Ausführbare Spezifikationen im Projektalltag – Ein Erfahrungsbericht

Jens Nerche

Kontext E GmbH
Fetscherstr. 10
01307 Dresden
j.nerche@kontext-e.de

Abstract: Im Umfeld der agilen Softwareentwicklung sind Behavior Driven Development und Ausführbare Spezifikationen verbreitet. Die Anforderungsanalyse und -dokumentation sowie die Definition von Akzeptanzkriterien für die Anforderungen wird verbunden mit der Erstellung von maschinell ausführbaren Testfällen zur Prüfung der Akzeptanzkriterien, um eine Suite automatischer Testszenarien zu erhalten und den manuellen Aufwand auf das explorative Testen beschränken zu können. Dafür wurden Tools wie Cucumber und FitNesse entwickelt, so dass die Akzeptanztests wie Unit Tests ausführbar sind. Dabei fehlt entweder die IDE-Unterstützung oder es werden Interne DSLs verwendet, bei denen die Syntax der Hostsprache limitierend wirkt. Medienbrüche erschweren die Erstellung und das Refactoring der Testfälle. Außerdem sind keine echten, navigierbaren Referenzen vom Code auf die Anforderungen der Spezifikation möglich. Im Beitrag werden ausführbare Spezifikationen vorgestellt, bei denen diese Limitierungen überwunden wurden. Gherkin und Use Case Formulare werden direkt ausführbar, der Editor bietet den Komfort einer modernen IDE. Die Spezifikationen werden in Externen Domänenspezifischen Sprachen geschrieben, so dass der Fachabteilung bekannte Notationen und Textformatierungen verwendet werden können. Im Erfahrungsbericht wird der Weg über verschiedene Ansätze nachgezeichnet, Ausführbare Spezifikationen im täglichen Projektalltag einzusetzen. Den Schwerpunkt bilden die Erfahrungen, die in den letzten beiden Jahren mit Externen Domänenspezifischen Sprachen gesammelt wurden, die auf der Open Source Language Workbench MPS basieren.

1 Einleitung und Motivation

Das “Single Level of Abstraction”-Principle [Ma09] verlangt, dass verschiedene Abstraktionsniveaus nicht vermischt werden. Bei Unit Tests ist dies der Fall, sie werden in der selben Programmiersprache verfasst wie der Produktivcode. Hingegen sind Akzeptanztests entweder auf dem Abstraktionslevel der Anforderungen, aber nicht ausführbar. Oder sie sind ausführbar, dann jedoch häufig in der Progammiersprache des Produktivcodes geschrieben. Damit unterscheiden sie sich in der Sprache, der Notation und dem

Speicherort im Dateisystem von den Anforderungen. Wünschenswert wäre, die Spezifikation mit den Anforderungen mit den dazugehörenden Tests in einem gemeinsamen Artefakt zu pflegen – in der gleichen Sprache und der gleichen Notation. Anforderungen und dazugehörende Tests sollen gemeinsam in einem lebendigen Dokument verwaltet werden. Die bisherigen Lösungen, die in Kapitel 2 beschrieben werden, können dies nicht oder nur unbefriedigend leisten. In Kapitel 3 werden daher Anforderungen für ein neues Werkzeug beschrieben, dessen Entwicklung in Kapitel 4 beschrieben wird. Die Ergebnisse werden in Kapitel 5 dargestellt.

2 Stand der Technik

2.1 Textverarbeitungsprogramme

Am häufigsten zur Anforderungsbeschreibung benutzt, obwohl am wenigsten dafür geeignet, werden aktuell Textverarbeitungsprogramme. Die Autoren verwenden statt formaler Sprachschablonen (vgl. [RS07]) nur Prosaformulierungen, so dass Mehrdeutigkeiten und grobe Unvollständigkeiten üblich sind.

Textverarbeitungen bieten weder Werkzeuge zur Verwaltung von Anforderungen noch eine Unterstützung von Workflows, keine Versionskontrollsystemintegration, sowie nur mangelhafte Unterstützung paralleler Arbeit.

Für Textverarbeitungsprogramme spricht lediglich, dass sie überall verfügbar sind und die Bedienung zum Allgemeinwissen zählt.

2.2 Wandtafeln und Issue Tracking Systeme

Speziell in der Umgebung agiler Softwareentwicklung kommen Wandtafeln mit Klebezetteln oder Issue Tracking Systeme zum Einsatz. Letztere bieten ein Kanban-Board und Scrum-Sprintplanungswerkzeuge an. Diese Vorgehensweiste passt zwar sehr gut zur Planung in agilen Projekten, jedoch ist eine Integration der automatisierten Tests nicht möglich.

2.3 Spezialisierte RE&M Werkzeuge

In Großprojekten sind eher spezialisierte RE&M-Werkzeuge wie DOORS [ID14] oder Polarion [PS14] anzutreffen. Auch hier geschieht die Erfassung von Anforderungen textuell. Die Möglichkeiten zur Verwaltung sind oft hervorragend, jedoch muss die Testausführung noch manuell geschehen.

