Interface Markup Language (UIML), als Beispiel einer UIDL, ist eine Meta-Sprache zur plattformunabhängigen Beschreibung von Benutzungsschnittstellen. Mittels der in UIML definierten Vokabularen (analog zu XML-Schema / DTD) können abstrakt definierte Präsentationsmodelle auf beliebige konkrete Präsentationsmodelle (z. B. eine Benutzungsschnittstelle in Java mit der Grafikbibliothek bzw. dem Widget-Toolkit „Swing") oder auf andere UIDLs wie bspw. XAML oder XUL abgebildet werden.

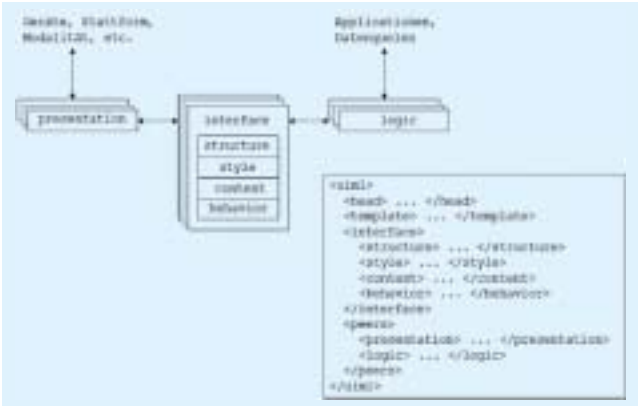
UIML wurde an der "Virginia Polytechnic Institute and State University" entwickelt. Später wurden die Arbeiten intensiv durch das Unternehmen Harmonia fortgeführt. Die erste Spezifikation der Sprache stammt aus dem Jahr 1997. Die offizielle Version 3.1 wurde am 11.03.2004 herausgegeben. Momentan wird die Version 4.0 bei der Organization for the Advancement of Structured Information Standards (OASIS) standardisiert, wo sie aktuell den Status des „Community Drafts" hat (OASIS UIML TC 2009).

3.1 Grundkonzept

UIML wurde mit dem Ziel entwickelt, jede mögliche Benutzungsschnittstelle beschreiben zu können. Zu diesem Zweck soll eine in UIML spezifizierte Benutzungsschnittstelle folgende Fragen beantworten können:

- Aus welchen Komponenten besteht eine Benutzungsschnittstelle?
- Wie kann diese Komponente dargestellt werden?
- Wie verhalten sich die einzelnen Komponenten?
- Wie können die Komponenten mit externen Komponenten (zum Beispiel der Applikationslogik) verbunden werden?
- Wie können Benutzungsschnittstellen auf eine Zielplattform abgebildet werden?

Bild 2: UIML Meta -Interface Modell



Die Beantwortung dieser Fragen wird durch die UIML-Grundstruktur, dem so genannten „Meta Interface Model", beschrieben. In Bild 2 ist diese Struktur sowohl mit den UIML-Elementen als auch grafisch dargestellt.

Eine UIML-Beschreibung besteht somit aus einem Wurzelelement <uiml>. Darunter finden sich die Top-Level-Elemente: <head> für die Metadaten der Beschreibung, <interface> für die Beschreibung der eigentlichen Benutzungsschnittstelle, <peers> für die Anbindung zwischen UIML-Vokabular und Implementierungssprache, sowie <template> für die Wiederverwendung von Teilelementen. Die folgenden Kapitel gehen näher auf die beiden wichtigen Elemente <interface> und <peers> ein. Struktur, Inhalt, Präsentation und Verhalten der Benutzungsschnittstelle werden in dem Element <interface> unter Verwendung der wichtigsten Unterelemente <structure>, <content>, <style> und <behavior> beschrieben.

Um diese Elemente einzuführen, wird im Folgenden ein durchgängiges Anwendungsszenario verwendet. Bei dem Beispiel handelt es sich um einen Teil eines Formulars (Eingabefeld für die Anzahl der zu buchender Zimmer), wie es typischerweise bei Hotelbuchungssystemen verwendet werden kann. Bei Eingabe von Daten in das Formular wird gleichzeitig eine Eingabeüberprüfung durchgeführt, die nicht im plattformabhängigen Code

ausprogrammiert werden muss. Es ist zu beachten, dass diese konkrete UIML-Darstellung auf einem generischen Vokabular basiert, in welchem bereits die Modalität feststeht – nämlich eine grafische Benutzungsschnittstelle – aber noch nicht die Zielplattform (z. B. Java, C#, etc.).

