GEBIT Solutions

Die Experten für Java, Anwendungsentwicklung und Requirements Engineering

Der FitNesse Test für Software

Ein Java-basiertes Open Source Testframework

Dehla Sokenou, GEBIT Solution, 27.11.2008

Agenda

- Was ist FitNesse und was ist es nicht?
- Ein bisschen Wiki-Syntax
- Allgemeine Funktionsweise
- Tiefer ins Framework
- Erweiterungsmöglichkeiten des Wikis
- Unsere Erfahrungen
- Fazit
- Diskussion

Einleitung

- Was ist FitNesse?
  - Ein Softwaretestwerkzeug für den Akzeptanz-/Integrationstest
  - Ein Webserver
  - Ein Wiki
  - Ein Framework
    - Eigentlich zwei Frameworks:
      - Fit – Testtabellen und Auswertung
      - FitNesse – Wiki und Webserver
- Was leistet FitNesse *nicht*?
  - Kein Capture&Replay-Tool wie z.B. Abbot
  - Kein Tool für Unit Tests wie JUnit, sondern ein Tool zum Schreiben von Akzeptanz-/Integrationstests
  - Kein einfaches Tool (auch wenn das immer behauptet wird)
    - Aber man kann die Nutzung einfacher machen

Starten und Stoppen des FitNesse-Servers

- Starten und Stoppen des Servers
  - Kommandozeile, IDE, Ant
- Starten
  - Ausführen der Klasse **`fitnesse.FitNesse`**
    - -p <port number> → Port für den Webserver (default: 80)
    - -d <working directory> → FitNesse-Verzeichnis (default: Aufrufverzeichnis)
    - -r <page root directory> → FitNesse-Seiten-Root (default: FitNesseRoot)
    - -a {user:pwd | user-file-name} → Authentifizierung (kein default)
  - Angabe Classpath nötig, z.B. für Wiki-Erweiterungen!
- Stoppen
  - Ausführen der Klasse **`fitnesse.Shutdown`**
    - -h <hostname> → Host (default: localhost)
    - -p <port number> → Port des Webserver (default: 80)
    - -c <username> <password> → Authentifizierung (kein default)
  - Alternativ: http://<hostname>:<port number>/?responder=shutdown