2.4 Interne DSLs

Die ersten Integrations- und Akzeptanztests für diesen Kunden wurden mit ScalaTest [ST14] erstellt. Scala [Sc14] wurde wegen der Typsicherheit, der guten Eignung der flexiblen Syntax für Interne DSLs und der Verfügbarkeit auf der Java Virtual Machine (JVM) gewählt. Zwar ist ScalaTest-Code für Progammierer sehr gut lesbar, aber die Hostsprache ist noch so deutlich erkennbar, dass sie für die gemeinsame Testfallentwicklung mit dem Kunden oder als alleinige Dokumentation nicht geeignet ist.

2.5 Externe DSLs ohne Language Workbench

Bekannte Externe DSLs für Ausführbare Spezifikationen sind FIT [FIT14], FitNesse [FN14] und die JBehave/RSpec/Cucumber-Familie [Cu14]. Ohne Language Workbenches fehlen die Funktionalitäten, die mit dieser zusätzlich zur Sprache entwickelt würden wie Editor, Refactorings, Inspections und Quick Fixes.

2.6 Externe DSLs mit Xtext-basierter Language Workbench

Auf dem Markt erhältlich sind seit längerem Produkte basierend auf Xtext-Language Workbenches [Xt14]. Die bekanntesten Vertreter dürften Jnario [Jn14] und NatSpec [NS14] sein.

2.7 Externe DSLs mit MPS-basierter Language Workbench

mbeddr [mb14] ist eine Sammlung Externer DSLs basierend auf MPS [MPS14]. Obwohl mbeddr die Entwicklung in Embedded Systemen mit C zum Ziel hat, kann man darin auch Anforderungen erfassen. Eine Besonderheit ist, dass die Lösung in C auch in mbeddr geschrieben wird. Dadurch werden Links zwischen Anforderungen und Implementierung möglich. Diese Links können einfach navigiert werden, außerdem sind Prüfungen vorhanden wie z.B. ob für eine Anforderung eine Implementierung vorliegt.

3 Anforderungen

In dem neu zu entwickelnden Werkzeug sollten die drei wichtigsten Eigenschaften von bestehenden Produkten kombiniert werden, um die Programmwechsel, Medienbrüche und Inkonsistenzen durch Wiederholungen zu vermeiden:

(1) Erklärende Erläuterungen sollen wie in Textverarbeitungsprogrammen geschrieben werden können. Der Text muss in Kapitel mit Überschriften gliederbar sein, Texthervorhebungen durch kursive und fette Schrift, die Vorder- und Hintergrundfarbwahl, das Einfügen von Tabellen und Bildern, Grafiken und Diagrammen sollen möglich sein. Teile oder die ganze Spezifikation sollen als PDF exportiert werden können.

(2) Man soll Anforderungen erheben und verwalten können wie in RE&M-Tools. Verschiedene Notationsformen und Sichten wie Anwendungsfall- oder Anforderungsformulare müssen möglich sein. Symbole und Konventionen der Domäne und Fachabteilungen sind für die Lesbarkeit durch Sachbearbeiter und Manager wichtig. Der Fortschritt der Entwicklung muss leicht nachvollziehbar sein. Anforderungen müssen nach verschiedenen Kriterien gefiltert und Statistiken zur Übersicht angezeigt werden können. Bestimmte Aspekte wie z.B. das Datenmodell sollen semiformal oder formal modellierbar sein. Oft ist es hilfreich, Beispieldokumente mit den Anforderungen zu verknüpfen, d.h. einen navigierbaren Link zwischen ihnen zu setzen.

(3) Die Akzeptanzkriterien in Form von automatisierten Tests müssen wie in der IDE programmierbar sein. Übliche Hilfen wie Syntax Highlighting, Intentions, Inspections, Quick Fixes, Vervollständigungshilfen, ein Debugger, Integration in den Continuous Integration Build, Möglichkeit zum Continuous Delivery, Refactorings, Stacktraceanalyse müssen vorhanden sein. Sowohl einzelne Tests als auch Testgruppen und ganze Testsuiten sollen direkt aus dem Programm heraus gestartet und der Testreport angezeigt werden. Die gängigen Versionsverwaltungssysteme Git [Git14] und Subversion [Svn14] sind zu unterstützen.

Als erheblich einschränkende Randbedingung gilt es zu beachten, dass es "nur" ein internes Unterstützungswerkzeug ist und nicht der Hauptteil des Projekts. Somit darf der Entwicklungsaufwand nur relativ gering sein. Zum Ausgleich braucht die Lösung nicht perfekt zu sein, sondern vorerst für internen Gebrauch hinreichend ausgearbeitet.

4 Lösungsentwicklung

Getrieben von der Einschränkung des begrenzten Entwicklungsbudgets fiel die Entscheidung, im ersten Schritt vor allem eine solide technische Grundlage zu legen. Wenngleich zunächst nur die in diesem Projekt benötigten Teile ausprogrammiert wurden, sollte der Entwurf grundsätzlich offen für Erweiterungen sein.

Basierend auf dem in Kapitel 2 beschriebenen Stand der Technik wurde eine Language Workbench als erfolgversprechendster Ansatz ausgewählt. Als wichtigste Entscheidungskriterien sind aufzuzählen:

- Textverarbeitungen kamen wegen der vielen o.g. Nachteile nicht in Frage.
- Gegen Issue Tracking Systeme sprach, dass sie für den dritten Punkt der Anforderungen (Programmierbarkeit wie in einer IDE) keinerlei Unterstützung bieten, diese Anforderung aber seitens der Entwickler als die wichtigste eingestuft wurde.
- Gleiches gilt für spezialisierte RE&M-Werkzeuge sowie für Externe DSLs ohne Language Workbench.
- Interne DSLs sind auf die Syntaxmöglichkeiten der Hostsprache begrenzt. Diese Grenzen bilden das Ausschlusskriterium bzgl. des ersten Punktes der Anforderungen (erklärende Erläuterungen wie in einer Textverarbeitung).