3.2 Strukturmerkmale

Das <structure>-Element gibt eine hierarchische Gliederung der Benutzungsschnittstelle wieder. Dabei lässt sich die Benutzungsschnittstelle in Teilelemente (<part>) weiter untergliedern. Ein Teilelement ist ein einzelner Baustein, der unabhängig von jeglichem Typ und konkreter Realisierung existiert. Ein Teilelement kann wiederum beliebig viele weitere Teilelemente enthalten (Containment-Prinzip), wodurch sich strukturierte und komplex verschachtelte Benutzungsschnittstellen definieren lassen.

Zur Unterstützung verschiedener Geräteklassen kann ein UIML-Dokument mehrere <structure>-Elemente enthalten. Es wäre beispielsweise denkbar, dass eine Benutzungsschnittstelle in UIML eine Struktur für einen PC (komplexe Benutzungsschnittstelle) und eine andere Struktur für ein Handy (einfache Benutzungsschnittstelle) enthält. Listing 1 zeigt ein Beispiel für mehrere <structure>-Elemente, die verschiedene Benutzungsschnittstellen definieren. Dabei entspricht die Struktur der <structure>-Elemente dem abstrakten Präsentationsmodell.

```
<structure id=„ComplexUI“>
      <part class=„c2“ id=„n3“>
      <part class=”c1” id=”n2”/>
   </part>
</structure>

<structure id=„SimpleUI“>
   <part class=„c1“ id=„n1”/>
</structure>

<structure id=„default“>
   <part class=„c1“ id=„n1”/>
   <part class=„c2“ id=„n2”/>
</structure>
```

Listing 1: Beispiel für <structure>-Elemente eines UIML-Dokuments

In dem konkreten Formular-Beispiel (Listing 2) besteht die Struktur aus einem Top-Level-Container (topContainer), der folgende Elemente enthält: eine Textanzeige (labelRooms), ein Feld für die Eingabe der Zimmeranzahl (editRooms), einen Container (buttonArea) für zwei Schalter zum Inkrementieren (buttonUP) bzw. dekrementieren (buttonDOWN) der Zimmeranzahl und einen Schalter zum abschicken der Anfrage (buttonSUBMIT).

```
<structure>
   <part id=„topContainer“
         class=„G:TopContainer“>
      <part id=„labelRooms“
            class=„G:TextBox”/>
      <part id=„editRooms“
            class=„G:TextField”/>
      <part id=„buttonArea“
            class=„G:Area“>
         <part id=„buttonUP“
               class=„G:Button”/>
         <part id=„buttonDOWN“
               class=„G:Button”/>
      </part>
   <part id=„buttonSUBMIT“
         class=„G:Button”/>
   </part>
</structure>
```

Listing 2: Konkrete Struktur des Beispielformularteils

3.3 Inhalt- und Präsentationsmerkmale

Der Inhalt jedes <part>-Elements kann mithilfe des <content>-Elements beschrieben werden. Beispielsweise können Texte, Bilder oder Klänge enthalten sein. Das <content>-Element ist wiederum eine Auflistung von beliebig vielen <constant>-Elementen, die die Daten beinhalten. Über diese Elemente können Aspekte wie Internationalisierung und Personalisierung in die Benutzungsschnittstelle integriert werden. Es ist somit möglich, mehrere <content>-Elemente zu definieren, deren Inhalte (<constant>-Elemente) z. B. verschiedene Sprachen repräsentieren.

Das <style>-Element enthält eine Liste aller Präsentationseigenschaften und Werte (z. B. Schriftart, Farbe). In jedem UIML-Dokument muss mindestens ein <style>-Element enthalten sein. Das <style>-Element kann beliebig viele <property>-Elemente beinhalten. Dieses Element verbindet ein Name-Wert-Paar mit einem Teil der Benutzungsschnittstelle oder einem Ereignis wie z. B. einem Mausklick.

Weiterhin ist das Layout für die Präsentation wichtig. Bis zur Version 3.1 war es die Aufgabe des UIML-Renderers das Layout festzulegen. Teilweise waren die Ergebnisse des Rendering-Prozesses nicht im Sinne des Designers. Deshalb besteht ab Version 4.0 die Möglichkeit die Position einzelner Elemente der Benutzungsschnittstelle anhand von Regeln zu definieren (z. B. Element A ist oberhalb von Element B platziert).

