Lebedev Alexander Borisovič

Entwurf und Implementierung eines Dashboards in Java zur Visualisierung von Testergebnissen

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuende Prüferin : Prof. Dr. -Ing. Karin Landenfeld
Zweitgutachter : Dipl.-Inf.(FH) Olaf Lange

Abgegeben am 12. März 2015

Lebedev Alexander Borisovič

Thema der Bachelorthesis

Entwurf und Implementierung eines Dashboards in Java zur Visualisierung von Testergebnissen

Stichworte

Dashboard, Test-Editor, Continuous Integration, Continuous Delivery, Qualitätssicherung, Java, Eclipse 4, RCP, SWT, Plug-In

Kurzzusammenfassung

Diese Arbeit beschäftigt sich mit der Konzeption und Realisierung eines Dashboards zur Visualisierung von Testergebnissen im Test-Editor, einem Tool für die Testautomatisierung. Nach der Behandlung der zugrunde liegenden Konzepte des Continuous Integration, des Continuous Delivery und der Qualitätssicherung wurden die Test-Editor-Umgebung und Lösungsmöglichkeiten analysiert. Dabei wurden die Anforderungen an das Dashboard mit Hilfe einer Benutzerumfrage ermittelt und spezifiziert. Nach der Festlegung des Lösungskonzeptes und Designs folgte die Umsetzung des Dashboards als Java-Plug-In integriert in den Test-Editor.

Lebedev Alexander Borisovič

Title of the paper

Design and implementation of the dashboard for test result visualization in Java

Keywords

Dashboard, Test-Editor, Continuous Integration, Continuous Delivery, Quality Assurance, Java, Eclipse 4, RCP, SWT, Plugin

Abstract

This paper deals with the conception and development of a dashboard visualizing test results in the Test-Editor, a test automatization tool. Upon addressing the fundamental concepts of continuous integration, continuous delivery, and quality assurance, an analysis of the Test-Editor environment, and possible development solutions was carried out. Thereby, the dashboard requirements were identified, and specified based on the outcome of a survey conducted among its future users. After defining the solution and design concept, the implementation of the dashboard as Java plugin integrated in the Test-Editor tool followed.

Danksagung

An dieser Stelle möchte ich mein Dank an das Akquinet AG Engineering Team und die Mitarbeiter von Signal Iduna, die mich fachlich und persönlich unterstützt haben, aussprechen. Vielen Dank für die gegebene Möglichkeit, bei Ihnen zu forschen und zu arbeiten.

Besonders möchte ich mich ganz herzlich bei meiner Betreuerin Frau Prof. Dr. Karin Landenfeld, bei meinem Betreuer Herrn Koray Bayraktar (Akquinet AG) und meinem Zweitprüfer Olaf Lange (Akquinet AG) bedanken.

Frau Prof. Dr. Landenfeld unterstützte mich durch ihre hilfreichen Anregungen und Ratschläge.

Durch seine fachliche Erfahrung und konstruktive Kritik verhalf Herr Bayraktar mir zu einer durchdachten These und einer erfolgreichen Realisierung des Dashboards. Vielen Dank für die Zeit und Mühen, die Sie in meine Arbeit investiert haben.

Herr Olaf Lange übernahm die Anfangsbetreuung und war immer für meine Fragen offen.

Zudem gilt mein Dank auch den Herren Felix Borchardt und Gerrit Borchardt für das Korrekturlesen meiner Arbeit.

Inhaltsverzeichnis

Danksagung 1
Abbildungsverzeichnis 4
Tabellenverzeichnis 5
Abkürzungsverzeichnis 5
1. Einleitung 7
1.1 Firmenprofil 7
1.2 Problemstellung und Entwicklungsvorgang 7
2. Softwarelieferprozess 9
2.1 Continuous Integration 9
2.2 Continuous Delivery 9
2.3 Softwarequalitätssicherung 12
3. Test-Editor 15
3.1 Test-Editor – allgemeine Beschreibung 15
3.2 Verfahren 16
3.2.1 Framework FitNesse 16
3.2.2 Test-Treiber (Fixture) 17
3.3 Hauptbedienelemente 17
4. Dashboard – Visualisierung der Testergebnisse 20
4.1 Dashboard – allgemeine Beschreibung 20
4.2 Software-Metriken 21
4.3 Erstellung der Anforderungen für das Dashboard 22
4.3.1 Umfrage zu Informationsmetriken und –anzeigen 24
4.4 Konzeption 28
4.4.1 Analyse der Test-Editor Umgebung 28
4.4.2 Lösungsansätze und Festlegung des Konzepts 31
4.4.3 Designentwurf und Funktionen 34
4.5 Realisierung 42
4.5.1 Technologien 42

4.5.2 Implementierung 45
5. Schlussbetrachtung 68
Anhangsverzeichnis 71
Literaturverzeichnis 72
Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit 75

Abbildungsverzeichnis

Abbildung 1: Von der Idee bis zum Kunden – Die Continuous Delivery Pipeline ist ein Teil dieser Kette (BIRK u. LUKAS 2014) 10
Abbildung 2: Prozess des Continuous Delivery (vgl. HUMBLE 2010) 11
Abbildung 3: Continuous Delivery Pipeline (BIRK u. LUKAS 2014) 11
Abbildung 4: Zusammenspiel Test-Editor >Test-Treiber >AUT (Test-Editor 2014b) 16
Abbildung 5: Test-Editor-Benutzeroberfläche (Test-Editor 2014a) 18
Abbildung 6: Test-Editor Testhistorie (Test-Editor 2014a) 22
Abbildung 7: FitNesse (Test-Editor 2014a S.47) 23
Abbildung 8: Umfrage zu Dashboard-Metriken 25
Abbildung 9: Testergebnisse in XML-Datei 30
Abbildung 10: FitNesse Test History 31
Abbildung 11: BIRTs verschiedene Dashboards Ansichten (BIRTWORLD 2015) 33
Abbildung 12: Erster Designentwurf 35
Abbildung 13: Endgültiges Design 35
Abbildung 14: Umschaltung der Ansichten 36
Abbildung 15: Tooltip für Bereichs Überschrift 36
Abbildung 16: Tooltip für Spalten Überschrift 36
Abbildung 17: Tabelle „Letzte Testläufe“ – Bereich 1 37
Abbildung 18: Icons: Testfall, Testsuite 38
Abbildung 19: Icons: ok, fail, warning 38
Abbildung 20: Projekte in Dropdown Menü 39
Abbildung 21: Tabelle „Alle Testläufe“: Testsuitelauf – Bereich 2 39
Abbildung 22: Tabelle „Alle Testläufe“: Testfalllauf – Bereich 2 40
Abbildung 23: Diagramm „Testdauertrend“ – Bereich 3 41
Abbildung 24: Tabelle „Fehler“ – Bereich 4 42
Abbildung 25: Das Eclipse 4.2 SDK besteht aus unterschiedlichen Komponenten (TEUFEL u. HEIMING 2012, S. 30) 43
Abbildung 26: Zusammenhänge der Klassen und Dashboard-Elementen 46
Abbildung 27: Fragment.e4xmi 47
Abbildung 28: Drop-Down-Menu für Projekte fragment.e4xmi 49

Tabellenverzeichnis

Tabelle 1: Umfrageergebnisse……………………………………………………………22

Abkürzungsverzeichnis

API……………………………………………………..Application Programming Interface

AUT……………………………………………………………….Application Under Test

AWT…………………………………………………………Abstract Windowing Toolkit

BIRT………………………………………...Business Intelligence and Reporting Tools

CD…………………………………………………………………Continuous Delivery

CI………………………………………………………………….Continuous Integration

CSS…………………………………………………………….Cascading Style Sheets

CSV………………………………………………………….Comma-separated values

DI…………………………………………………………………Dependency Injection

DOC………………………………………………………….Microsoft Word Document

DSL………………………………………………………….Domain Specific Language

EMF………………………………………………………..…Eclipse Modeling Framework

Fat………………………………………………………………….File Allocation Table

Fit…………………………………………………………Framework for Integrated Test

HTML…………………………………………………………Hypertext Markup Language

ID……………………………………………………………………………….. Identifier

IDE…………………………………………………...Integrated development environment

IoC…………………………………………………………………….Inversion of Control

JAR………………………………………………………………………….Java Archive

JDT……………………………………………………………..…Java Development Tools

JVM………………………………………………………………....Java Virtual Machine

OSGi…………………………………………………….Open Services Gateway initiative

PDE……………………………………………….…Plugin Development Environment

PDF……………………………………………………….Portable Document Format

PHP………………………………………………………….Hypertext Preprocessor

PPT………………………………………….Microsoft Powerpoint Präsentationsgrafik

PSC1, PSC2, PSC3……………………………………………....PartSashContainer

RCP………………………………………………………….…Rich Client Platform

Rest………………………………………………….…Representational State Transfer

SDK……………………………………………………….Software Development Kit

SOAP……………………………………………….…Simple Object Access protocol

SQL………………………………………………………Structured Query Language

SWT……………………………………………………….Standard Widget Toolkit

TF……………………………………………………………………………Testfallläufe

TS…………………………………………………………………………Testsuiteläufe

URI……………………………………………………….Uniform Resource Identifier

XLS…………………………………………………………..Tabelle (Microsoft Excel)

