VoiceXML Test-Framework

Abstract

VoiceXML[1] ist als W3C Standard eine etablierte Sprache zur Entwicklung sprachbasierter Anwendungen. Es ist für Sprache das, was XHTML für grafisch aufbereitete Webseiten ist. So wäre es beispielsweise auch für nicht versierte Fachkräfte möglich, Anwendungen für eine Telefonanlage vollständig mit VoiceXML auf einfache Art und Weise zu programmieren. Die kommende VoiceXML 3.0 Version zielt aber auch auf Anwendung ohne Telefonie-Anbindung. Ein erster Schritt in diese Richtung ist der aktuell veröffentlichte MMI Standard[2] zur Entwicklung multimodaler Anwendungen. Obwohl die Sprache ausgereift und insbesondere im Bereich Telefon-basierte Anwendungsentwicklung stark verbreitet ist, existieren nur wenige ausgefeilte Testwerkzeuge, die direkt für VoiceXML eingesetzt werden können. In der Regel müssen die Entwickler selber zum Telefonhörer greifen, um die korrekte Funktionsweise der Anwendung zu überprüfen. Gerade im Hinblick auf größere Deployments ist dieses Verfahren aber ungeeignet.

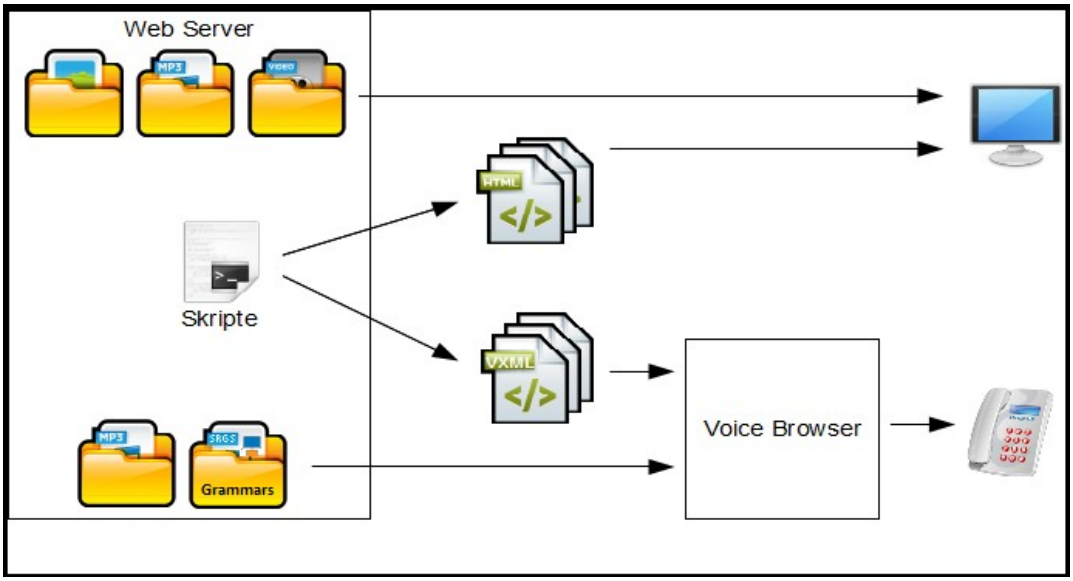

Abb. 1: Deployment

Automatisierte Tests[3], wie sie auch aus anderen Entwicklungen, zum Beispiel mit JUnit[4], bereitgestellt werden, fehlen hier. Es gibt erste Ansätze, wie den SoftRobot von Devoteam[5], die jedoch auf Systemtests zielen und somit in der Entwicklungsphase ungeeignet erscheinen. Der Einsatz von testgetriebenen Entwicklungsansätzen ist in einem solchen Umfeld schlichtweg unmöglich. Es fehlen geeignete Hilfsmittel für Unit-Test[6] (Komponententest) und Integrationstest.

VoiceXML ist vom Design her eine relativ einfache Programmiersprache und damit leicht zu validieren. Eine große Beschränkung besteht darin, dass manuell nur wenige Tests aufgrund des zu hohen zeitlichen als auch technischen Aufwands durchgeführt werden können. Im Besonderen für Designer und Entwickler besteht das gleiche Problem, weil diese während ihrer Arbeit eine große Menge verschiedener Dokumente in VoiceXML erstellen. Änderungen am Framework führen nicht zur Gewissheit, mögliche Regressionen, vorhandene Bugs oder unerwünschte funktionelle Veränderungen zu finden, ohne automatisiert die Prinzipien von Unit-Test und Integrationstest zur Verfügung zu haben.