In dem durchgehenden Beispiel wird das Layout in Listing 3 näher festgelegt. So wird z. B. mit den ersten drei Properties bestimmt, dass das Textfeld den Text „No. of Rooms:“ erhält und das Texteingabefeld nur ein Zeichen breit und mit dem Wert „1“ vorbelegt sein soll.

```
<style>
   <property
      part-name=„labelRooms“
      name=„g:text“>No. of Rooms:
   </property>
   <property part-name=
      “editRooms“
      name=„g:columns“>1
   </property>
   <property part-name=
      “editRooms“
      name=„g:text“>1
   </property>
   …
</style>
```

Listing 3: Teil einer konkreten Stilbeschreibung in UIML

3.4 Verhaltensbeschreibung

In UIML wird das Verhalten der Benutzungsschnittstelle mit dem <behavior>-Element spezifiziert. Das <behavior>-Element beschreibt, wie die Benutzungsschnittstelle reagieren soll, wenn ein Benutzer mit ihr interagiert. Hierfür enthält jedes <behavior>-Element eine Reihe von <rule>-Elementen (Regeln), die wiederum jeweils ein <condition>-Element und mehrere <action>-Elemente enthalten. Das <rule>-Element verbindet somit eine Bedingung (<condition>) mit Aktionen (<action>). Ist die Bedingung erfüllt, dann wird die entsprechende Aktion ausgeführt. Der Wahrheitswert einer Bedingung ist entweder rein von einem Ereignis oder von einer Kombination des Ereignisses und eines booleschen Ausdrucks aus den Datenwerten abhängig. In dem ersten Fall ist die Bedingung nur

dann wahr, wenn das Ereignis eintritt. In dem zweiten Fall muss zusätzlich der Ausdruck ausgewertet werden. Häufig hat ein Ereignis der Benutzungsschnittstelle Auswirkungen sowohl auf die Anzeige als auch auf die dahinter stehende Applikationslogik (MVC-Paradigma). Diese Verhaltensbeschreibung entspricht im Wesentlichen der Modellierung des Dialogmodells.

Mit der Version 4.0 ist es neben der rein ereignisbasierten Verhaltensauswertung auch möglich, den internen Zustand des Dialogs mit zu berücksichtigen. Zu diesem Zweck gibt es ein <variable>-Element, welches solche Zusatzinformationen speichern kann, und das auch einfache arithmetische Operationen zulässt. So lässt sich mit UIML bspw. eine Verifikation von Nutzereingaben realisieren, ohne dass eine zusätzliche Kommunikation mit der Applikationslogik erfolgen muss. Dies ist vor allem im Hinblick auf Benutzungsschnittstellen für mobile Endgeräte relevant, um eine reaktive Bedienung zu gewährleisten.

Auch in dem bislang verfolgten Beispiel werden Variablen für diesen Zweck verwendet. Innerhalb des <behavior>-Elements in Listing 4 werden zuerst die Werte für die maximale und minimale Anzahl von zu buchenden Zimmern gesetzt. Anschließend werden Regeln als Paare von Bedingungen und Aktionen definiert. In dem Fall, dass z. B. der Text direkt in das Texteingabefeld eingegeben wurde, muss das entsprechende Ereignis gesetzt sein und der Inhalt des Textfeldes größer dem Wert von „MinNoRooms“ und kleiner als „MaxNoRooms“ sein. Diese Elemente werden in UIML über das <op>-Element logisch verknüpft. Nur dann kann der in <action> definierte Teil ausgeführt werden. Um auf die einzelnen Werte zuzugreifen, wird das entsprechende <property> bspw. über das Attribut „part-name“ angesprochen.

```
<behavior>
  <variable id=„MinNoRooms“
    type=„integer“>1
  </variable>
  <variable id=„MaxNoRooms“
    type=„integer“>4
  </variable>
  <variable id=„curNoRooms“
    type=„integer“>1
  </variable>
  <rule id=„textEntered“>
  <condition>
    <op name=„and“>
      <event part-name=
      „editRooms“
      class=
      „g:actionperformed“/>
    <op name=„greaterthan“>
      <property part-name=
      „editRooms“
      name=„g:text“/>
      <variable variable-name=
      „underMinNoRooms“/>
    </op>
    <op name=
    „lessthanandequal“>
      <property part-name=
      „editRooms“
      name=„g:text“/>
      <variable variable-name=
      „MaxNoRooms“/>
      </op>
    </op>
  </condition>
  <action>
    <op name=„set“>
      <variable variable-name=
      „curNoRooms“/>
      <property part-name=
      „editRooms“
      name=„g:text“/>
    </op>
  </action>
  </rule>
  …
</behavior>
```

Listing 4: Teil einer konkreten Verhaltensbeschreibung in UIML

In der Verhaltensbeschreibung können beliebig viele Regeln spezifiziert sein, die durch ihre IDs referenzierbar sind.