XML……………………………………………………Extensible Markup Language

1. Einleitung

1.1 Firmenprofil

Die Akquinet AG ist ein international tätiges IT-Beratungsunternehmen mit dem Hauptsitz in Hamburg. Mit 500 Spezialisten werden nationale und internationale IT-Projekte realisiert. Von der Entwicklung bis zum kompletten Hosting und Systemservice betreut die Akquinet AG zahlreiche Anwendungen auf höchstem Sicherheits- und Technologiestand. Die Akquinet AG realisiert Projekte gemeinsam mit den Kunden, um deren Unternehmenserfolg langfristig zu steigern. Im Bereich Testmanagement werden Möglichkeiten von automatisierten, quantitativen und qualitativen Tests angeboten. Aus den Projekten sind einige Open-Source-Produkte entstanden. Einer davon ist der Test-Editor, welcher zur Erfassung und automatischen Ausführung von Akzeptanztests dient (vgl. AKQUINET 2015). Der Test-Editor wird unter anderem von der Akquinet AG und der Signal Iduna entwickelt. Die Anwendung Test-Editor stellt eine intuitiv zu bedienende Oberfläche bereit, so dass Testfälle auch ohne Entwickler-Know-how erfasst werden können. Als Unterbau wird das Open-Source Framework FitNesse genutzt.

1.2 Problemstellung und Entwicklungsvorgang

Die zeitnahe Auslieferung von neuen Features einer Software erhöht den Mehrwert für den Endbenutzer. Damit die Qualität der Software abgesichert werden kann und dennoch in kurzen Zyklen ein Release gebaut werden kann, wird das Konzept des Continuous Delivery (CD) in die Projekte eingebracht. Dabei werden die Schritte für das Erzeugen eines Releases und seiner Qualitätssicherung automatisiert. Im Rahmen der Qualitätssicherung ist die Automation von Akzeptanztests essentiell.

Generell ist es sinnvoll nach einem Vier-Augen-Prinzip sowohl den Entwickler als auch den Experten der fachlichen Domäne in den Continuous-Delivery-Prozess zu integrieren. Dabei werden Rückmeldungen der kontinuierlichen Ausführung der Tests benötigt, die entsprechend aufbereitet werden müssen. Idealerweise sollten diese Informationen nahe an den Testfällen im Test-Editor in einem Dashboard, welches einen Überblick über die Testergebnisse visuell darstellen sollte, zur Analyse angezeigt werden. Das angestrebte Ergebnis dieser Arbeit ist die Realisierung solch eines Dashboards.

Die vorliegende Arbeit besteht aus zwei Teilen. In dem ersten Teil (Kapitel 2 bis 3) wird eine wissenschaftliche Grundlage geschaffen. Generell geht es dabei um die Qualitätssicherungsprozesse im Kontext agiler Softwareentwicklung. Es ist wichtig zuerst die Hintergrundverfahren wie Continuous Delivery und deren Bestandteil Continuous Integration sowie die Konzepte und Basisverfahren vom Test-Editor zu erläutern.

Der zweite, praktische Teil (Kapitel 4) beinhaltet die Definition der Anforderungen, Konzipierung und Entwicklung des Dashboards für die Darstellung der Testergebnisse in dem Test-Editor. Zunächst werden die möglichen Funktionen, die ein Dashboard zur Verfügung stellen kann, analysiert. Benötigte Funktionalitäten sowie Metriken werden mit Hilfe einer an der Zielgruppe orientierten Umfrage spezifiziert. Nach der Spezifikation der Anforderungen folgt eine Analyse der bereits vorhandenen Software-Metriken und für die Testergebnisse relevanten Daten. Im darauf folgenden Kapitel werden die Lösungsmöglichkeiten in Betracht gezogen und anschließend ein Designentwurf und Entwicklungskonzept des Dashboards festgelegt. In der Konzipierung werden die durch die Entwicklungsumgebung des Test-Editors gegebenen Techniken und priorisierten Nutzeranforderungen beachtet. Weiterhin wird die Implementierungsphase dargestellt, in welcher verschiedene Programmiertechniken verwendet werden, die in den zugehörigen Kapiteln erläutert werden. Schließlich folgt die Erläuterung des Implementierungsprozesses und es wird das Feedback von den Nutzern des Produktes präsentiert.

2. Softwarelieferprozess

„Im Kern geht es bei agiler Softwareentwicklung um möglichst häufige Rückkopplungsprozesse und zyklisches (iteratives) Vorgehen auf allen Ebenen: bei der Programmierung, im Team und beim Management“ (it-agile 2015a).

Im folgenden Kapitel werden die wichtigen Komponenten für die Unterstützung und Erweiterung des agilen Softwareentwicklungs- und Softwarelieferprozesses beschrieben. Dabei werden Continuous Integration, Continuous Delivery und die Qualitätssicherung als unverzichtbare Bestandteile der agilen Softwareproduktion in den Fokus gestellt.

2.1 Continuous Integration

Eines der größten Probleme, dem viele Entwicklungsteams mit mehreren Akteuren bei der Softwareherstellung gegenüberstehen, sind Integrationsfehler, welche schwer zu vermeiden bzw. zu beseitigen sind (vgl. SCHLUFF u. FEUSTEL 2012). Laut Fowler beschäftigt sich Continuous Integration (CI) damit, die Software-Komponenten mehrerer Entwickler in ein Produkt zeitlich effizient und fortlaufend zusammenzubringen, mit dem Ziel, die Qualitätssicherung zu verbessern. Das System wird kontinuierlich, begleitend von automatisiert ablaufenden Tests, neu gebaut. Im Laufe jeder Iteration sind frühe Warnungen über die Integrations- und Kompatibilitätsprobleme effizient, ebenso frühzeitig werden Fehler durch Unit-Tests erkannt (vgl. FOWLER 2004). Dank früher Fehlererkennung in der Entwicklungsphase werden mehr Fehler und somit Extrakosten in der Produktion vermieden. Somit sorgt die Methode des Continuous Integration für eine synchronisierte Arbeit des Entwicklerteams, minimiert das Fehlerrisiko und ist kosteneffizient.

2.2 Continuous Delivery

Heutzutage steigt die Anzahl von Unternehmen rasant, die das klassische Phasenmodell zu einer iterativen Vorgehensweise umwandeln, bei der „auch Zwischenstände der entwickelten Software schon qualitätsgesichert […] an den Kunden ausgeliefert werden. Die Kette aus Entwicklung, Qualitätssicherung und Lieferung wird also bereits mehrfach im Entwicklungsprozess der Software durchlaufen“ (BIRK u. LUKAS 2014) (Abbildung 1).

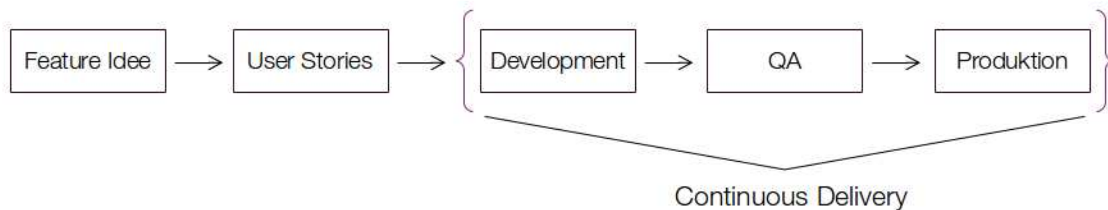


Abbildung 1: Von der Idee bis zum Kunden – Die Continuous Delivery Pipeline ist ein Teil dieser Kette (BIRK u. LUKAS 2014)

Laut Birk und Lukas zeichnet das klassische Phasenmodell der Softwareentwicklung für jedes Softwarerelease einen einmaligen Zyklus von drei Schritten aus: Entwicklung, Qualitätssicherung und Auslieferung. Jede Entwicklungsphase wird von der Qualitätssicherungsphase gefolgt. Die letztere umfasst die Untersuchung aller festgesetzten Änderungen auf die Erfüllung der Anforderungen und gegebenenfalls die Optimierung dieser für das nächste Release. Der aus der Testphase resultierende Softwarestand wird in der anschließenden Phase ausgeliefert (vgl. BIRK u. LUKAS 2014). So „wird bei Continuous Delivery nach der Fertigstellung eines Features dieses sofort im Rahmen der fortlaufenden Auslieferung genutzt" (KAMANN u. WENDT 2012).