- Für Externe DSLs mit Language Workbench spricht die reichhaltige Unterstützung des auf Sprachentwicklung mit IDE-Unterstützung spezialisierten Werkzeugs. Möglichkeiten zur Anforderungsverwaltung fehlen dafür komplett. Diese für eine relativ geringe Anzahl von Anforderungen nachzubilden ist aber der geringere Aufwand, weswegen die Entscheidung zugunsten der Language Workbench ausfiel.

Auf Grund der langjährigen Erfahrungen mit externen domänenspezifischen Sprachen, speziell mit der Language Workbench MPS, fiel die Entscheidung auf dieses Produkt.

MPS wird seit über zehn Jahren bei der Firma JetBrains entwickelt. Es steht als Open Source unter der Apache 2.0 Lizenz [AL04] zur Verfügung. MPS verfolgt den Ansatz des "Projektionalen Editors". Dabei wird nicht wie sonst üblich eine vom Programmierer erzeugte Textdatei geparst und in einen Abstrakten Syntaxbaum (AST) umgewandelt, sondern der AST wird direkt editiert und das Parsen entfällt. Die größte Stärke dieser Vorgehensweise ist, dass Programmiersprachen sehr einfach modularisiert, erweitert und kombiniert werden können. Da die Language Workbench selbst als Plugin-Architektur realisiert ist, wird die Offenheit für Erweiterungen sowohl durch die Plattform als auch durch die Sprachen maximal unterstützt.

Um die Bearbeitung des AST für den Programmierer so einfach wie möglich zu machen, wird für jedes Sprachkonzept eine oder mehrere Editor-Repräsentationen definiert. Der AST wird also in die Darstellungsform projiziert, womit sich der Name "Projektionaler Editor" erklärt. Durch die Projektion bekommt die abstrakte Syntax der Sprache eine oder mehrere konkrete Syntaxen. [Vö13] gibt eine umfassende Einführung.

Der Ansatz, auf Grundlage einer erweiterbaren Plattform domänenspezifische Sprachen zu entwickeln und mit diesen die Lösungen zu erstellen, wurde von Völter später als "Generic Tools Specific Languages" bezeichnet. [Völ14] Älter und auf die klassische Entwicklung ausgerichtet ist Martin Fowlers Bezeichnung "Internal Reprogrammability" [Fo13].

In einer ersten Aufteilung des Projekts wurden zwei große Domänen zu unterscheiden:

- die Domäne der Erstellung ausführbarer Spezifikationen
- die Domäne des kundenspezifischen Projekts

Mit einer Entwurfsentscheidung wurde diesem Rechnung getragen und eine Trennung in zwei technische Entwicklungsprojekte vorgenommen: eines für wiederverwendbare Sprachen der Domäne der ausführbaren Spezifikationen und eines für projektspezifische Sprachen.

In bisherigen Projekten wurden vor allem mit zwei Formen der formalen Notation von Anforderungen Erfahrungen gesammelt: Gherkin [Gh14] und Use Case Formulare.

Zu Gherkin heißt es auf der Cucumber-Homepage "Cucumber's plain text DSL (Gherkin) somehow came out from the Agile community, mostly based on distillations made

by Dan North, Chris Matts, Liz Keogh, David Chelimsky and dozens of people on the RSpec and Cucumber mailing lists. And me [Aslak Hellesøy].“ [Cu14]

Die Use Case Formulare hingegen wurden intern auf Basis von [Oe06] und [RS07] entwickelt. Mit ihren Metadatenfeldern war im Textverarbeitungsprogramm eine rudimentäre Verwaltung möglich, wenn die Zahl der Use Cases gering war.

Um eine Entscheidung treffen zu können, wurden von beiden Formen Prototypen erstellt. Abbildung 1 zeigt den Gherkin-Prototyp. Die Anforderung wird in Gherkin “Feature” genannt, die Akzeptanztests als “Scenarios” oder “Scenario Outlines” definiert. Ein Feature hat einen Namen (“Serve coffee”) und eine Beschreibung, traditionell in der Form “In order/As a/I Must/Should” oder “As a/I want/So that”. Optional kann ein “Background” angegeben werden. Das Scenario hat auch einen Namen und folgt der bei Tests üblichen Form Arrange-Act-Assert, die bei Gherkin “given”, “when” und “then” genannt werden. Ein Scenario Outline ist ein Meta-Scenario, welches durch die Angabe von Beispielen konkretisiert wird. Im Beispiel werden die Variablen “start”, “eat” und “left” durch die Tabelle mit Werten befüllt. Es entstehen so viele Szenarien wie die Tabelle Zeilen hat. Features, Scenarios und Scenario Outlines können mit “Tags” versehen werden (“coffee”, “important”).

```
@coffee
Feature Serve coffee
  In order to earn money
  As a Customer
  I Must/Should be able to buy coffee at all times

  Background:
    given a global administrator named "Greg" +
    and a blog named "Greg's anti-tax rants" +
    and a customer named "Dr. Bill" +
    and a blog named "Expensive Therapy" owned by "Dr. Bill" +

  @important
  Scenario: Buy coffee
    given there are 1 coffees left in the machine +
    and   I have deposited 1$                     +
    when  I press the coffee button               +
    then  I should be served a coffee             +

  <<add a tag>>
  Scenario Outline: Eat cucumbers
    given there are <start> cucumbers +
    when I eat <eat> cucumbers +
    then I should have <left> cucumbers +

    Examples:
```