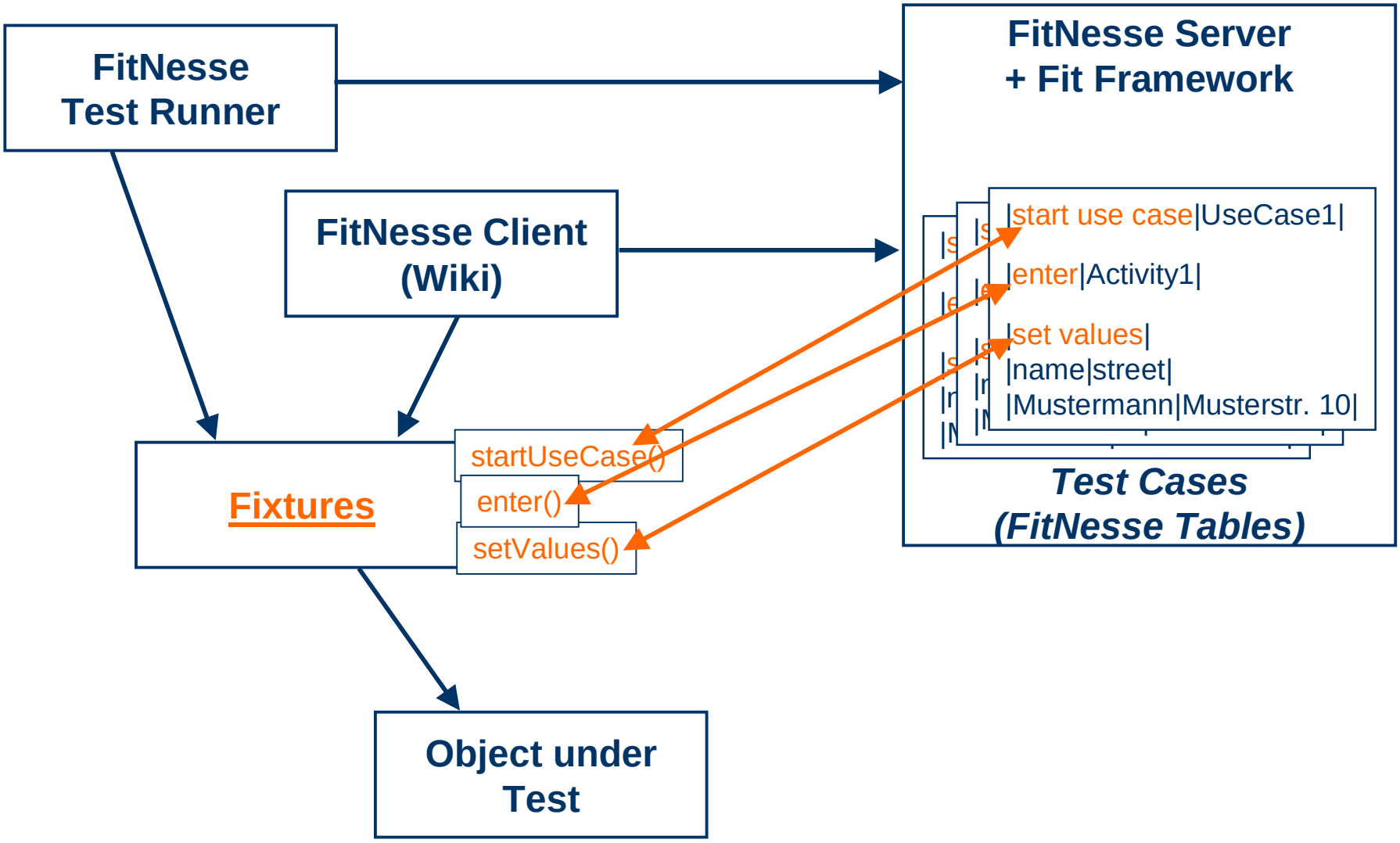
Ein bisschen Wiki-Syntax (1)

- Normale Wiki-Syntax wie Auszeichnung von Überschriften, Schriftgrad, Aufzählungen, ...
  - Beispiele: ---gestrichen---, '''bold'''
- Automatische CamelCase-Interpretation
  - Jedes Wort in CamelCase wird als Wiki-Seite interpretiert
  - Automatische Umsetzung in CamelCase (in Testtabellen)
    - Beispiel: "`press button`" wird interpretiert als "`pressButton`" (z.B. Methodenaufruf
  - Automatische Suche nach dem passenden Klassennamen (in Testtabellen)
    - Beispiel: "`my sample class`" wird zu "`de.gebit.MySampleClass`"
  - Auszeichnung nicht existierender Seiten mit `?` (Editieren)
    - Anlegen Root-Seiten einfach *CamelCase-Namen* eintippen
    - Anlegen Unterseite mit `^`*CamelCase-Name*
    - Anlegen auch direkt durch Eingabe nicht existierender Seite im Browser

Ein bisschen Wiki-Syntax (2)

- Spezielle Testsyntax
  - Seiten, deren Name mit "Test" beginnt, sind Testseiten
    - Direkte Auszeichung von Testseiten aber über Properties der Seite möglich
  - Testseiten können zu einer Testsuite zusammen gefasst werden
    - Auszeichnung per Properties
    - Testseiten als Subwiki oder via Include anderer Testseiten
    - Spezielle SuiteSetUp- und SuiteTearDown-Seiten (Achtung: geerbt im Subwiki!)
    - Achtung: Ausführungsreihenfolge von Tests wird durch alphabetische Sortierung definiert!
  - Definition des Classpath:
    - `!path C:/MySamples/samples`
    - `!path ${MYSAMPLES_HOME}/samples`
  - Definition von Testtemplates:
    - `!fixture MySamplesFixture`
  - Definition von Variablen:
    - `!define name {value}`