Neben CI ist Continuous Delivery ein weiteres iteratives Vorgehen zur Optimierung des Softwarelieferprozesses. Nach dem grundlegenden Prinzip von Continuous Delivery wird die Softwareproduktion so gestaltet, „dass kontinuierlich neue Funktionen erfolgreich in Betrieb genommen werden, um so schnell wie möglich genutzt werden zu können. Den Entwicklern gibt das ein schnelles Feedback" (KAMANN u. WENDT 2012). Die CD-Methode wurde von Jez Humble und Dave Farley konzipiert. Dafür wurde das CI-Fundament mit finalen für die Produktion wichtigen Phasen erweitert. Abbildung 2 zeigt den iterativen auf CI basierten CD-Prozess mit seinen Phasen, die später in diesem Kapitel genauer erläutert werden.

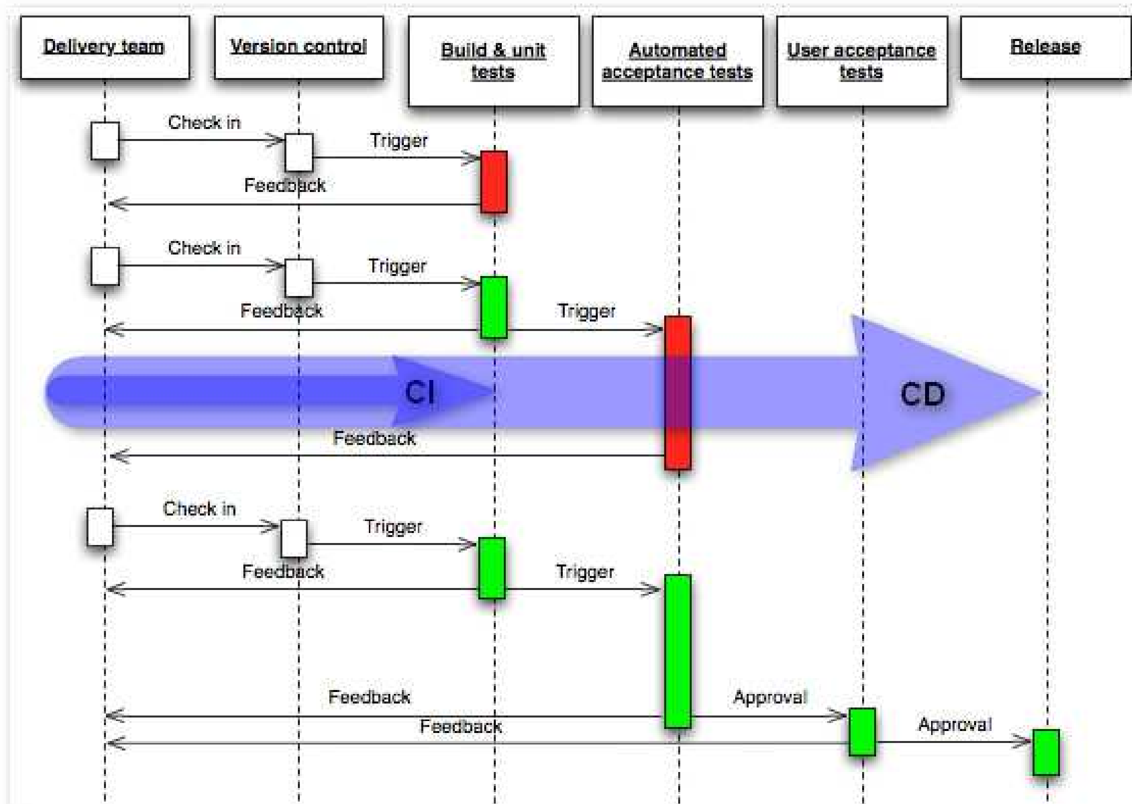


Abbildung 2: Prozess des Continuous Delivery (vgl. HUMBLE 2010)

Insbesondere bei der Software, die mit agilen Methoden entwickelt wird, bei denen ein Lieferzyklus oder Iteration innerhalb weniger Wochen erfolgen sollte, stößt man auf den mit Regression verbundenen Aufwand.

Die Flexibilität agiler Entwicklungsmethoden hat zur Folge, dass sich im Laufe der Softwareproduktion neue bzw. geänderte Anforderungen herauskristallisieren können. Der Entwicklungsumfang kann sich durch die iterative Vorgehensweise zum Vorteil von bereits vorhandenen Erfahrungen beeinflussen lassen (vgl. it-agile 2015b). Demzufolge wird der Testaufwand immer komplexer und soll sich der Flexibilität der Anforderungsdefinition anpassen. Continuous Delivery versucht dieses Problem durch die Testautomatisierung zu beheben.

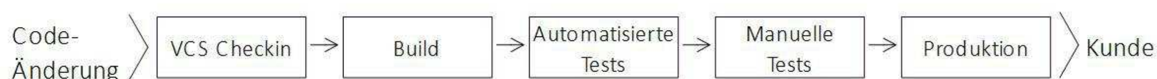


Abbildung 3: Continuous Delivery Pipeline (BIRK u. LUKAS 2014)

Wie Birk und Lukas beschreiben, wird die Qualitätssicherung durch eine Kombination von automatischen und manuellen Tests realisiert (Abbildung 3). In der Continuous Delivery Pipeline wird jeder Softwarestand mehrere Teststufen durchlaufen. Funktionale und nichtfunktionale Anforderungen werden durch unterschiedliche Tests mit hohem Automatisierungsgrad validiert. In der ersten Teststufe (Commit Stage), in der die Komponenten der Software gebaut werden, decken die Unit-Tests einzelne Komponenten isoliert in ihren Funktionen ab. Die nachfolgende Teststufe (Acceptance Test Stage) beinhaltet die Akzeptanztests, die die Übereinstimmung des Produktes mit den Anforderungen aus Sicht des Benutzers überprüft, sowie die Integrationstests, die das Zusammenspiel mehrerer Komponenten testen. Schließlich werden nichtfunktionale Anforderungen durch Performancetests oder statische Codeanalyse überprüft (vgl. BIRK u. LUKAS 2014b). Nicht alle Tests können automatisiert werden, so bleibt die manuelle Testphase nicht ausgeschlossen „Die Tests werden in mehreren Stufen organisiert und diese nacheinander für jeden Softwarestand ausgeführt. Nur wenn die eine Teststufe erfolgreich war, starten die Tests der nächsten Stufe überhaupt“ (BIRK u. LUKAS 2014a). Somit entsteht die sogenannte Continuous Delivery Pipeline.

Der oben beschriebene Prozess des Continuous Delivery (Abbildung 1) bietet jedem Beteiligten die Möglichkeit, schnell das Feedback zu jeder Software-Änderung zu erhalten (vgl. HUMBLE 2010). Dabei werden die wichtigsten Prinzipien des Continuous Delivery unterstützt: Jede Änderung an der Software wird getestet, und ein einziger Build wird zentralisiert gebaut. So werden die notwendigen Qualitätssicherungsschritte automatisiert und iterativ durchgeführt, wobei das Risikopotenzial und der Zeitaufwand gleichzeitig reduziert werden (vgl. BIRK u. LUKAS 2014).

2.3 Softwarequalitätssicherung

Es ist wichtig zu betonen, dass die Qualitätssicherung ein fest eingebundenes Glied in der CD-Pipeline ist. Die ISO/IEC 25000 (früher ISO 9126) definiert die Qualität der Software als „die Gesamtheit der Merkmale und Merkmalswerte eines Software-Produkts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen“ (BALZERT 1998, S. 257). Qualitätssicherung ist ein wichtiger Bestandteil des Prozesses von Continuous Delivery. Im Laufe der

Qualitätssicherung werden folgende Qualitätsmerkmale getestet: „Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit“ (FRANZ 2007, S. 20). Die Flexibilität und Komplexität der Anwendungen und das Produktangebot steigen und demzufolge werden Tests der Softwareprodukte immer komplexer und umfangreicher.

Wie Berner betont, sind Softwaretests die wichtigste qualitätssichernde Maßnahme zur Überprüfung der Qualität und Erfüllung von Anforderungen eines Softwaresystems. Vielen Entwicklungsteams unterlaufen immer wieder dieselben Fehler: Anforderungen und Softwaredesign werden nicht ausreichend auf Testbarkeit geprüft (vgl. BERNER 2008, S. 4). Demzufolge können die Tests zu spät starten und zu lange dauern. Das ist für den Prozess des Continuous Delivery und generell für die agile Entwicklung schädlich. Nicht weniger Problemen sehen sich auch die Auftraggeber gegenüber gestellt, deren Aufgabe es ist, die Akzeptanzkriterien zu prüfen (durch Akzeptanz- oder Abnahmetests). Die Funktionen, Belastbarkeit und Kompatibilität der Software wiederholend zu prüfen bedeutet einen hohen manuellen Aufwand, der längere Testphasen beansprucht. Infolgedessen können Verzögerungen in der Auslieferung und Nutzung der neuen Softwarefunktionen, Systemausfälle und Datenverluste auftreten, die für den Geschäftsprozess nicht zu vernachlässigbare finanzielle Verluste bedeuten können. „Je später ein Fehler entdeckt wird, desto aufwändiger ist seine Behebung“ (BERNER 2008, S. 4).