3.5 Abbildung von UIML in eine konkrete Plattform

Mittels des <peers>-Elementes ist es möglich, Abbildungen von UIML auf konkrete Plattformen (konkretes Präsentationsmodell) zu definieren. Das <peers>-Element besteht aus den Elementen <presentation> und <logic>. Das <presentation>-Element beschreibt die Abbildung der in UIML definierten Elemente der Benutzungsschnittstelle auf eine konkrete Plattform (z. B. in Java mit Swing). Mittels des <logic>-Elementes ist es möglich, die Schnittstelle zur Applikationslogik zu beschreiben.

Der UIML-Entwickler muss nicht bei jeder Zielplattform ein neues Vokabular definieren, sondern kann auf bereits vorhandene – auch fremde – Vokabulare zurückzugreifen. In dem durchgehenden Beispiel wurde dies angewendet, um auf das generische Vokabular für grafische Benutzungsschnittstellen zuzugreifen:

```
<peers>
  <presentation base=
  „Generic_1.3_Harmonia_1.0“>
</peers>
```

Listing 5: Anbindung eines externen Vokabulars

Bild 3 zeigt anhand eines Diagramms den Zusammenhang zwischen abstraktem Präsentationsmodell, konkretem Präsentationsmodell und der Ausführungsebene. In der Mapping-Ebene, als Zwischenschicht des abstrakten und konkreten Präsentationsmodells, wird die Zuordnung von korrespondierenden konkreten Elementen eines Vokabulars zu bisher rein abstrakten Elementen durchgeführt. Aus dem konkreten Präsentationsmodell wird in der Compiler-Ebene, als Zwischenschicht des konkreten Präsentationsmodelles und der Ausführungsebene, direkt Quellcode für die Zielplattform erzeugt.

3.6 Wiederverwendbarkeit

UIML wurde auch im Hinblick auf die Wiederverwendbarkeit von (Teilen der) Benut-

Bild 3: UIML-Abbildungskonzept einer generischen Benutzungsschnittstelle

zungsschnittstelle entwickelt. Zu diesem Zweck existiert das <template>-Element, welches ein nahezu beliebiges UIML-Element mit all seinen Unterstrukturen enthalten kann. Ein solches <template>-Element kann im <interface>-Teil referenziert werden, wobei es unterschiedliche Regeln gibt, nach denen die <template>-Inhalte eingefügt werden können. Diese Regeln orientieren sich an den Mechanismen der Cascading Style Sheets (CSS). In UIML 4.0 gehen die Templates über eine reine Schablonenfunktionalität hinaus, indem sie parametrisierbar sind. Folgendes Beispiel soll den Sinn der Parametrisierung verdeutlichen: In einem Template könnte z. B. ein Schalter mit „OK" definiert sein, in einem anderen Template ein Schalter mit der Beschriftung „Yes". In beiden Fällen wird der Schalter benutzt, um einen Dialog abzubrechen. Es ist also naheliegend auch das Verhalten im Template mit zu spezifizieren. Allerdings benötigt das <action>-Element unterschiedliche IDs, da sonst immer nur der gleiche Teil der Applikationslogik angesprochen werden kann. Daher können Template-Parameter eingesetzt werden, die dann für die entsprechenden Fälle unterschiedliche Werte besitzen.

3.7 Werkzeugunterstützung

Eine XML-basierte Modellierungssprache ist nur dann praxistauglich, wenn es entsprechende Werkzeugunterstützung gibt, vor allem im Bereich grafischer Editoren (Authoring Tools) und Entwicklungsumgebungen (Petrasch 2007). Mittlerweile wird an mehreren Entwicklungsumgebungen gearbeitet. Im Folgenden werden zwei Entwicklungsumgebungen vorgestellt.

LiquidApps

Die Firma Harmonia hat eine aktuelle Testversion von LiquidApps 6.0.1 (LiquidApps 2009), einer Entwicklungsumgebung für UIML, zur Verfügung gestellt. Mit dieser Software ist es möglich, Benutzungsschnittstellen nach dem WYSIWYG-Prinzip zu entwerfen. In einem zweiten Schritt lässt sich entweder die UIML-Struktur exportieren oder in direkten Quellcode einer Programmiersprache transformieren.