Funktionsweise



Beispiel 1

- Division einer Zahl

```
/**
 * Sample 1 class under test.
 */
public class Sample1Division {

    /**
     * Divide method under test.
     */
    public static double divide(double aNumerator, double aDenominator) {
         if (aDenominator != 0) {
            return aNumerator / aDenominator;
        } else {
            throw new IllegalArgumentException("denominator = 0!");
        }
    }
}
```

FitNesse-Tabellen im Wiki

- Jede Tabelle in einer Testseite wird von FitNesse zunächst als Test betrachtet
  - Unterdrückung der Auswertung mit `!- MyTable -!`
  - Unterdrückung geht auch für andere Wiki-Syntax, z.B. CamelCase-Auswertung
- Tabelle editieren

```
| tablecell1   |
| tablecell21 | tablecell22 |
```

- Angezeigte Tabelle im Wiki

tablecell1	
tablecell21	tablecell22

- Zu prüfender Output mit **?** markiert

Beispiel 1: FitNesse-Tabelle

```
|de.gebit.fitnesse.samples.Sample1|
|numerator|denominator|quotient?  |
|10       |2          |5          |
|12.6     |3          |4.2        |
|100      |4          |33         |
|100      |0          |error      |
```

de.gebit.fitnesse.samples.Sample1		
numerator	denominator	quotient?
10	2	5
12.6	3	4.2
100	4	33
100	0	error

Test schlägt fehl

de.gebit.fitnesse.samples.Sample1		
numerator	denominator	quotient?
10	2	5
12.6	3	4.2
100	4	33 *expected* 25.0 *actual*
100	0	error

Erwartung: Test liefert Exception (!), schlägt aber nicht fehl

FitNesse-Binding (Fixtures)

- Fixtures binden zu testenden Code an FitNesse-Tabellen
- Verschiedene Arten von Fixtures (knapp 200 Unterklassen von Fixture in FitNesse)
  - ColumnFixture – Input und erwarteter Output
    - Diverse Subklassen z.B.
      - RowFixture – Test von Mengen zurückgelieferter Werte, zusätzliche und fehlende Werte werden angezeigt
      - ParamRowFixture – Übergabe einer Targetklasse als Parameter
  - ActionFixture – Fixture zum Ausführen von Methoden
  - DoFixture – Alternative zu ActionFixture, erlaubt u.a. zusätzlich das einfache Hinzufügen weiterer Zellen in der Testergebnis-Tabelle
- Eigenes Binding überschreibt eine der Fixture-Klassen
  - Welche Fixture-Klasse als Oberklasse?
    - Natürlich möglich, einfach Fixture zu überschreiben, aber nicht sinnvoll
    - Möglichst wenig Methoden sollten überschrieben werden

Beispiel 1: Einfache FitNesse-Fixture

```
/**
 * Sample 1 FitNesse test (division).
 */
public class Sample1 extends ColumnFixture {

    public double numerator;
    public double denominator;

    /**
     * Simple fixture code for division test.
     */
    public double quotient() {
        return Sample1Division.divide(numerator, denominator);
    }
}
```

de.gebit.fitnesse.samples.Sample1		
numerator	denominator	quotient?

Standard-Input-Verhalten

Standard-Output-Verhalten
(inklusive Ergebnisprüfung)

von ColumnFixture!