Um diese Probleme zu beseitigen, müssen Entwickler, Testteam und Auftraggeber die Verantwortung für die Qualitätssicherung der Software übernehmen. Im Bezug darauf „ist es notwendig, den Testprozess in den Vordergrund zu stellen“ (BERNER 2008, S. 5), für die Entwickler die Testbarkeit zu erhöhen, um einen hohen Grad an Testautomatisierung zu erreichen und dann überwiegend die automatisierten Testwerkzeuge zu nutzen. Bereits ab Beginn der Entwicklungsphase werden die Ergebnisse kontinuierlicher automatisierter Tests nicht nur zur rechtzeitigen Behebung von Softwarefehlern genutzt, sondern sichern zugleich den Entwicklungsprozess ab und erleichtern die Abnahmetests für den Auftraggeber.

Für die agile Softwareentwicklung ist „das sogenannte Test-Driven Development (TDD), in dem automatisierte Testfälle bereits vor der eigentlichen Funktion geschrieben werden“ (BERNER 2008, S. 5) eine sinnvolle Vorgehensweise, die eine

gewisse Flexibilität für die Entwickler ermöglicht. Es kann angepasst werden, „denn das Wichtigste dabei ist, dass am Ende der Code automatisch getestet wird" (BERNER 2008, S. 5). Die Vorteile eines automatisierten Tests liegen in der Möglichkeit mit geringerem Zeitaufwand zu jedem Zeitpunkt zugleich Performance- und Regressionstests durchzuführen. Daher ist es notwendig während des gesamten Entwicklungsprozesses kontinuierlich die Testautomatisierung einzusetzen (vgl. BERNER 2008, S. 5).

Die Abschlussphase der Qualitätssicherung in Continuous Delivery bilden die Akzeptanztests, „die der Auftraggeber in einer produktionsnahen Systemumgebung oder sogar in der Produktionsumgebung durchführt. Anhand wichtiger und „normaler" Geschäftsvorfälle werden Funktionalität, Benutzbarkeit und Effizienz aus Sicht des Auftraggebers geprüft" (FRANZ 2007, S. 271).

Für den Zweck der Akzeptanztests wurde von der Akquinet AG ein automatisiertes Testwerkzeug Test-Editor entwickelt. In dem folgenden Kapitel wird der Test-Editor genauer beschrieben.

3. Test-Editor

In diesem Kapitel werden die Funktionalitäten, Hintergrundtechniken und Hauptbedienelemente des Test-Editors genauer beschrieben.

3.1 Test-Editor – allgemeine Beschreibung

„Der Test-Editor ist ein Open-Source-Werkzeug zur Erfassung und automatischen Ausführung von Akzeptanztests [mit automatisierter Testauswertung und Fehleranalyse, A. L.]. Die Anwendung stellt eine intuitiv zu bedienende Oberfläche bereit" (Test-Editor 2014a, S. 3). Es können Oberflächen aus einer fachlichen Sichtweise beschrieben und getestet werden. Ein Wissen über die Implementierung der jeweiligen Anwendung ist dabei nicht notwendig. Als Unterbau (Backend) um die Struktur von Testfällen, Suiten etc. abzubilden und um Tests auszuführen wird das Open-Source Framework FitNesse genutzt (vgl. Test-Editor 2014a, S. 3). „Der *Test-Editor* an sich stellt lediglich ein optimiertes Benutzerinterface bereit" (Test-Editor 2014b, S. 6). „Der *Test-Editor* unterstützt verschiedene Typen von zu testenden Applikationen, z.B. Web-Anwendungen, Desktop-Anwendungen oder Webservices. In diesem Zusammenhang wird häufig der Begriff AUT (Application Under Test) genannt, er beschreibt das zu testende System" (Test-Editor 2014b, S. 6).

Als weitere Funktionalität ist Test-First möglich, d.h. die zu testende Anwendung muss für die Erfassung der Tests noch nicht existieren. Tests werden in einer selbst definierten Fachsprache (aka. DSL = Domain Specific Language) formuliert. Es werden folgende Architekturen unterstützt: Webanwendungen, Swing Fat-Client, SWT/RCP Fat-Client, Webservices (SOAP und Rest), Mainframe-Anwendungen (kostenpflichtiges Plug-In erforderlich) (vgl. Test-Editor 2014c).

Dank leichter CI-Integration bietet das Tool außerdem die Möglichkeit, in Verbindung mit Continuous Integration Werkzeugen (z. B. Jenkins/Hudson) Tests vollautomatisch auszuführen. Der Test-Editor nutzt das weit verbreitete Open-Source-Framework FitNesse als solide Test-Engine. Die Einfachheit des Test-Editors erfordert kein Entwickler-Know-How. Unter anderem sind benutzergeführte Testfall-Erfassung, automatische Eingabe-Validierung und Auto-Vervollständigung weitere Funktionen im Arsenal des Tools. (vgl. Test-Editor 2014c)

3.2 Verfahren

In diesem Unterkapitel werden die vom Test-Editor benutzen Verfahren (Abbildung 4) kurz erläutert.

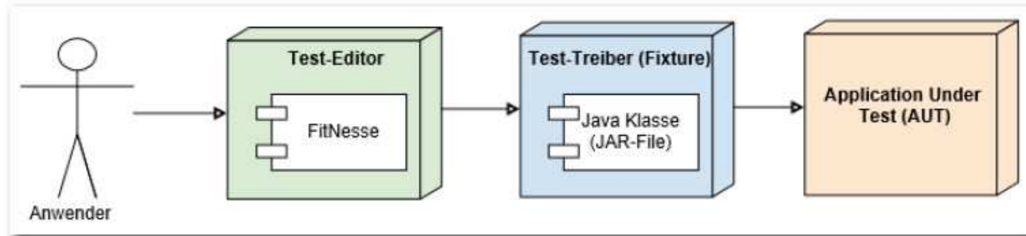


Abbildung 4: Zusammenspiel Test-Editor >Test-Treiber >AUT (Test-Editor 2014b)

3.2.1 Framework FitNesse

Das Framework FitNesse, basiert auf dem Framework Fit (Framework for Integrated Test), ist ein Open-Source-Framework für die Automatisierung von Akzeptanztests. Es wurde hauptsächlich von Robert C. Martin und Micah Martin entwickelt und wird zusätzlich zu den Eigenschaften von Fit als WiKi gehostet (vgl. FEATHERS 2011, S. 78).

Daten für die FitNesse-Tests werden in einfacher Form tabellarisch unter Nutzung eines eigenen Wiki-Systems erstellt. Die erstellten Tabellen beinhalten Ein- und Ausgabe-Parameter der Software. Im Gegensatz zur Eingabe werden in der Ausgabe-HTML die erfolgreich ausgeführten Zellen in der Tabelle grün und nicht erfolgreiche rot markiert. Zur Nutzung von FitNesse wird der Programmiercode für den Umgang mit den Tabellen angepasst. Tests kann man schnell und zu einem beliebigen Zeitpunkt einzeln oder in einer Suite kombiniert ausführen. FitNesse fördert das Test-Driven-Development und die Kommunikation zwischen Entwicklern und Designern. Tests können vor der Entwicklung definiert werden und gelten zuerst als nicht bestanden, folglich, um die Tests zu bestehen, werden die Funktionen nach und nach implementiert (vgl. FEATHERS 2011, S. 77). Akzeptanztests können plattform- und personalunabhängig voneinander, gemeinsam von mehreren Mitwirkenden wie Kunden, Testern, Entwicklern, Fachbereichsmitarbeitern usw. in einfacher Prosaform erstellt und getestet werden.

Fitnesse wurde in Java entwickelt; Das Framework kann aber Software unterstützen, die in Sprachen wie Java, C#/.NET, C, C++, Python, Ruby, PHP, Delphi und anderen implementiert wurde (vgl. MARTIN 2014, S. 205-206).

Bei der FitNesse-Anwendung werden die Code-Adapter-Klassen, auch Fixtures genannt, erstellt. Diese Test-Treiber werden aufgerufen und bilden eine Verbindung zwischen dem Wiki und dem zu testenden System (AUT) (vgl. OSHEROVE 2010, S. 263).

3.2.2 Test-Treiber (Fixture)

Zum Testen werden unterschiedliche Anwendungstypen vorliegen, demgemäß werden auch verschiedene Test-Treiber benötigt. Fixtures werden als JAR-Datei ausgeliefert, die eine oder mehrere Java-Klassen beinhaltet. Die Klassen beinhalten Plain Old Java Objects, die keine externen Abhängigkeiten aufweisen, und realisieren während der Test-Ausführung die automatische Fernsteuerung des AUT (z.B. „klicke auf Button", „gib in das Textfeld ein" usw.). Einige Test-Treiber werden mit dem Test-Editor in den zugehörigen Demo-Projekten ausgeliefert und können nach Bedarf mit eigenen spezifischen Treibern erweitert werden (vgl. Test-Editor 2014b, S. 6).