	start	eat	left
normal	5	2	3
null	0	0	0

Abbildung 1: Das Beispiel der Cucumber-Webseite in Gherkin

In Abbildung 2 ist der Prototyp des Use Case Formulars zu sehen. Nur ausgewählte Felder wurden implementiert. Die “ID” wird automatisch als laufende Nummer ermittelt. Ein Use Case hat einen Namen und ein Ziel (“Goal”). Der Akteur (“Actor”) ist eine Rolle aus einer separaten Rollendefinition. Datum (“Date”), Autor (“Author”) und Version sind manuell zu pflegende Informationsfelder. Oben sind die Testszenarien (“Scenarios”) zusammengefaltet dargestellt. Man sieht, dass zwei Szenarien existieren. Zur Bearbeitung werden die Testszenarien aufgefaltet (auf Grund der Größe im Paper nicht darstellbar).

Use Case Diagramme lassen sich automatisch erstellen, indem eine neue Projektion des AST zu UML-Use Case Diagrammen erstellt wird.

```
ID:        1
Name:      Anonymes Surfen
Goal:      Auf der Webseite kann mit einem Browser navigiert werden.
Actor:     Anonymer Webnutzer
Scenarios: 2 Test Scenarios
Date:      09.12.2012
Author:    Jens Nerche
Version:   1
```

Abbildung 2: Notation als Use Case Formular

Im Projekt gab es keinen Stakeholder, der sich für die im Use Case Formular erfassten Metadaten wie Nummer, Datum, Autor oder Version interessierte. Hingegen äußerten Vertreter der Fachabteilung Featurewünsche mit erklärenden Prosatexten. Damit wurde die Entscheidung für Gherkin getroffen, da die Erläuterungen der Fachabteilung leichter in den Gherkin-Prototyp mit Testszenarien darstellbar waren.

Die textuellen Erklärungen der Fachabteilung sollten direkt an die formalen Testdefinitionen in den Scenarios geschrieben werden können, um Medienbrüche zu vermeiden und zusammen gehörende Fakten lokal zusammenhängend zu notieren. Um dies zu ermöglichen, wurde eine Sprache "prose" erstellt, die einfache Textformatierungen erlaubt: kursiv, fett, verschiedene Schriftgrößen, Überschriften sowie Vorder- und Hintergrundfarben.

Die Beispieldokmente waren in diesem EAI-Projekt XML-Nachrichten. Je nach Geschäftsregel war nur ein kleiner Teil, also ein spezieller Attributwert oder ein XML-Tag, relevant. Eine Spracherweiterung der bei MPS mitgelieferten XML-Sprache trug dem Rechnung, indem die relevanten Teile eingefärbt und ein Verweis auf das dazugehörige Testszenario gesetzt werden konnte. Das Testszenario verlinkt auch die XML-Datei, so dass in dieser 1:1-Relation eine Navigation in beide Richtungen erfolgen kann – dargestellt in Abbildung 3.

```
ccemailtouserid="no" level="2"
(--> die Mail des Kunden beinhaltet eine Tabelle mit mehreren Invoices )
```

Abbildung 3: Attribute eines XML-Elements mit navigierbarem Verweis auf ein Testszenario

Hier ist die Besonderheit, dass im Attribut "level" jetzt eine "2" steht. Der Name des Testszenarios ("die Mail des...") wird in die XML-Datei projiziert und nicht kopiert. Dies ist ein Beispiel dafür, wie durch die Vermeidung von Medienbrüchen und die Verwaltung aller Artefakte in einem Programm die Refactoringsicherheit gegeben ist und Inkonsistenzen vermieden werden: wird der Szenarioname angepasst, werden alle Verweise darauf auch automatisch mit geändert, da der Text nicht kopiert, sondern an die entsprechenden Stellen hineinprojiziert wird.

Die Testszenarien können hierarchisch organisiert werden. Das Feature ist als Kompositum ausgelegt, kann also als Kindelemente auch weitere Features enthalten.

Bezüglich der Filterbarkeit von Testszenarien wurde die Entwurfsentscheidung getroffen, dass vorerst die Filterung nach Tags genügt. Jedes Feature kann eine Filterung bzgl. der Tags vornehmen. Zur Ausführung kommen dann nur die Szenarien, die eines der gewünschten Tags tragen.