Beispiel 2: Variablen in Tabellen

- Zustände über Fixtures hinweg merken (hier: ColumnFixtures)

de.gebit.fitnesse.samples.Sample1		
numerator	denominator	=quotient?
10	2	five
12.6	3	four.two

Ausgabe der Methode **`quotient`**

de.gebit.fitnesse.samples.Sample4		
quotient=	value	sum?
five	7	12
four.two	4.2	8.4

Setzen in Feld **`quotient`** der Fixture **`Sample4`**

de.gebit.fitnesse.samples.Sample1		
numerator	denominator	=quotient?
10	2	five = 5.0
12.6	3	four.two = 4.2

de.gebit.fitnesse.samples.Sample4		
quotient=	value	sum?
five = 5.0	7	12
four.two = 4.2	4.2	8.4

Tiefer ins Framework

- Einige wichtige Eigenschaften
  - 1 Testtabelle ≙ 1 Fixture-Klasse
    - Aber: Auswertung aller Tabellen einer Seite durch eine Klasse möglich
    - Für jede Tabelle muss eine Fixture zurückgeliefert werden
  - Tabellen sind Objekte, eigene Interpretation möglich (folgt)
  - Binding
    - Binden von Tabellenspalten an Funktionen, z.B. TypeAdapter oder Finden von Feldern und Methoden
    - Verschiedene vordefinierte Bindings mit vordefinierter Funktionalität, z.B.
      - SetBinding → Parsen der Zellen mit Hilfe des TypeAdapters
      - QueryBinding → Überprüfen der Zellen mit Hilfe des TypeAdapters
      - NullBinding → Ignorieren von Zellen
  - TypeAdapter mappen Tabellen auf Objekte
    - Einige vordefinierte TypeAdapter
    - Einführen eigener TypeAdapter durch Subklassen von TypeAdapter möglich

Beispiel 3: Fixtures aufeinander aufbauen

```
/**
 * Sample 2 FitNesse test (division with execute).
 */
public class Sample2 extends fitlibrary.DoFixture {

    /**
     * Execute division using Sample1
     * column fixture.
     *
     * @return Sample1 division fixture
     */
    public ColumnFixture execute() {
        return new Sample1();
    }
}
```

de.gebit.fitnesse.samples.Sample2

execute		
numerator	denominator	quotient?
10	2	5
12.6	3	4.2

execute		
numerator	denominator	quotient?
100	4	33
100	0	error

Beispiel 4: Binding und TypeAdapter

- ColumnFixture: Überschreiben von **`createBinding`**
  - Hier außerdem:

```
/** Adding hint to cell. */
public void wrong(Parse aCell, String aActual) {
    wrong(aCell);
    aCell.addToBody(" is not an integer!");
}
```

- Binding: In der Regel überschreiben von **`doCell`**
- TypeAdapter:

```
/** An adapter that parses BigDecimal values. */
public class BigDecimalAdapter extends TypeAdapter {
   public Object parse(String s) throws Exception {
      return new BigDecimal(s);
   }
   // eventuell müssen noch die Methoden get, set, invoke, equals
   // überschrieben werden
}
```

Beispiel 5: Binding und TypeAdapter

- FitNesse-Tabelle

de.gebit.fitnesse.samples.Sample3
big integer value?
278020082470987
abc

- Testresult-Tabelle

de.gebit.fitnesse.samples.Sample3
big integer value?
278020082470987
abc is not an integer!

Parsen von FitNesse-Tabellen

this	Fixture (id=18)
tables	Parse (id=20)
body	null
end	"</table>"
leader	"\n"
more	null
parts	Parse (id=26)
body	null
end	"</tr>"
leader	"\n"
more	Parse (id=34)
body	null
end	"</tr>"
leader	"\n"
more	Parse (id=44)
parts	Parse (id=45)
body	"big integer value?"
end	"</td>"
leader	""
more	null
parts	null
tag	"<td>"
trailer	"\r\n"
tag	"<tr>"
trailer	null
parts	Parse (id=35)
body	"de.gebit.fitnesse.samples.Sample3"
end	"</td>"
leader	""
more	null
parts	null
tag	"<td>"
trailer	"\r\n"
tag	"<tr>"
trailer	null
tag	"<table border="1" cellspacing="0">"
trailer	"\n"