3.3 Hauptbedienelemente

Im Folgenden wird eine kurze Beschreibung für die Hauptbedienelemente des Test-Editors gegeben. Nach dem Start des Test-Editors gelangt man zur Hauptansicht. Vor der Entwicklung des Dashboards war es auch die einzige Ansicht mit allen Bedienelementen.

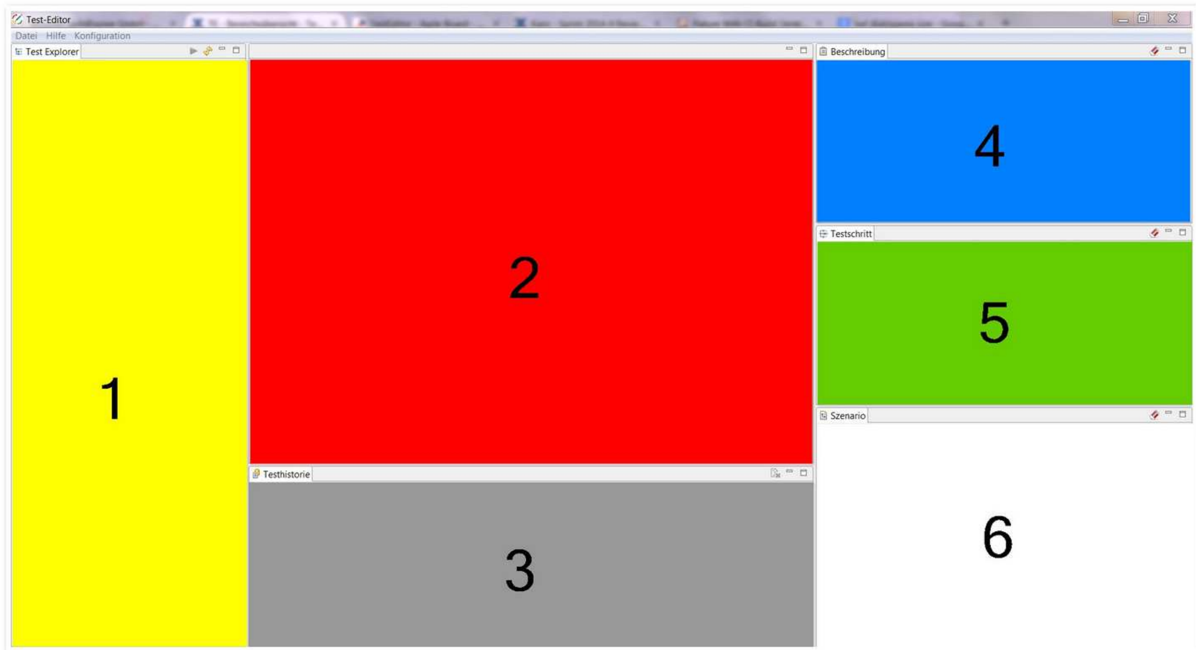


Abbildung 5: Test-Editor-Benutzeroberfläche (Test-Editor 2014a)

Der Test-Editor ist in sechs Bereiche aufgeteilt (Abbildung 5).

In dem „Explorer“ (Bereich 1) werden einzelne Projekte, Szenarien, Testfälle und Testsuiten in einer hierarchischen Struktur abgebildet. Im Test-Explorer befinden sich die Projekte in der obersten Ebene der Hierarchie und bestehen in der Regel aus mehreren Testsuiten, Testfällen und den Projekteinstellungen. Diese hierarchische Aufbaulogik ist vor allem für die spätere Implementierung der Datenauslesealgorithmen des Dashboards wichtig. Testsuiten werden zur Gruppierung von Testfällen beziehungsweise untergeordneten Test-Suiten verwendet und können zum Beispiel einen logischen Testabschnitt einer bestimmten Funktion des AUT oder die Testsammlung für die neuen Funktionen aus dem letzten Release definieren. Testfälle sind hierarchisch in der unteren Ebene angeordnet und beinhalten die Testabläufe (Testschritte) (vgl. Test-Editor 2014a, S. 5).

> Anschaulich beschreibt ein Testfall das Ausprobieren einer Software. Möchte man dies präzisieren, kommt man darauf, dass man drei Teilschritte erfassen muss: die Vorbedingungen, die Ausführung und die Nachbedingungen. Generelles Ziel der Testfallspezifikation muss es sein, den gleichen Test unter den gleichen Vorbedingungen Immer wieder durchzuführen und dann zu den gleichen Ergebnissen zu kommen (KLEUKER 2013, S. 25).

Der „Editor“ (Bereich 2) ist eine Ansicht, die die Projektkonfiguration und das Erstellen der Testfallbeschreibung ermöglicht. Beim ausgewählten Testfall im Test-Explorer werden die zugehörigen Testschritte in der vom Benutzer vorgegebenen

Reihenfolge durch die unterschiedliche Texthervorhebung dargestellt (vgl. Test-Editor 2014a, S. 5). Bei der Testausführung werden die hier definierten notwendigen Eingaben (Daten, Signale, Zeitbedingungen, usw.) an die AUT gesendet und nach der Ausführung die Ist-Zustände (Ergebnisse, Ausgaben, Reaktionen) mit den Soll-Zuständen (Erwartungen) verglichen (vgl. Test-Editor 2014a, S. 57).

Die „Testhistorie“ (Bereich 3) enthält eine Tabelle mit den Informationen bezüglich des Erfolgs oder Misserfolgs der Testausführung eines Testfall oder einer Testsuite (vgl. Test-Editor 2014a, S. 47).

In der „Beschreibung“ (Bereich 4) kann der Benutzer eine Beschreibung eines Testfalls erstellen und editieren (vgl. Test-Editor 2014a, S. 5).

In dem Bereich 5 „Testschritte“ werden einzelne Schritte für den Testfall definiert. Die möglichen Schritte werden projektabhängig aus den vordefinierten Masken ausgewählt. Masken erlauben eine Gruppierung von allen Eingaben und Prüfungen, die sich auf einer Seite des Testobjektes (z.B. eine Seite einer Web-Anwendung) befinden (vgl. Test-Editor 2014a, S. 5).

Über den Bereich 6 „Szenario“ können Szenarien ausgewählt und für einen Testfall verwendet werden. Szenarien sind in der Logik zusammenpassende, universell wiederverwendbare Abschnitte von Testschritten und können auch als Vorlage verwendet werden. Weiterhin können Szenarien in Szenariosuiten gruppiert werden (vgl. Test-Editor 2014a, S. 5).

4. Dashboard – Visualisierung der Testergebnisse

In dem Kapitel „Softwarelieferprozess“ ist immer wieder das Schlagwort Feedback gefallen. Im Prozess des Continuous Delivery sollte ein Feedback schnell und kontinuierlich erfolgen. In Anbetracht der Phase der Akzeptanztests, wie bereits erwähnt, bietet sich das Konzept vom Dashboard als eine praktische Lösung zum Monitoring der vom Test-Editor durchgeführten Akzeptanztests und somit zur Einholung von Feedback. Im Folgenden werden die Anforderungen an das Dashboard definiert. Die Konzipierung basiert unter anderem auf den Ergebnissen einer Umfrage unter den potenziellen Nutzern des Tools.

4.1 Dashboard – allgemeine Beschreibung

Die im Kapitel Continuous Delivery beschriebene Acceptance Test Stage wird nach erfolgreichen Commit Stage ausgeführt.

> Für die Umsetzung der Acceptance Test Stage einer Continuous Delivery Pipeline ist die Anwendung auf eine Testumgebung zu deployen. Dazu müssen zum einen Testumgebungen bereitstehen, zum anderen ist das Deployment und die Konfiguration der Anwendung vollständig zu automatisieren. Sind diese Hürden genommen, kann man zeitnahes Feedback jedoch nur dann gewährleisten, wenn man in der Continuous Delivery Pipeline die Testausführung an vielen Stellen parallelisiert. Um bei der großen Menge an Feedback und Testergebnissen den Überblick zu behalten, ist die Aggregation der Ergebnisse in einem Dashboard sinnvoll (BIRK u. LUKAS 2014b).

In der Acceptance Test Stage nach dem Deployment des Softwarestands auf eine Testumgebung werden die Akzeptanztests ausgeführt und anschließend wird ein Feedback erzeugt (vgl. BIRK u. LUKAS 2014b).

Da für einen Softwaretester nicht nur die quantitative und qualitative Testausführung im Vordergrund steht, ist auch die analytische Auswertung des gesamten Prozesses ein wichtiger Aspekt. „Eine zentrale Aufgabe der Continuous Delivery Pipeline ist das Erzeugen von möglichst zeitnahem Feedback für die beteiligten Entwickler“ (BIRK u. LUKAS 2014b). Der Test-Editor führt die Akzeptanztests schnell und automatisiert durch. Es wird eine Menge von Testergebnissen produziert; es kann unübersichtlich, überladen, schwer lesbar sein und dadurch eine korrekte Interpretation der Messungen erschweren.