Abbildung 4 zeigt weitere Funktionalitäten:

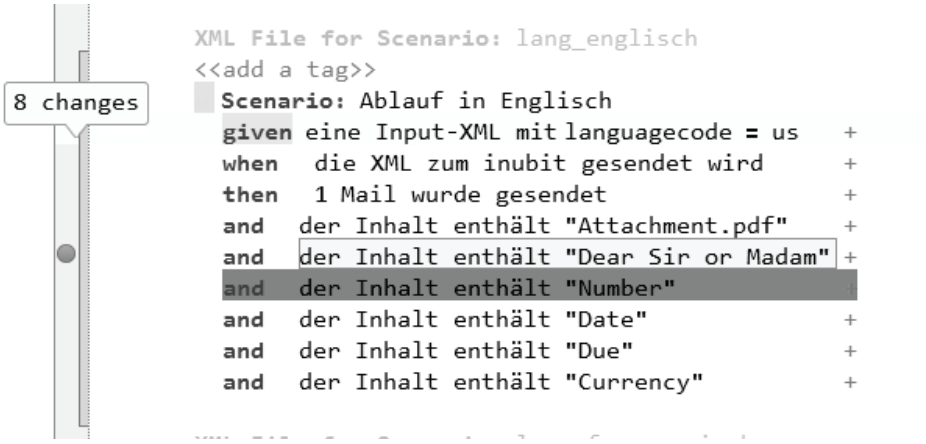

Abbildung 4: Weitere Funktionalitäten

Der Implementierungsfortschritt ist als farbliches Viereck vor dem Wort “Scenario” codiert für: “unbearbeitet”, “in Arbeit” und “fertig”. Farbig hinterlegt ist ein “Offener Punkt” der Zeile 'der Inhalt enhält “Number”'. Das sind Details, die noch mit dem Kunden zu klären sind. Die VCS-Integration ist an der Blase “8 changes” erkennbar. Über den Balken an der linken Seite kann man sich auch die Änderungsdetails anzeigen lassen und sie wieder rückgängig machen. Der Punkt links daneben ist ein Breakpoint und deutet die Debugger-Funktion an. Dieser funktioniert wie von einer IDE gewohnt. Der Programmablauf stoppt in der Zeile über dem Offenen Punkt und betrifft den eingerahmten Bereich mit “Dear Sir or Madam”.

Die oben erwähnte Verknüpfung zu einem XML-Dokument ist hellgrau über dem Szenario kenntlich gemacht.

Für die Fehlervermeidung schon bei der Eingabe wurden eine Reihe von zusätzlichen Beschränkungen und Quick Fixes (IDE-Hilfen zur einfachen Behebung von Fehlern) implementiert. Beispiele sind:

- für den Testfall ist keine Beispiel-XML definiert
- Namen für Features und Scenarios dürfen nicht doppelt vergeben werden

Die Testszenarien sind einzeln, alle gefilterten, alle des Dokuments, alle des Modells inkl. Untermodelle oder des gesamten Projekts ausführbar. Die Ausführung geschieht wie von einer IDE gewohnt: per Kontextmenü im Testszenario, im Feature, im Editor, im Projekt-Toolwindow oder als Run-Konfiguration. Nach der Ausführung wird der von automatischen Tests bekannte grüne oder rote Balken angezeigt. Ein Testreport zeigt die Übersicht der ausgeführten Tests, den Erfolg und die Ausführungszeit. Tritt während der Testausführung eine Exception auf, kann der Stacktraceparser die dazugehörende Zeile im Szenario ermitteln und einen Link zur Navigation in den Stacktrace einfügen.

Zwei Tags haben eine besondere Semantik bekommen: Ignore und Duplicate. Wenn ein Scenario damit annotiert ist, wird es bei der Testausführung ausgeschlossen.

Während der Erstellung der Lösung wurde beobachtet, dass die Arbeit mit der Kombination aus (Extensible) Language Workbench und domänenspezifischen Sprachen sehr produktiv ist. Kleine Helfer für den Alltag sind schnell erzeugt:

- Statistiken zeigen einen groben Überblick über den Projektstand
- Glossar für Fachtermini: für Begriffe aus der Domäne, die einer Erklärung für den Softwareentwickler bedürfen, wurde eine Verknüpfung zwischen dem Begriff und der Erklärung erstellt, die bidirektional navigierbar ist
- Todo-Liste: eine einfache Liste mit anstehenden Aufgaben, mit dem Vorteil der Tool- und Lösungsintegration gegenüber einer einfachen Textdatei
- Preview für Apache FOP [AF14]: in dem Projekt sollten aus per XML gelieferten Daten und einer Vorlage per Formatting Objects [Fo06] PDF-Dateien erzeugt werden. Der manuelle Turnaround zwischen Vorlage bearbeiten, PDF erzeugen, öffnen und beurteilen ist bei vielen Änderungen zeitaufwändig. Ein MPS-Plugin, welches eine Echtzeitvorschau auf den gerade geöffneten Editor bietet, erhöhte die Entwicklungsgeschwindigkeit erheblich.

Neben den wiederverwendbaren Sprachen wurde noch die projektspezifische Sprache für das konkrete EAI-Projekt entwickelt. Sie enthält unter anderem die Details zur Ansteuerung des EAI-Servers sowie den Generator für den projektspezifischen Code. Es wurden Testdoubles für einen FTP-Server und einen Mail-Server erstellt, Routingen zur Prüfung des entfernten Dateisystems erstellt sowie die Möglichkeit geschaffen, die EAI-Server-Konfiguration in der projektspezifischen Sprache zu schreiben und automatisiert zu installieren.