Tabelle

1. Tabellenzeile

2. Tabellenzeile

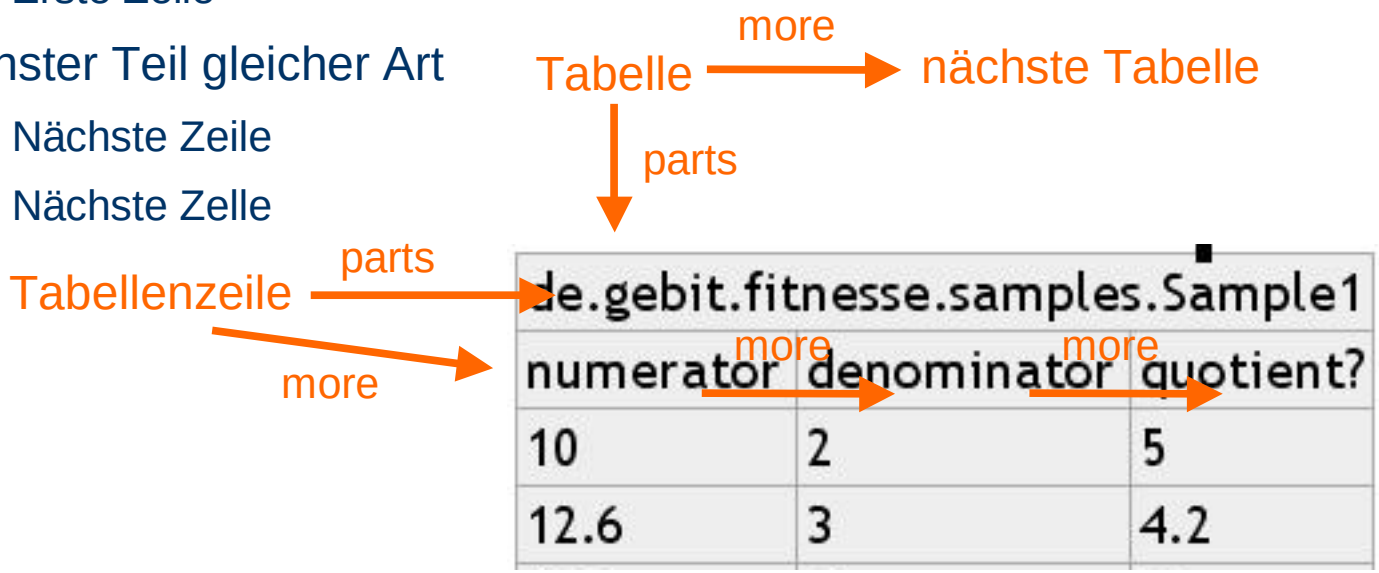
2. Zeile, 1. Zelle

1. Zeile, 1. Zelle

de.gebit.fitnesse.samples.Sample3
big integer value?
278020082470987
abc

Parsen von FitNesse-Tabellen

- Jede FitNesse-Tabelle sowie alle Zeilen und Zellen von Typ **`Parse`**
- Wichtige Felder:
  - body – Aktueller Zelleninhalt
    - Bei Tabelle und Zeile üblicherweise null
  - parts – Erster Teil
    - Bei Tabelle: Erste Zeile
    - Bei Zeile: Erste Zelle
  - more – Nächster Teil gleicher Art
    - Bei Zeile: Nächste Zeile
    - Bei Zelle: Nächste Zelle



Generischer als bisher

- Bisher: für jede Tabelle eigene Fixture-Klasse
  - Viel zu aufwendig
  - Vorgehen nur für Softwareentwickler geeignet
- Besser:
  - Generische Fixtures
    - Meist Subklassen von `fitlibrary.DoFixture`
  - Nutzung von Java-Reflections
  - Bindung an andere Frameworks
    - Die beste Variante, da dadurch meist eine sehr abstrakte Bindung erfolgen kann → Testfälle robuster gegen Änderungen im Sourcecode