Ein Dashboard (Englisch für „Armaturenbrett, Instrumententafel“) ist ein Verlaufs-report oder Werkzeug zur Kontrolle und Überwachung der Prozesse, welches sich als Feedbackwerkzeug bewährt hat. Dashboards können verschiedenste Anzeigen,

Tabellen, Grafiken, Reports beinhalten, um den wichtigsten Vorteil-Gesamtüberblick ohne dabei überfrachtet zu werden zu erfüllen. Dazu werden Dashboards auf bestimmte Anforderungen des Systems angepasst. Bei einem Fahrzeug wäre es ein Armaturenbrett mit Geschwindigkeitsanzeige, Warnleuchten, Zustandsanzeigen usw. Der Fahrzeugführer kann den aktuellen Zustand überwachen und kontrollieren. Um den Überblick zu beschleunigen und zu vereinfachen sind die wichtigsten Informationen möglichst in einer Ansicht konzentriert. Für die Softwarequalität ist nicht nur der aktuelle Zustand wichtig. Das Dashboard zeigt auch die zeitliche Entwicklung, um den Entstehungspunkt der Fehler zu identifizieren. Es sind grafische Darstellungen mit verschieden Farben und Symbolen zur Verdeutlichung der wichtigen Kennzahlen (Metriken) zur schnellen Analyse möglich.

Einige wichtige Anforderungen, die ein Dashboard für die Qualitätssicherungs-werkzeuge erfüllen kann, sind im Folgenden beschrieben. Erstens können solche Leistungsmessungen wie Testdauer visuell angezeigt werden. Zweitens werden negative Tendenzen und Ausreißer identifiziert. Drittens werden Erfolg und Misserfolg gemessen und detaillierte Reports generiert. Weiterhin kann eine gesamte Projektübersicht geschaffen werden, um den Anwender dabei zu unterstützen, die richtigen Entscheidungen anhand der Ergebnisse zu treffen. Außerdem wird dem Anwender dabei geholfen, Teststrategien und Ziele der Tests zu definieren. Schließlich kann der Zeitaufwand dank der Automatisierung minimiert werden.

4.2 Software-Metriken

Für die Anzeigen benutzt das Dashboard Software-Metriken. Eine übergreifende Definition der Metriken gilt auch für die Software-Metriken im Kontext der Qualitätssicherung:

> Metrics are tools designed to facilitate decision making and improve performance and accountability through collection, analysis, and reporting of relevant performance-related data. The purpose of measuring performance is to monitor the status of measured activities and facilitate improvement in those activities in applying corrective action, based on observed measurements (SWANSON 2003, S. VII).

Es gibt viele Metriken, die automatisch berechnet und extrahiert werden. Diese können interne Informationen über den Quellcode, zum Beispiel seine Komplexität oder Qualität, sowie für Akzeptanztests und folgend die für das Dashboard interessanten externen Informationen über das Verhalten der Software gegenüber den Anforderungen liefern. Oft werden die Software-Metriken (Fehleranzahl, Dauer

usw.) als Argumentationsgrundlage gegenüber dem Management genutzt. Da Dashboards im Umfang, Messungen und Anzeigen sehr unterschiedlich sind, sollten die Metriken abhängig von spezifischen Anforderungen für das spezifische Konzept gewählt werden, um Überfrachtung zu vermeiden.

Unter anderem können solche mögliche Auswertungen der Messungen für die Akzeptanztests aufgezählt werden wie Hinweise auf Problemstellen und Handlungsbedarf, zum Beispiel Testdauer oder Fehleranzahl, Entwicklung der Ergebnisse oder Testfall-Abdeckung über die Zeit (Trend-Analyse) oder der Vergleich mit anderen Projekten.

4.3 Erstellung der Anforderungen für das Dashboard

Vor der Erstellung der Anforderungen für das Test-Editor-Dashboard wird die Zielgruppe spezifiziert und die im FitNesse-System existierenden Testverlaufsanzeigen analysiert.

Der Test-Editor ist für die Akzeptanztests konzipiert und die Zielgruppe wird aus den Auftraggebern verschiedener Softwareprodukte gebildet. Programmierkenntnisse sind nicht erforderlich und die Produkte werden von Seiten der Zielgruppe nur auf die Akzeptanzkriterien getestet.

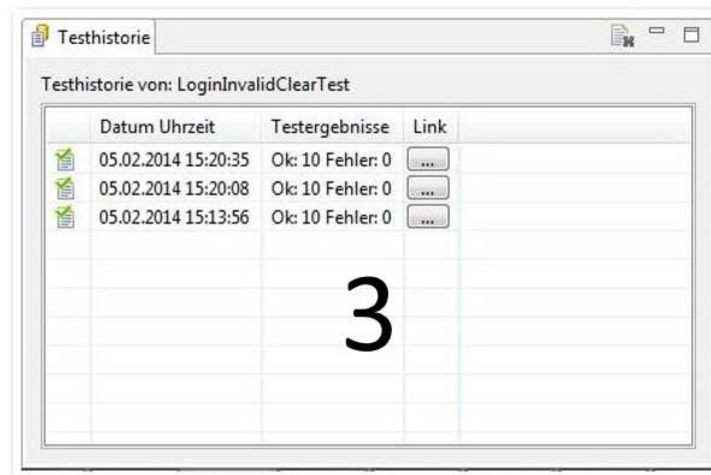


Abbildung 6: Test-Editor Testhistorie (Test-Editor 2014a)

Der Test-Editor besitzt bereits eine Testhistorie (Abbildung 6), diese liefert aber nur wenige Informationen zu einem ausgewählten Testfall: Daten der Testläufe, Anzahl der erfolgreich oder fehlerhaft ausgeführten Testschritte, Link zum FitNesse. Ein Gesamtüberblick über den Testprozess in einem oder mehreren Projekten ist nicht

gegeben, und für die Qualitätsanalyse fehlen viele andere mögliche Messungen. Über den Link gelangt man zum FitNesse-Server, dort werden Informationen zum ausgewählten Testlauf angezeigt. Neben den Testergebnissen werden auch viele weitere nicht unbedingt für den Auftraggeber relevante Informationen wie Testtabellen, die FitNesse-Kenntnisse erfordern, angezeigt. Wie im Kapitel 3.2.1 beschrieben, ist FitNesse ein Testwerkzeug, dessen Funktionen ohne Fachkenntnisse überfrachtet wirken können (Abbildung 7). Es sind mehrere verschiedene Messungen vorhanden, und man kann sowohl die Trends der Test-ausführung beobachten, als auch Reports generieren.

Abbildung 7: FitNesse (Test-Editor 2014a S.47)

Die grobe Anforderung ist daher, das Dashboard auf die ausgewählte Zielgruppe zu spezifizieren und komplett in die Applikation zu integrieren. Eine Möglichkeit, das Dashboard im Test-Editor in einem Fenster anzuzeigen ist bequemer als eine Anzeige als Webseite, zudem findet kein Stilbruch statt, als es bei einem ständigen Wechsel zwischen mehreren Applikationen der Fall wäre. Eine Nutzung der externen FitNesse-Werkzeuge wäre somit nicht mehr nötig.

Als Nächstes werden die Metriken für das Test-Editor-Dashboard spezifiziert. Die Metriken müssen sorgfältig und zielorientiert gewählt werden, deswegen wird die Grundlage aus der Umfrage unter den potenziellen Nutzern gebildet.

4.3.1 Umfrage zu Informationsmetriken und –anzeigen

4.3.1.1 Umfragemethode

Bei der Gestaltung des Fragebogens wurde insbesondere die Methodik von Jakob Rüdiger („Umfrage – Einführung in die Methoden der Umfrageforschung“) verwendet. Die Umfrage richtet sich an die Nutzer des Test-Editors, die über ein ausreichendes Testprozesswissen verfügen. Dabei wurden zunächst die allgemeinen Informationen über die laufenden Projekte in Bezug auf den Testumfang und Zeitaufwand abgefragt. Demzufolge wurden die Informationen über die Testfallentwicklungsprozesse gesammelt. Die zentrale Frage bezog sich auf die Informationsmetriken, durch welche sich der Softwaretester für die Analyse des Testprozesses einen guten Gesamtüberblick verschaffen könnte. Damit der Befragte nicht durch die Fragestellung beeinflusst würde und nur auf sein eigenes Empfinden bezogene Informationen gäbe, wurden offene Fragen verwendet und keine Antworten vordefiniert. Alle Fragen wurden einfach als vollständige Sätze formuliert. Bei Schwierigkeiten mit der Beantwortung der Fragen wurden die Hilfsfragen mit einem Verweis auf mögliche Antwortvorgaben gestellt. Der erstellte Fragebogen ist kurz und gut lesbar und wurde von drei Personen begutachtet.