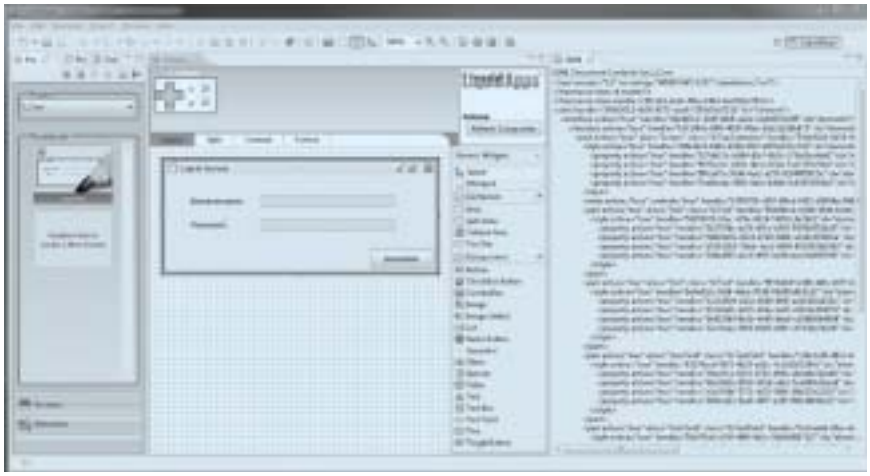


Bild 4: Screenshot eines Beispielprojektes in LiquidApps 6.0.1

Mittels eines vordefinierten generischen Vokabulars, wie bspw. dem „Generic_1.3_Harmonia_1.0", welches in der Testversion von LiquidApps integriert ist, lässt sich nunmehr eine Benutzungsschnittstelle mit generischen Elementen definieren. Die daraus entstehende Benutzungsschnittstelle, die von konkreten Elementen losgelöst ist, wird dann mittels eines integrierten Compilers in finale Benutzungsschnittstellen bzw. Quellcode, wie z. B. HTML, Java mit Swing oder C++ mit Qt transformiert.

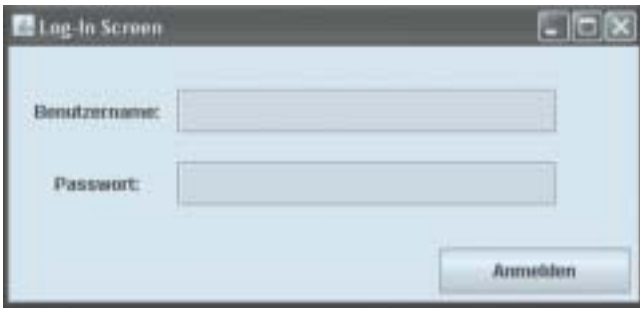
Abbildung 4 zeigt einen Screenshot der aktuellen LiquidApps Oberfläche. Die Oberfläche der grafischen Entwicklungsumgebung ist in mehrere Bereiche aufgeteilt: „Screens" zur Verwaltung der einzelnen Sichten (Fenster) der zu entwickelnden Benutzungsschnittstelle, "Behaviours" zum Spezifizieren des Verhaltens (Dialogmodell), „Properties" zur Einstellung der Parameter einzelner Interaktionsobjekte wie z. B. der Hintergrundfarbe von Textfeldern, den aktuellen „Screen" zum Entwerfen der jeweiligen Sicht und einer Anzeige der „Generic Widgets", die die verfügbaren Interaktionsobjekte enthält. Ein weiteres Fenster, welches eingeblendet werden kann, zeigt den (automatisch erstellten) UIML-Quellcode (in Bild 4 rechts dargestellt).

Beispielhaft wurde für den vorliegenden Beitrag eine einfache Benutzungsschnittstelle, ein Log-In Screen, entworfen und nach HTML (Bild 5), Java/Swing (Bild 6) und C++/Qt (Bild 7) exportiert. Die Erstellung dieses Beispiels mit LiquidApps konnte ohne Einarbeitung in sehr kurzer Zeit durchgeführt werden. Zur besseren Abgrenzung wurde die Hintergrundfarbe der „Textboxen" gelb hinterlegt.

UIML.NET

Einen anderen Weg geht das Projekt UIML.NET (Luyten und Coninx 2004), das am Expertise Centre for Digital Media