Beispiel 6: Call durch Reflection

```
/**
 * Calls the given method on object of given class with given arguments.
 * Expected order of cell in given parse:
 * class name, method name, arguments.
 *
 * @param aClassNameParse the parse to use
 * @param aNumberOgfArgs the number of arguments
 * @return the result of method call
 * @throws Exception if method cannot be called
 */
private Object call(Parse aClassNameParse, int aNumberOgfArgs)
                                                    throws Exception {
   Class tempClass = Class.forName(aClassNameParse.body);
   Object tempObject = tempClass.newInstance();
   MethodTarget tempMethodTarget =
      MethodTarget.findSpecificMethod(
         aClassNameParse.at(1).body, aNumberOgfArgs, tempObject, this);
   return tempMethodTarget.invoke(aClassNameParse.at(2));
}
```

Beispiel 6: Call durch Reflection

- FitNesse-Tabelle

de.gebit.fitnesse.samples.Sample5					
testExpression	de.gebit.fitnesse.samples.SampleSum	add	2	4	=6
testExpression	de.gebit.fitnesse.samples.SampleSum	add	4.9	5.0	<10
testExpression	de.gebit.fitnesse.samples.SampleS	add	4.9	5.0	<10
testExpression	de.gebit.fitnesse.samples.SampleSum	ad	4.9	5.0	<10

Auswertung durch **`call`**

- Testresult-Tabelle

de.gebit.fitnesse.samples.Sample5					
testExpression	de.gebit.fitnesse.samples.SampleSum	add	2	4	=6
testExpression	de.gebit.fitnesse.samples.SampleSum	add	4.9	5.0	<10
testExpression	de.gebit.fitnesse.samples.SampleS - Class not found!	add	4.9	5.0	<10
testExpression	de.gebit.fitnesse.samples.SampleSum	ad - Method cannot be invoked	4.9	5.0	<10

Debuggen

- Mit Hilfe des FitNesse-Testrunners `fitnesse.runner.TestRunner`
  - Aufruf
    - `java fitnesse.runner.TestRunner [options] <host> <port number> <page name>`
  - Optionen
    - -v verbose → Ausgabe des Testfortschritts nach Standardout
    - -results <filename|'stdout'> → Testprotokoll als Plaintext
    - -html <filename|'stdout'> → HTML-Testprotokoll
    - -xml <filename|'stdout'> → XML-Testprotokoll

    } automatische Auswertung möglich

    - -debug → Ausgabe FitServer-Protokoll nach Standardout
    - -nopath → Kein Hinzufügen von Wiki-Classpath-Elementen
    - -suiteFilter <filter> → Ausführen nur von mit Filter gekennzeichneten Tests
  - nopath-Option wichtig, wenn Wiki-Classpath relativ und TestRunner nicht in FitNesse-Verzeichnis ausgeführt wird

Erweiterungsmöglichkeiten des Wiki (1)

- Erweiterungen werden in plugin.properties spezifiziert
- 6 Erweiterungsmöglichkeiten:
  1. Wiki Page
     - Erlaubt die eigene Erzeugung von Wiki-Pages, z.B. aus einer Datenbank
     - **`WikiPage = <fitnesse.wiki.WikiPage>`**
  2. Authenticator
     - Erlaubt die Implementierung eigener User-Authentifizierung
     - **`Authenticator = <fitnesse.authentication.Authenticator>`**
  3. Responders
     - Definition der Reaktion auf einen HTTP-Request:
       1. Aufruf durch http://myfitnesselocation/xyz?responder=key oder
       2. durch Nutzung des **`key`** in HTML-Formularen:
          `HtmlUtil.makeInputTag("hidden", "responder", key")`
     - **`Responders = <key:fitnesse.Responder> +`**

Erweiterungsmöglichkeiten des Wiki (2)