5 Ergebnisse und Erfahrungen

Die Entscheidung für die technische Basis einer Language Workbench mit projektionalem Editor war sehr gut. Das Hauptziel, die Stärken einer Textverarbeitung, eines RE&M-Tools und einer IDE zu verbinden, wurde erreicht. Der Funktionsumfang ist zwar weit von dem der spezialisierten Werkzeuge entfernt, jedoch ist mit relativ gerin-

gem Aufwand und schmalem Budget in der Nebenentwicklung ein mächtiges Werkzeug entstanden, welches genau auf den benötigten Funktionsumfang zugeschnitten ist.

Für dieses Projekt mussten letztendlich nur vier Sprachen entwickelt werden. Diese werden mit ihren Zuständigkeiten hier zusammengefasst:

- Gherkin: erweitert die bei MPS mitgelieferte Sprache "unitTest" und implementiert die vorhandenen DSL Gherkin [Cu14] in MPS.
- XML-Erweiterung: erweitert die bei MPS mitgelieferte Sprache "XML" um die Hervorhebung der Besonderheiten für ein Scenario und ermöglicht die Navigation zwischen Scenario und XML-Beispiel
- Prose: erweitert eine von Sascha Lisson bereitgestellte Sprache "richtext" [Li14] und ermöglicht es, formatierten Freitext einzugeben
- Projektspezifische Sprache: erweitert sowohl Gherkin und die XML-Erweiterung als auch die bei MPS mitgelieferte Sprache "baseLanguage" (eine Java-Implementierung, die aus Markenrechtsgründen nicht so heißen darf), kapselt die projektspezifischen Aspekte und stellt den Generator zur Verfügung

Wie erwartet wuchs die Anzahl der Geschäftsregeln und Ausnahmen im Projektverlauf schnell an. Dabei waren die automatisierten Tests sehr hilfreich, insbesondere die Integration der XML-Beispieldokumente. In den Statustelefonaten konnte über Anforderungen, den Umsetzungsstand usw. sofort Auskunft gegeben werden, offene Punkt waren leicht identifizierbar und konnten schnell geklärt werden. Durch die textuelle Beschreibung ist jederzeit der Inhalt klar. Die durch die Verlinkung und Projektionen entstandene Refactoringsicherheit über mehrere Artefakte hinweg rentierte sich in Folge der vielen Änderungen schnell.

Die Bedienung eines Projektionalen Editors ist gewöhnungsbedürftig, geht dann aber flott von der Hand. Sehr wichtig sind Mechanismen, die die Usability erhöhen und das Arbeiten flüssig und erwartungskonform ermöglichen. Damit konnten dann die Testfälle schneller als mit einer herkömmlichen IDE programmiert werden, insbesondere bei Verwendung von Scenario Outlines. Bei diesen entstehen die Testfälle ja durch ein einfaches Hinzufügen einer neuen Zeile in die Testwertetabelle.

Weil die Eingabehilfen kontextspezifisch nur die passenden Möglichkeiten anbieten, ist das Programmieren oder Modellieren der Testfälle weniger fehleranfällig. Leicht zu erstellende projektspezifische "Constraints" und "Checking Rules", wenn möglich mit passenden "Quick Fixes", senken die Fehlerrate bei der Eingabe noch weiter. Weil weniger Fehler gesucht und behoben werden müssen, wird die Testautomatisierung zusätzlich beschleunigt.

Neben den Vorteilen sind auch Nachteile zu nennen:

- Anfangsaufwand ist höher: statt vorhandener Lösungen mussten die benötigten Sprachen erst entwickelt werden. Existieren die benötigten Sprachen aber schon, verschwindet dieser Nachteil.

- Für den Entwickler kommt jetzt zu dem klassischen Softwareengineering noch die Aufgabe des Language Engineerings. Das setzt ein größeres Wissens- und Erfahrungsspektrum voraus, welches aktuell noch wenig verbreitet ist. Es ist schwieriger, für das Team Mitarbeiter zu finden.
- Trotz vieler Verbesserung der Bedienbarkeit Projektionaler Editoren ist der Arbeitsfluss noch nicht auf dem Niveau der neuesten IDEs.
- Sprachen und in diesen entwickelte Lösungen sind zwischen aktuellen Language Workbenches nicht austauschbar. Wenn man sich für ein Produkt entscheidet, müssen alle Teammitglieder dieses benutzen.

Die Spezifikation wurde stets auf aktuellem Stand gehalten, weil sie als ein Projektergebnisartefakt mit ausgeliefert wurde. Jedoch wird sie im Nachhinein vom Kunden lt. eigener Aussage nicht mehr gelesen. Da funktionierende Software vorhanden ist, besteht dafür offenbar kein Grund. Dies wird auch durch das “Agile Manifesto” gestützt: "Working software over comprehensive documentation" [AM14]. Für Ausführbare Spezifikationen waren in diesem Projekt also die Entwickler die Hauptzielgruppe, andere Stakeholder kamen nicht mit ihnen in Berührung. Das erklärt auch, warum die hier beschriebenen Erfahrungen ausschließlich aus der Sicht der Entwickler beschrieben werden.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Eine Language Workbench mit Projektionalem Editor ist aus technischer Sicht ein klarer Fortschritt gegenüber textbasierten Ansätzen aus folgenden Gründen:

- Die Sichtbarkeitssteuerung kann viel detaillierter erfolgen. Theoretisch kann für jedes einzelne Element angegeben werden, ob es angezeigt werden soll oder nicht. So sind einfach verschiedene Detailstufen definierbar.
- Die Projektion von Inhalten anderer Artefakte ermöglicht, das Prinzip der Vermeidung von Wiederholungen (Don't Repeat Yourself, DRY) weitreichender als bisher einzusetzen.
- Textformatierungen ermöglichen, dass die Kommentare und Erläuterungen übersichtlicher und deutlicher werden.
- Die Anordnung der Inhalte – untereinander, nebeneinander – kann so gewählt werden, dass eine maximale Übersichtlichkeit besteht.
- Mehrere Darstellungen desselben AST-Inhalts (Sichten) sind möglich.
- Nicht nur Text, sondern auch Tabellen, Bilder, mathematische Symbole usw. können verwendet werden.
- Bei MPS sind sogar alle Swing-Controls möglich. Würde die Spezifikation nicht nur wie in unserem Projekt von den Entwicklern, sondern auch von Sachbearbeitern erstellt und gepflegt, könnte die Benutzerschnittstelle weitgehend

an formularbasierte GUIs angepasst werden, wie sie in den Domänen üblich sind.

Üblicherweise arbeitet man im Projekt immer in mehreren Domänen gleichzeitig. Manche Domänen – besonders die technischen – sind universell. Für die universellen Domänen bereitstehende Sprachen können leicht verwendet, angepasst und erweitert werden, wenn wie im vorliegenden Fall die Language Workbench das Modularisieren und Kombinieren von Sprachen als ein Hauptfeature vorweist.

Ein dem Projekt angepasstes Werkzeug mit erstaunlich geringem Funktionsumfang ist schon ausreichend, um erhebliche Erleichterungen zu schaffen.

Im Projektalltag brachten diese Features den größten Produktivitätsgewinn:

- Tests einzeln und in Gruppen ausführen
- Übersicht und Organisation: Fortschritt und offene Punkte leicht erkennbar
- Integration aller Projektartefakte und Verweise zwischen ihnen
- Git-Integration ermöglicht, dass die Tests im selben Quelltextbaum geführt werden wie der Rest des Projektes; auch die Behebung von Konflikten bei gleichzeitiger Bearbeitung ist gut möglich.

6.2 Ausblick

Bei einer eventuellen Weiterentwicklung stehen diese Erweiterungen an erster Stelle:

- Weitere Reports und Dashboards mit Statistiken, Szenarien gruppiert nach Fortschritt usw.
- Localization: Schlüsselwörter und Text in mehreren Sprachen
- Kopplung mit Projekt- und Issuetrackingtools: moderne Werkzeuge wie Atlassian JIRA [AJ14] und JetBrains YouTrack [JT14] bieten Kanban- und Scrum-Boards und APIs zum Zugriff auf die Daten
- Notationen und Layout optimieren, mehr gestaltende Elemente wie Rahmen, Trennstriche, Symbole, Buttons, Checkboxen usw. verwenden, um das Design mehr an die formularbasierten Anwendungen anzugleichen, mit denen Sachbearbeiter üblicherweise arbeiten
- Datenbank-Integration: eine große Anzahl von Anforderungen in der Datenbank halten, nur relevante Teile laden
- Usability des Projektionalen Editors weiter erhöhen
- Anforderungsverwaltungsfunktion erweitern: selbst definierbare Suche über Standard- und selbst definierbare Meta- und Inhaltsdaten sowie weitere Sichten auf die Anforderungen, Akzeptanzkriterien und Testszenarien

Bisher wurden die beschriebenen Ausführbaren Spezifikationen nur in drei ähnlichen und relativ kleinen Projekten des selben Kunden verwendet. Daher sind weitere Evaluierungen nötig:

- Welche Art von Projekten hält die Systemspezifikation stets auf dem aktuellen Stand, definiert Akzeptanzkriterien und würde von automatisierten Akzeptanztests, die direkt in der Spezifikation notiert werden, profitieren? Insbesondere sollten große Projekte mit mehreren Management-Ebenen, verschiedenen Abteilungen etc. betrachtet werden.
- Der Einsatz anderer Notationsformen wie z.B. Use Case Formulare oder SOPHIST-Sprachschablonen [RS07] könnte die Lesbarkeit für Nicht-Techniker weiter erhöhen.
- Die Verwendbarkeit und Ergonomie im Vergleich zu Textverarbeitungsprogrammen aus Sicht des Projektmanagements, der Fachabteilung, des Anforderungsanalysten usw. kann zur Weiterentwicklung zum Erschließen einer breiteren Zielgruppe hilfreich sein.
- Ist die Freitexteingabe überhaupt sinnvoll oder sollte generell ein formularbasierter Ansatz bevorzugt werden? In [VWK14] und [Vö14b] wurde festgestellt, dass Business-Anwender eher Formulare mögen.
- Unter welchen Bedingungen sind Ausführbare Spezifikationen ein möglicher Ersatz für spezialisierte RE&M-Werkzeuge wie DOORS, Polarion etc.?
- Wenn die Lösung mit dem selben Werkzeug erstellt wird, ist die Verlinkung von Anforderung und Realisierung möglich, siehe [mb14]. Des Weiteren können das Datenmodell, Zustandsmaschinen oder formale Regeln sowohl von den Testspezifikationen als auch vom Produktivcode verwendet werden. Ist das sinnvoll oder sollte der Test generell eine unabhängige Implementierung sein?
- Wo liegen die Gemeinsamkeiten und Unterschiede zum Literate Programming [Kn92]?