4.3.1.2 Umfragegestaltung

Vor der eigentlichen Umfrage (Abbildung 8) unter den Nutzern des Test-Editors wurde eine kurze Einführung in das Konzept des Dashboards durchgeführt. Im Einzelnen wurde auf die Einschränkungen des Test-Editors und FitNesse hingewiesen, die im Kapitel (4.3 Anforderungen für den Test-Editor) bereits beschrieben wurden. Weiterhin wurde den Befragten das Konzept des Dashboards und seine Vorteile wie visuelle Darstellung, schneller Überblick, Identifizierung negativer Tendenzen, Zeitaufwandoptimierung vorgestellt. Es wurde um ausführliche Antworten und eigene Vorschläge gebeten.

Umfrage zur Visualisierung der Testergebnisse im Test-Editor mit einem Dashboard.

Allgemeine Informationen über den Befragten

- Wie lange sind Sie als Softwaretester tätig?

Allgemeine Informationen über Projekte

- Wie viele Projekte werden getestet?
- Wie viele Testfälle werden pro Projekt erstellt?
- Wann wird getestet?
- In welchen Zeitabschnitten werden die Tests wiederholt?

Informationen über die Testfallentwicklung

- Welche Informationen über die Entwicklung von Testfällen sind interessant?

Informationen über den Testablauf

- Welche Informationen über den Ausführungsprozess sind interessant?
- Welche Zeitverläufe, Builds, Messungen wären für eine grafische Darstellung relevant?
- Welche Hot-Spots und andere Metriken sind wichtig?
- Werden Berichte erstellt? (Form, Inhalt)

Abbildung 8: Umfrage zu Dashboard-Metriken

Folgende Hilfsfragen sollten, falls notwendig, die Beantwortung erleichtern:

Testfallentwicklung

- Wie viele Testfälle existieren für das Projekt? (Anzeige der gesamten Testfallanzahl).
- Welche Bereiche der AUT sind mit wie vielen Testfällen abgesichert? (Dient zur besseren Einschätzung der Testfallabdeckung der besonders kritischen Funktionen).
- Welche Tests sind in der Entwicklung (Schreiben, Prüfen)?

- Welche Tests sind entwickelt und können ausgeführt werden?
- Wie viele Testfälle sind automatisiert / manuell? (evtl. wird ein Testfall-Lebenszyklus benötigt)

Testablauf

- Wie viele Tests sind zum Zeitpunkt X durchgelaufen?
- Wie viele Tests wurden zum Zeitpunkt X nicht ausgeführt?
- Wie viele Tests waren zum Zeitpunkt X erfolgreich?
- Wie viele Tests waren zum Zeitpunkt X fehlerhaft?
- Wie viele Tests waren zum Zeitpunkt X blockiert?
- Welche Hot-Spots und andere Metriken sind wichtig? Hot-Spots sind signifikante Stellen oder Zahlen die eine Häufigkeit der bestimmten Ereignisse oder Ausreißer kennzeichnen und deswegen besonders für die Analyse von Testprozessen interessant sind.
- Welche Metriken zeigen, dass der Test gut oder schlecht ist?
- Wie ist der Trend der Testausführung? (Entwicklung der Ausführungsdauer, Anzahl der Testfälle und Ergebnisse)

4.3.1.3 Analyse der Umfrageergebnisse

Um überwiegende Wünsche der Befragten zu erfüllen, werden für die relevanten Anforderungen die Anhäufungen ähnlicher Antworten aus der Umfrage in Betracht gezogen. Folgende Zusammenfassung der Umfrageergebnisse (Tabelle 1) zeigt die Häufigkeit ähnlicher Antworten, die zu einzelnen Fragen gehören. Die im Team priorisierten Bereiche sind in rot markiert. Die kompletten Umfrageergebnisse sind in dem Anhang 21.

	Antwortanzahl
Wie viele Projekte werden getestet?	
1 Projekt	4
5 Projekt	1
3 Projekt	1
Wie viele Testfälle werden pro Projekt erstellt?	

1	1
50	1
70	1
Über 1000	3
Wann wird getestet?	
Nicht definiert	3
Jeden Tag	2
Alle 2 Monate	1
Alle 3 Monate	1
In welchen Zeitabschnitten werden die Tests wiederholt?	
Nicht definiert	1
Täglich	3
Alle 2 Monate	2
Welche Informationen über die Entwicklung von Testfällen sind interessant?	
Nicht relevant	5
Entwicklung der Testfallanzahl	2
Prozentuale Abdeckung des AUT	1
Welche Informationen über den Ausführungsprozess von Testfällen sind interessant?	
Welche Testfälle sind fehlgeschlagen, welche waren erfolgreich.	6
Fehleridentifizierung	3
Geringe Abweichung von Sollwert	1
Testschritte in einem Testfall	1
Fehlerhafte Testschritte	1
Welche Zeitverläufe, Builds, Messungen wären für eine grafische Darstellung relevant?	
Dauer einzelner Tests	4
Dauervergleich, Ausreißer	2

Ergebnisse des letzten Builds	1
Zeitbereich frei definierbar	1
Jahrelange Speicherung der Ergebnisse	1
Aktuelle Anzeige	2
Trend der Fehlerentwicklung einzelner Testfalls	1
Welche Hot-Spots und andere Metriken sind wichtig?	
Komplexität der Testfälle	1
Testfallergebnis ist Fehlerhaft, Erfolgreich (allgemein)	1
Prozentuale Angabe, wie oft war Test "rot" oder "grün"	1
Werden Berichte erstellt?	
Keine Berichte	4
Speichern der Berichte	1

Tabelle 1: Umfrageergebnisse

Aus der Umfrage konnten folgende Anforderungen für das Dashboard priorisiert werden:

- Es muss die Möglichkeit bestehen mit mehreren Projekten zu arbeiten;
- In einem Testzyklus können über 1000 Testfälle ausgewertet werden;
- Vorrangig sollen aktuellste Ergebnisse angezeigt werden;
- Die Anzeigen müssen zu jedem Zeitpunkt aufrufbar sein;
- Hinweise auf erfolgreiche/fehlerhafte Testdurchläufe müssen schnell ersichtlich sein;
- Es sollen Informationen zu den fehlerhaften Schritten abrufbar sein;
- Als wichtige Metrik soll die Test-Dauer und Dauertrend angezeigt werden.

4.4 Konzeption

4.4.1 Analyse der Test-Editor Umgebung

Die Anzeigen des Dashboards werden mit den Daten aus den Testergebnissen gefüllt. Zur Festlegung der Lösungswege, die es realisieren werden, muss zuerst die Beschaffenheit der benötigten Daten und Informationen analysiert werden.

4.4.1.1 Lokale Testergebnisdateien

Eine Möglichkeit wäre, die Testlaufergebnisse aus lokal gespeicherten Dateien zu extrahieren. Dazu muss man erklären, wo die Testergebnisse gespeichert werden.

Die Speicherung der Testergebnisse wird am Beispiel eines Projektes erläutert. Das Projekt *DemoSwingTest* beinhaltet eine Testsuite (*GeburtstagVerwaltungsSuite*), die aus zwei Testfällen (*AnlegenUndPruefenTest* und *MassenAnlageTest*) besteht. Beim Erzeugen eines Projektes im Test-Editor mit Standard-Einstellungen wird ein Ordner mit dem Namen des Projektes lokal und zwar in dem Verzeichnis

```
C:\Users\Benutzername\.testeditor
```

angelegt. Der komplette Pfad lautet dann

```
C:\Users\Benutzername\.testeditor\DemoSwingTests
```

Bei der Ausführung eines Testfalls oder einer Testsuite werden einmalig weitere Ordner, die die Testergebnisdateien beinhalten, erstellt. Diese Ordner werden mit den Namen des Projektes, der Suiten und der Testfälle versehen und unter dem folgenden Pfad gespeichert:

```
C:\Users\Benutzername\.testeditor\DemoSwingTests\FitNesseRoot\files\testResults
```

Die eigentlichen Testresultatdateien, die die Ergebnisse des Testlaufs beinhalten, werden nach jeder Testausführung in den zugehörigen Ordnern abgelegt. Wie schon im Kapitel 3.3 angedeutet, wird bei der Ordner-Namensgebung eine hierarchische Struktur verwendet. Als Beispiel werden Testresultatdateien von dem Testfall *AnlegenUndPruefenTest* im folgenden Verzeichnis zu finden sein:

```
C:\Users\Benutzername\.testeditor\DemoSwingTests\FitNesseRoot\files\testResults\
DemoSwingTests.GeburtstagVerrwaltungsSuite.AnlegenUndPruefenTest
```