- 6 Erweiterungsmöglichkeiten (Fortsetzung):
  4. Content Filter
     - Definition valider Wiki-Seiten, Speicherung nur von nichtgefilterten Seiten
     - **`ContentFilter = <fitnesse.responders.editing.ContentFilter>`**
  5. WikiWidget
     - Eigenes Wiki-Look&Feel oder eigene Wiki-Funktionalität
     - **`WikiWidgets = <fitnesse.wikitext.WikiWidget>`**
  6. Html Page Factory
     - Erlaubt die Anpassung des Wiki-Look&Feel
     - **`HtmlPageFactory = <fitnesse.html.HtmlPageFactory>`**

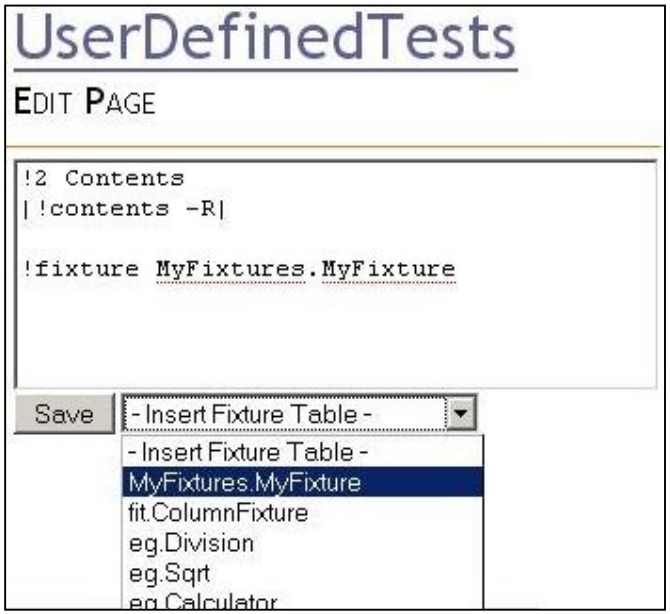
- Einfaches eigenes Wiki-Look&Feel auch möglich durch Anpassung der FitNesse-CSS-Dateien:
  - fitnesse.css (Anzeige) und
  - fitnesse_print.css (Drucken)

Existierende Erweiterungen

- BlogPlugin
  - Responder & WikiWidget, um FitNesse zu einem (rudimentären) Blog zu machen
- DatabasePlugin
  - Erlaubt das Speichern von FitNesse-Seiten in einer Datenbank statt im Dateisystem
- LinuxPamAuthenticator
  - Authentifizierung von FitNesse-Users per PAM
- WikiMailPlugin
  - Erlaubt das Hinzufügen neuer Seiten oder Seiteninhalte durch Senden einer Mail an den FitNesse-Server
- MavenPlugIn
  - Erlaubt das Ausführen von FitNesse-Tests aus Maven heraus
  - Im Gegensatz zu den anderen kein FitNesse, sondern ein Maven-Plugin

Eigene Testtemplates (1)

- Eigene Testtemplates werden in 3 Schritten erstellt
  1. Integration des Templates in die Wiki-Pages:
     - `!fixture MyFixtures.MyFixture`
  2. Definition des Responders als Antwort auf Template-Auswahl
     - Folgt auf der nächsten Folie
  3. Definition des Templates
     - Einfacher String mit FitNesse-Tabelle, z.B. `"|press|@BUTTON|\n"`



Eigene Testtemplates (2)

- Definition durch einen Responder
  - Woher kommt der **`key`**?
  - Alle Schlüssel sind definiert in der Klasse **`fitnesse.responders.ResponderFactory`**
  - Beispiele: **`edit`**, **`search`**, **`refactor`**, **`renameFile`**, ...
  - Bei der Definition eines Responders sollte man eine der existierenden Responderklassen überschreiben
  - Schlüssel in unserem Fall: **`tableWizard`**
  - Ergo: Zu überschreibende FitNesse-Klasse in unserem Fall: **`fitnesse.responders.editing.TableWizardResponder`**