---

1 http://www.w3.org/TR/voicexml21/

2 http://www.w3.org/2002/mmi/

3 Andreas Spillner, Tilo Linz: Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard, 2010

4 http://www.fh-wedel.de/~si/seminare/ws02/Ausarbeitung/6.junit/layout2.htm

5 Wolfgang Sachse: SoftRobot, 2009, http://www.devoteam.de/fileadmin/P_DG/2012/WhitePapers/SoftRobot_Devoteam_1_8_0_wp_deu.pdf

6 Kent Beck: Test Driven Development By Example, 2002

VoiceXML Test-Framework

Zielsetzung

In dieser Präsentation soll unser Ansatz[7] aufgezeigt werden, der auf den bestehenden Standards aufsetzt und sich gut in aktuelle Entwicklungsumgebungen integrieren lässt. Er bietet eine gute programmatische Einbettung und zielt auf ein JUnit-artiges Framework für VoiceXML.

Das Framework wurde mit Hilfe des Open Source Voice Browsers JVoiceXML[8] prototypisch implementiert und ist selbst als Open Source verfügbar[9]. Es bietet eine solide Basis, um testgetriebene Entwicklung von sprachbasierten Applikationen überhaupt zu ermöglichen. Es ist ebenso möglich, sowohl auf JUnit basierende Testfälle zu schreiben, als auch das Framework über einen Webservice (HTTP/SOAP/XMPP) zu betreiben.

Einführung

Wenn man den Gedanken des Standards „Software Unit Test“, also die Norm IEEE 1008[10], folgt, so sollten Unit Tests geschrieben werden, bevor ein Entwickler auch nur eine Zeile Code für das zu entwickelnde System schreibt. Man spricht hier vom „Test first“-Ansatz. Auf diese Art und Weise entsteht gleichzeitig eine Dokumentation über die abgelieferte Qualität. Das ist prinzipiell unabhängig vom gewählten Vorgehensmodell.
Im Zuge von testgetriebener Entwicklung („Test Driven Development“: TDD) stößt man aber zwangsläufig auf moderne agile Methoden. Prominente Vertreter finden sich zum Beispiel in “Scrum” oder “Extreme Programming” (XP). In Form von sogenannten „Greybox“-Tests werden die Tests bereits vor den entsprechenden Komponenten implementiert. Zudem sind die Tests so zu gestalten, dass sie möglichst weitgehend automatisiert durchgeführt werden können. So ist es möglich, nach einem “commit” oder “push” in das jeweilige Repository, einen automatischen Test auszulösen. Die zuletzt eingepflegten Änderungen werden dann gegen die nach wie vor bestehenden Anforderungen validiert. Damit kann möglichen und unabsichtlich eingebauten Seiteneffekten vorgebeugt werden, die von Entwicklern zumeist nicht betrachtet werden, da sie meist unabhängig vom aktuell geänderten oder implementierten Code an völlig anderen Stellen und in unerwarteten Erscheinungen auftauchen. Werden solche Fehler zu spät entdeckt, verursachen sie enorme Kosten beim Endbenutzer („10er Kosten-Regel“).

Autorenbeschreibung

Dr. Dirk Schnelle-Walka (E-Mail: dirk@tk.informatik.tu-darmstadt.de) hat 2007 im Bereich Voice User Interface Design an der TU Darmstadt promoviert. Seit 2009 ist er Gruppenleiter am Fachgebiet Telekooperation an der TU Darmstadt und forscht auf dem Gebiet multimodaler Interaktion in Smart Environments.

Raphael Groner (E-Mail: raphgro@web.de) hat Produktionsmanagement und Logistik an der Hochschule Reutlingen studiert. Er hat eine Zertifizierung zum CTFL vom ISTQB®.

7 http://sourceforge.net/apps/mediawiki/jvoicexml/index.php?title=UnitTest
8 http://sourceforge.net/apps/mediawiki/jvoicexml/index.php?title=Main_Page
9 http://jvoicexml.org
10 http://standards.ieee.org/findstds/standard/1008-1987.html