◀ **Bild** 5: HTML Log-In Screen



▲ **Bild** 6: Java/Swing Log-In Screen

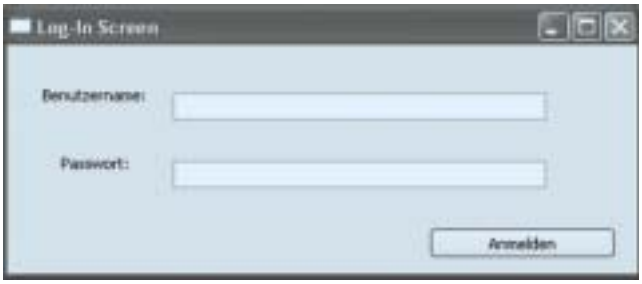
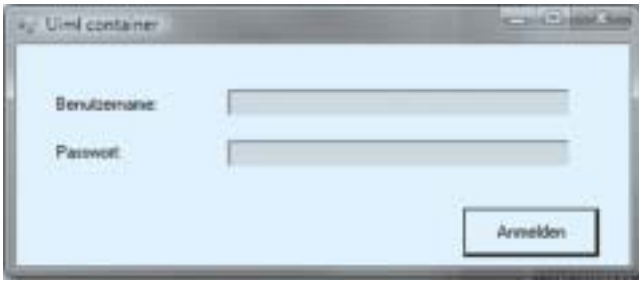
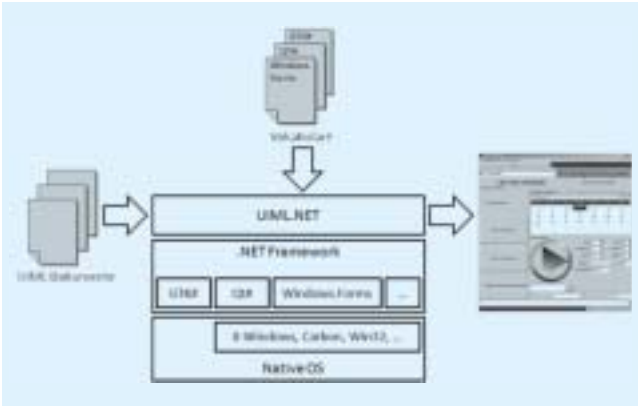


Bild 7: C++/Qt Log-In Screen





▲**Bild 8:** Grobe Darstellung der UIML. NET-Architektur (Luyten und Coninx 2004)

◀ **Bild 9:** Log-In Screen des in UIML.NET (System.Windows.Forms) interpretierten UIML-Dokumentes

(EDM) der Hasselt-Universität in Belgien entwickelt wird. UIML.NET ist ein freier Interpreter für UIML, umgesetzt in C#. Dabei unterstützt UIML.NET verschiedene Vokabulare bzw. Widget-Toolkits: GTK#, System.Windows.Forms, Compact System.Windows.Forms und wx.NET. Das Projekt wird unter der General Public License (GPL) entwickelt, so dass der Quellcode des Projektes frei zugänglich ist.

Ziel von UIML.NET ist nicht die Überführung bzw. das Kompilieren von UIML in diverse Programmiersprachen, sondern das Interpretieren bzw. Rendern von UIML-Dokumenten (Luyten und Coninx 2004) auf Abruf. Dabei wird das Widget-Toolkit zur Laufzeit gewählt, was eine flexiblere Anpassung der Repräsentation der Benutzungsschnittstelle erlaubt. Bild 8 zeigt eine grobe Darstellung der UIML.NET-Architektur.

Um den Interpreter-Vorgang ausführen zu können, werden ein UIML-Dokument und ein Vokabular benötigt. Aus diesen Komponenten wird die (grafische) Benutzungsschnittstelle berechnet. Bild 9 zeigt einen Screenshot des interpretierten Log-In Screens, welches das Widget-Toolkit der System.Windows.Forms verwendet.

Mittlerweile gehört zu UIML.NET auch ein Authoring-Tool, namens Gummy (Meskens et al. 2008), welches über einen ähnlichen Funktionsumfang wie LiquidApps verfügt. UIML.NET und auch Gummy befinden sich, im Gegensatz zu LiquidApps, noch in einem frühen Entwicklungsstadium. Der Quellcode von UIML.NET und die entsprechenden Anwendungen können von der Projekt-Webseite kostenfrei heruntergeladen werden.

4. Bewertung

UIML bietet zwar keine Möglichkeit, ein Aufgabenmodell zu spezifizieren (vgl. Petrasch 2007), jedoch kann dieses Problem mit einer Anbindung bereits existierender Notationen für Aufgabenmodelle, wie z. B. useML (Meixner und Görlich 2008) behoben werden. Somit wäre es möglich, einen durchgängigen modellbasierten Entwicklungsprozess, ganz im Sinne von MBUID, zu entwickeln.