Beispiel 7: Eigene Testtemplates

- Beispiel Subklasse von TableWizardResponder

```
public class MyTableWizardResponder extends TableWizardResponder {
[...]
   protected void initializeResponder(WikiPage aRoot, Request aRequest) {
      super.initializeResponder(aRoot, aRequest);
      fixtureName = ((String) aRequest.getInput("fixture")).trim();
      oldPageContent = (String) aRequest.getInput("text");
   }

   protected String createPageContent() throws Exception {
      if (isMyFixture(fixtureName)) {
         return oldPageContent + createMyFixtureTable();
      } else {
         return super.createPageContent();
      }
   }
[...]
}
```

Merken der relevanten Daten

Einfügen des Fixture-Templates in ursprüngliche Seite

Aufruf der ursprünglichen Funktion, wenn kein eigenes Fixture-Template

Unsere Erfahrungen

- Pilotprojekt: Java Enterprise Application
  - JBoss Applicationserver, Apache Webserver, JSF, mobile Geräte via WLAN eingebunden
  - ~ 2 Jahre Projektdauer, ~ 10 Personenjahre
    - ~ 10% = ~ 1 Personenjahr FitNesse Testaktivitäten
  - Modellbasiertierte Entwicklung mit dem TREND Framework:
    - ~ 100 Use-Cases mit zugeordneten Aktivitäten
  - ~ 2000 Java-Klassen (~ 120 Business-Klassen)
  - ~ 200000 LOC
  - ~ 100 Datenbanktabellen
  - In-Time fertiggestellt; seit Ende 2007 im produktiven Einsatz
- Inzwischen Einsatz von FitNesse in diversen Projekten

Unsere Erfahrungen

- Ca. 30 generische Fixtures im Pilotprojekt entwickelt
- Testsuiten
  - Haupttestsuite: 350 Tests mit 17450 Assertions
  - Zusätzliche Testsuite: 220 Tests mit 15000 Assertions
  - ~ 500 Anpassungen der Testcases (~ 2 Testcases/Tag)
- Coverage
  - Haupttestsuite: 100% Coverage der Use-Cases, Aktivitäten und deren Transitionen
- Gefundene Fehler
  - ~ 500 von 800 Bugs im Issue-Tracking-System
- Ziele
  - Haupttestsuite jeden Tag ohne Fehler laufenlassen
    - Regressiontests: Vermeidung von neuen Fehlern im getesteten Code
    - Nebeneffekt: Jederzeit ein stabiles Produkt an den Kunden lieferbar

Fazit

- Nachteile
  - Bessere Unterstützung der Testfallerzeugung nötig
    - Tabelle sind unhandlich, Kunden oft nicht bereit zur Einarbeitung
    - Erweiterung z.B. um Capturing von GUI-Events oder automatische Generierung z.B. aus Modellen
  - FitNesse Server muss gestartet werden, auch beim Ausführen von FitNesse **`TestRunner`**
  - Modularisierung, Hierarchie, Reihenfolge, Benennung der Testfälle
  - Integrationstest verschiedener Komponenten, z.B. Client/Server, nicht wirklich unterstützt
  - Extrem mangelhafte Dokumentation des Source Code

Fazit

- Vorteile
  - Offene Architektur, Open Source
  - Einfache Erweiterung für Fixtures & Wikis
  - Einfache Integration auch in andere Frameworks möglich, z.B. TREND Framework
  - Nicht fokussiert auf GUI oder Back-End-Testing (prinzipiell sogar Unit-Test möglich, aber mühsam)
  - FitNesse-Wiki erlaubt einheitlichen Zugriff auf Tests und Dokumentation
  - Nicht auf Java beschränkt
    - Ebenso verfügbar für C++, .NET, Perl, Python, Smalltalk, Ruby, …
  - Große Akzeptanz in der Community
    - Integration in diverse Produkte/Plattformen und andere Frameworks

Fragen?