Literaturverzeichnis

[Ad11] Adzic, Gojco: Specification by Example, Manning Publications Co., 2011
[AF14] Apache FOP, http://xmlgraphics.apache.org/fop/, 2014
[AJ14] Atlassian JIRA, https://de.atlassian.com/software/jira, 2014
[Al07] Alvestad, Kristian: Domain Specific Languages for Executable Specifications, Master Theses at Norwegian University of Science and Technology, 2007
[AL04] Apache Licence, Version 2.0, http://www.apache.org/licenses/LICENSE-2.0, 2004
[AM14] Manifesto for Agile Software Development, http://agilemanifesto.org/, 2014
[ASU86] Aho, Alfred v.; Sethi, Ravi; Ullman, Jeffrey D.: Compilers – Principles, Techniques, and Tools, Addison-Wesley, 1986
[Be03] Beck, Kent: Test-Driven Development by Example, Addison-Wesley, 2003
[Cu14] Cucumber Homepage, http://cukes.info, 2014
[ID14] IBM Rational DOORS, http://www-03.ibm.com/software/products/de/ratidoor, 2014
[FIT14] Fit: Framework for Integrated Test, http://fit.c2.com/, 2014
[FN14] FitNesse: The fully integrated standalone wiki and acceptance testing framework, http://www.fitnesse.org/, 2014

[Fo06] W3C: Formatting Objects als Teil der XSL-Spezifikation, http://www.w3.org/TR/xsl11/, 2006
[Fo13] Fowler, Martin: Internal Reprogrammability, http://martinfowler.com/bliki/ InternalReprogrammability.html, 2013
[Gh14] Gherkin, https://github.com/cucumber/cucumber/wiki/Gherkin, 2014
[Git14] Git distributed version control system, http://git-scm.com/, 2014
[HF10] Humble, Jez; Farley, David: Continuous Delivery, Addison-Wesley, 2010
[IP14] Intentional: The Intentional Platform, http://www.intentsoft.com/intentional-technology/, 2014
[Je14] Jenkins Continuous Integration Server, http://jenkins-ci.org/, 2014
[Jn14] Jnario Executable Specifications for Java, http://jnario.org/, 2014
[JT14] JetBrains YouTrack, https://www.jetbrains.com/youtrack/, 2014
[Kn92] Knuth, Donald: Literate Programming., Cambridge University Press, 1992
[Li14] Lisson, Sascha: Multiline text with embedded nodes for MPS , https://github.com/slisson/mps-richtext, 2014
[LWC14]Language Workbench Challenge, http://www.languageworkbenches.net/, 2014
[Ma09] Martin, Robert: Clean Code – A Handbook of Agile Software Craftsmanship, Prentice Hall, 2009
[mb14] mbeddr: Engineering the Future of Embedded Software, http://mbeddr.com/, 2014
[Me07] Meszaros, Gerard: xUnit Test Patterns – Refactoring Test Code, Addison-Wesley, 2007
[MGD10] Matyas, Steve, Glover, Andrew, Duvall, Paul M.: Continuous Integration: Improving Software Quality and Reducing Risk, Addison-Wesley, 2010
[MPS14] MPS Meta Programming System, https://www.jetbrains.com/mps/, 2014
[NS14] NatSpec, http://nat-spec.com/, 2014
[Oe06] Oestereich, Bernd: Analyse und Design mit UML 2.1, Oldenbourg 2006
[OO14] OpenOffice, http://www.openoffice.org/, 2014
[PS14] Polarion Software, http://www.polarion.com/, 2014
[RS07] Rupp, Chris, SOPHISTEN: Requirementsengineering und Management, Hanser 2007
[Sc14] Scala Language: http://scala-lang.org/, 2014
[Se14] SeleniumHQ Browser Automation: http://www.seleniumhq.org/, 2014
[ST14] ScalaTest, http://www.scalatest.org/, 2014
[Svn14] Subversion version control system, https://subversion.apache.org/, 2014
[VMX13] Die Beauftragte der Bundesregierung für Informationstechnik: V-Modell-XT, http://www.cio.bund.de/Web/DE/Architekturen-und-Standards/V-Modell-XT/Ueberblick/ueberblick_node.html, 2013
[Vö13] Völter, Markus et.al.: DSL Engineering – Designing, Implementing and Using Domain-Specific Languages, dslbook.org, 2013
[Vö14] Völter, Markus: Generic Tools, Specific Languages, CPI Wöhrmann Print Service, 2014
[Vö14b] Völter, Markus: Introducing Language-Oriented Business Applications, https://www.youtube.com/watch?v=8fbgFR8HPio&list=UUzX5fpY5TpvIQrEjqKpKGjQ, 2014
[VWK14] Völter, Markus, Warmer, Jos, Kolb, Bernd: On the Way to DSLs for Non-programmers, http://www.infoq.com/presentations/dsl-business-people, 2014
[Xt14] Xtext, https://eclipse.org/Xtext/, 2014