Eine Einschränkung von UIML liegt (Souchon und Vanderdonckt 2003) zufolge darin, dass für verschiedene Plattformen, wie bspw. HTML oder Java/Swing, multiple Benutzungsschnittstellen (mittels des <presentation>-Elementes) definiert werden müssen, da jede der Plattformen über eigene Elemente zur Darstellung einer (finalen) Benutzungsschnittstelle verfügt. Diese Einschränkung wurde jedoch durch die Einführung eines generischen Vokabulars (vgl. Simon, Jank und Wegscheider 2004) behoben. Bei einem ersten Test, der Entwicklung eines einfachen Log-In Screens gab es keine Probleme bzgl. der Generierung der Benutzungsschnittstellen. Um eine Benutzungsschnittstelle auf mehrere Plattformen zu exportieren, musste nur einmal eine deklarative Beschreibung erstellt werden. Dabei ist besonders die einfach und intuitiv zu bedienende grafische Benutzungsschnittstelle von LiquidApps zu erwähnen, mit der in sehr kurzer Zeit ein Log-In Screen entwickelt werden konnte.

Aktuell ist LiquidApps in der Lage, aus einer Benutzungsschnittstellenbeschreibung des generischen Vokabulars „Generic_1.3_Harmonia_1.0" konkrete Benutzungsschnittstellen in Java/Swing, C++/Qt und HTML zu exportieren. Mittels weiterer Transformationsbeschreibungen für dieses Vokabular wäre es möglich, auf beliebige Plattformen zu exportieren. Der Entwicklungsaufwand beschränkt sich somit auf die Erstellung der Transformationsbeschreibungen der jeweiligen Zielplattform. Somit wird der Entwicklungsaufwand weiterer Benutzungsschnittstellen drastisch reduziert, da auf bereits vorhandene Transformationsbeschreibungen und Vokabulare zurückgegriffen werden kann.

Einen anderen Weg zur Generierung von Prototypen für Benutzungsschnittstellen bietet die Interpretation von UIML-Dokumenten, wie in UIML.NET gezeigt. Dadurch, dass UIML-Dokumente interpretiert und nicht in Quellcode von Programmiersprachen kompiliert werden, können Benutzungsschnittstellen dynamischer gehandhabt werden. Somit muss der Quellcode in der jeweiligen Zielplattform nicht noch in ein ausführbares Pro-

gramm kompiliert werden. Da sich das Projekt allerdings noch in einer sehr frühen Phase seiner Entwicklung befindet, kann hier noch keine konkrete Aussage über die Verwendbarkeit getroffen werden. Hier muss abgewartet werden, wie sich das Projekt und die praktische Einsetzbarkeit von UIML.NET entwickelt.

Ein weiterer wichtiger Aspekt bei der Verwendung von UIML als Beschreibungssprache für Benutzungsschnittstellen wird in der UIML-Spezifikation 4.0 klar dargestellt: UIML darf von jeder Person frei implementiert werden. Es fallen dabei keinerlei Lizenzgebühren an. Zusätzlich steht UIML kurz vor der Verabschiedung als OASIS-Standard, was seine Anwendung und Verbreitung zusätzlich fördern dürfte.

Zusammenfassend lässt sich schlussfolgern, dass mit einer Werkzeugunterstützung wie zum Beispiel LiquidApps die plattformübergreifende, abstrakte Definition von Benutzungsschnittstellen mit UIML bereits heutzutage einfach und schnell machbar ist; Anpassungen am Quelltext sind anschließend aber weiterhin notwendig. Die vollständige Laufzeitgenerierung von Benutzungsschnittstellen (quasi „on demand") steckt heute jedoch noch in einer frühen Entwicklungsphase und bedarf zunächst der Festlegung von Standards und deren Verbreitung. UIML.NET kann insbesondere im Hinblick auf aktuelle Entwicklungen im Bereich des Ubiquitous Computing als Alternative angesehen werden.

Literatur

Calvary, G.; Coutaz, J.; Thevenin, D.; Limbourg, Q.; Bouillon, L.; Vanderdonckt, J.: A unifying reference framework for multi target user interfaces. In: *Interacting with Computers* Band **15**, Heft **3** (2003) 289–308.

Gruhn, V.; Pieper, D.; Röttgers, C.: MDA. *Effektives Software-Engineering mit UML 2 und Eclipse.* Berlin: Springer, 2006.

Limbourg, Q.; Vanderdonckt, J.; Michotte, B.; Bouillon, L.; López-Jaquero, V.: Usixml: A language supporting multi-path development of user interfaces. In: *LNCS 3425* (Hrsg. Bastide, R.; Palanque, P.; Roth, J.), 200–220. Springer, 2004.

LiquidApps: http://www.liquidappsworld.com/index.php (Letzter Zugriff: 21.01.2009).

Luyten, K.: Dynamic User Interface Generation for Mobile and Embedded Systems with Model-Based User Interface Development. PhD thesis, Transnationale Universiteit Limburg, 2004.

Luyten, K.; Coninx, K.: UIML.NET: an open UIML Renderer for the .NET Framework. In: *Proc. of the 5th International Conference on Computer-Aided Design of User Interfaces.* Kluwer Verlag, (2004) 257–268.

Meixner, G.; Görlich, D.: Aufgabenmodellierung als Kernelement eines nutzerzentrierten Entwicklungsprozesses für Bedienoberflächen. Workshop „Verhaltensmodellierung: Best Practices und neue Erkenntnisse", Fachtagung Modellierung, 2008.

Meixner, G.; Görlich, D.; Schäfer, R.: Unterstützung des Useware-Engineering Prozesses durch den Einsatz einer modellbasierten Werkzeugkette, In: *VDI-Berichte 2041*, Düsseldorf: VDI-Verlag (2008) 219–232.

Meskens, J.; Vermeulen, J.; Luyten, K. et al.: Gummy for multi-platform user interface designs: shape me, multiply me, fix me, use me. In: *Proc. of the working conference on Advanced visual interfaces* (2008) 233–240.

Mori, G.; Paternò, F.; Santoro, C.: Design and Development of Multidevice User Interfaces through Multiple Logical Descriptions. In: *IEEE Transactions on Software Engineering* IEEE Press (2004) 507–520.

OASIS UIML TC: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml (Letzter Zugriff: 21.01.2009).

Paris, C.; Lu, S.; Vander Linden, K.: Environments for the Construction and Use of Task Models. In: *The Handbook of Task Analysis for Human-Computer Interaction.* (Hrsg. Diaper, D.; Stanton, N.) Lawrence Erlbaum Associates, 2004.

Paternò, F.: *Model-based design and evaluation of interactive applications.* London: Springer, 1999.

Petrasch, R.: Modellbasierte Entwicklung von Web 2.0 Anwendungen mit MDA. *i-com, Zeitschrift für interaktive und kooperative Medien* **6, 1** (2007) 28–32.

Puerta, A.: A Model-Based Interface Development Environment. *IEEE Software* Band **14**, Heft **4** (1997) 40–47.

Reuther, A.: useML – systematische Entwicklung von Maschinenbediensystemen mit XML. Fortschritt-Berichte pak, Band 8. Kaiserslautern: Technische Universität Kaiserslautern, 2003.

Simon, R.; Jank, M.; Wegscheider, F.: A Generic UIML Vocabulary for Device- and Modality Independent User Interfaces. In: *Proc. of the 13th International World Wide Web Conference* (2004) 434–435.

Souchon, N.; Vanderdockt, J.: A Review of XML-Compliant User Interface Description Languages. In: *Proc. of the 10th International Workshop on Interactive Systems: Design, Specification and Verification* (2003) 377–391.

Vanderdonckt, J.; Bodart, F.: Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection. In: *Proc. of the 1st Annual CHI Conference on Human Factors in Computing Systems* (1993) 424–429.

Zühlke, D.: *Useware-Engineering für technische Systeme.* Berlin: Springer, 2004.


1


2

1 Gerrit Meixner studierte im Diplom- und anschließend im Master-Studiengang Informatik an der Fachhochschule Trier. Seit 2007 arbeitet er als wissenschaftlicher Mitarbeiter am DFKI. Sein Hauptarbeitsgebiet ist die Weiterentwicklung des Useware-Engineering Prozesses mit dem Forschungsschwerpunkt der Entwicklung einer Werkzeugkette für die modellbasierte Bediensystementwicklung. Weiterhin war Gerrit Meixner als öffentlicher Gutachter im Standardisierungsprozess von UIML 4.0 aktiv.
E-Mail: Gerrit.Meixner@dfki.de
http://www.dfki.de, http://www.zmmi.de

2 Dr. rer. nat. Robbie Schäfer hat 2007 an der Universität Paderborn promoviert, bei der er von 2001 bis 2007 als wissenschaftlicher Mitarbeiter angestellt war. Das Thema seiner Dissertation lautet „Model-Based Development of Multimodal and Multi-Device User Interfaces in Context-Aware Environments". Weiterhin ist Robbie Schäfer als einer der Editoren der UIML 4.0 Spezifikation in der Standardisierung aktiv. Zurzeit arbeitet er bei einen IT-Dienstleister und Lösungsanbieter für Unternehmen der Energie- und Wasserwirtschaft.
E-Mail: schaefer.robbie@googlemail.com
http://www.c-lab.de/vis/robbie