Jeder Name der Testresultatdatei ist aus dem Zeitstempel (Timestamp) des Ausführungszeitpunktes und Anzahlen der erfolgreichen (*right*), fehlerhaften (*wrong*), ignorierten (*ignored*) Testschritten, sowie Ausnahmenanzahl (*exceptions*) gebildet. Der Zeitstempel ist aus dem Jahr, Monat, Tag, Stunden, Minuten und Sekunden zusammengesetzt. Beispielhaft ist diese Struktur im folgenden Dateinamen in entsprechender Reihenfolge zu sehen:

```
20141016112330_20_2_0_0.xml
```

Die Testresultatdatei ist im XML-Format erstellt und beinhaltet alle testrelevanten Daten. Es wird für die Ausführung benutzte FitNesse-Version angezeigt sowie der Testfallname, die Gesamtauswertung der Ergebnisse aller Testschritte, die gesamte Testlaufdauer. Unter anderem werden auch einzelne Aktionen (Testschritte) und Reaktionen des AUTs auf jeden Testschritt aufgelistet (Abbildung 9).

```
 1 <?xml version="1.0"?>
 2 <testResults>
 3   <FitNesseVersion>v20121220 Build-20140417</FitNesseVersion>
 4   <rootPath>MassenAnlageTest</rootPath>
 5     <result>
 6     <counts>
 7       <right>20</right>
 8       <wrong>0</wrong>
 9       <ignores>0</ignores>
10       <exceptions>0</exceptions>
11     </counts>
12     <runTimeInMillis>1313355</runTimeInMillis>
13     <content><![CDATA[<div class="collapsible closed"><ul><li>
14   <p class="title">Precompiled Libraries</p>
15   <div><br/><div class="collapsible closed"><ul><li><a href='#
16   <p class="title">Included page: <a href="DemoSwingTests.Scen
17   <div><table _TABLENUMBER=4978972598333689963>
18   <tr>
19     <td>import</td>
20   </tr>
21   <tr>
22     <td>org.testeditor.fixture.swing</td>
23   </tr>
24 </table>
```

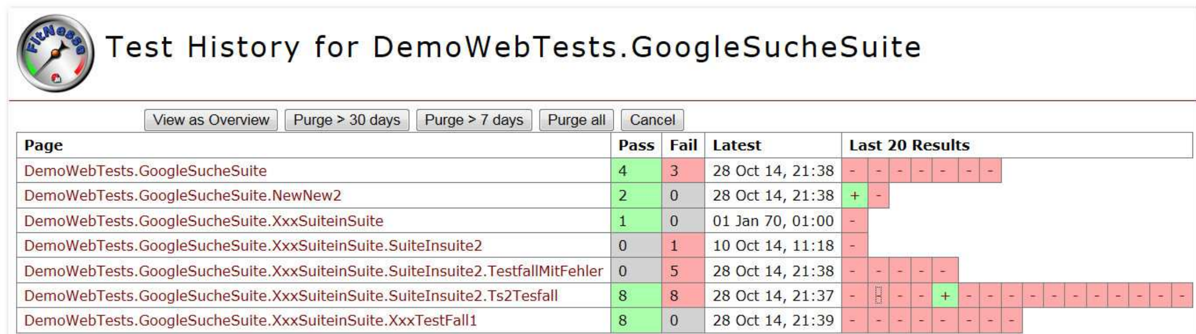
Abbildung 9: Testergebnisse in XML-Datei

Bei der Testsuiteresultatdatei werden zusätzlich alle beinhalteten Testfälle oder Testsuiten und deren Ergebnisse sowie Testlaufdauer getrennt angezeigt. Es ist zu beachten, dass die Namen der Testfall-, Testsuitenordner und deren Bezeichnung in den Testresultatdateien in einer hierarchischen Struktur gebildet werden. Bei den in einem Projekt übergeordneten Testfällen oder Testsuiten ist die Namensgebung wie *ProjektName.TestfallName* oder entsprechend *ProjektName.TestsuiteName* vorgesehen. Bei Testfällen, die zu einer Testsuite oder gar zu in einander geschachtelten Testsuiten gehören, wäre die Benennung als *ProjektName.TestsuiteName.TestfallName* und dementsprechend *ProjektName.TestsuiteName1.TesuiteName2.Testfallname* gestaltet. Die Testresultatdateien kann man mit definierten Algorithmen nach bestimmten Informationen durchsuchen und diese an die Anzeigen des Dashboards übergeben.

4.4.1.2 FitNesse-Server

Die andere Möglichkeit besteht darin, sich die Testlaufergebnisse direkt von dem Fitnesse-Server zu beschaffen. Dies könnte mit einem Representational State

Transfer Aufruf (abgekürzt REST) erfolgen. Damit könnte man aus einer FitNesse-Testergebnisseite mit den GET-Befehlen benötigte Informationen auslesen und für das Dashboard verwenden. Die Ergebnisse sind in einer spezifischen FitNesse-Struktur gegliedert (Abbildung 10), die man über mehrere Algorithmen in die für die Dashboards-Anzeigen gewünschte Form überarbeiten müsste.


Test History for DemoWebTests.GoogleSucheSuite

View as Overview | Purge > 30 days | Purge > 7 days | Purge all | Cancel

Page	Pass	Fail	Latest	Last 20 Results
DemoWebTests.GoogleSucheSuite	4	3	28 Oct 14, 21:38	- - - - - - -
DemoWebTests.GoogleSucheSuite.NewNew2	2	0	28 Oct 14, 21:38	+ -
DemoWebTests.GoogleSucheSuite.XxxSuiteinSuite	1	0	01 Jan 70, 01:00	-
DemoWebTests.GoogleSucheSuite.XxxSuiteinSuite.SuiteInsuite2	0	1	10 Oct 14, 11:18	-
DemoWebTests.GoogleSucheSuite.XxxSuiteinSuite.SuiteInsuite2.TestfallMitFehler	0	5	28 Oct 14, 21:38	- - - - -
DemoWebTests.GoogleSucheSuite.XxxSuiteinSuite.SuiteInsuite2.Ts2Tesfall	8	8	28 Oct 14, 21:37	- [illegible] - - + - - - - - - - - - - -
DemoWebTests.GoogleSucheSuite.XxxSuiteinSuite.XxxTestFall1	8	0	28 Oct 14, 21:39	- - - - - - - -

Abbildung 10: FitNesse Test History

4.4.2 Lösungsansätze und Festlegung des Konzepts

Nach der Analyse der Datenbeschaffung wird der effizienteste Einsatz der Lösungsmöglichkeiten ermittelt werden. Da die aus dem FitNesse-Server abgefragten Daten eine umfangreiche Überarbeitung der Datenstruktur verlangen, wird der Lösungsweg auf die lokal gespeicherten Testergebnisdaten fokussiert. Dieser Weg erfordert auch eine algorithmische Auslesung und Bearbeitung der Daten, die aber auf dem direkten Weg aus den Testresultatdateien abgerufen werden.

4.4.2.1 BIRT als Lösung

Während der Recherche für die bereits existierenden Open-Source Berichterstattungswerkzeuge ist besonders BIRT für den möglichen Einsatz aufgefallen.

Business Intelligence and Reporting Tools (BIRT) ist ein Open-Source-Projekt von Eclipse. Es stellt eine Berichtswesen- und Business-Intelligence-Funktionalität für Rich Clients und Web-Applikationen zur Verfügung. Insbesondere eignet es sich für Applikationen, die Java- bzw. JavaEE-basiert implementiert sind. BIRT ist einer der meist verbreiteten Visualisierungs- und Berichtserstattungstechnologien (vgl. BIRT 2015a).

Die Architektur basiert auf fünf Komponenten.

Die erste Komponente ist ein graphischer Berichte-Editor (Report Designer) innerhalb der Eclipse IDE. Dieser ermöglicht BIRT-Berichte zu entwerfen. Man kann Design, Tabellen, Grafiken usw. definieren und mit den zugehörigen Daten verknüpfen.

Die zweite Laufzeitkomponente (Design Engine) dient der Erzeugung und Modifizierung von Berichten und kann in jeder Java-Umgebung eingesetzt werden. Es wird intern von BIRT zum Konstruieren von XML-Designs verwendet.

Die dritte Komponente, Report Engine, generiert den Report unter Verwendung von Reportdesigndateien. Diese Komponente wird auch vom BIRT Web Viewer zur Reportanzeige genutzt.

Das BIRT-Projekt enthält auch eine vierte Komponente (Chart-Engine), die sowohl in den Berichte-Editor integriert ist, als auch eigenständig innerhalb einer Anwendung verwendet werden kann. Es ermöglicht den Entwickler ihre Software mit der Grafikgenerierung zu versorgen. Diese Komponente wird von der Design und Report Engine genutzt.

Die letzte, fünfte Komponente (BIRT Viewer) macht die Vorschau der Berichte innerhalb des Eclipse möglich. Es kann auch in das Softwareprodukt eingebettet werden.

Die Berichte können zum Beispiel als HTML, PDF, XLS, DOC, PPT und Postscript ausgegeben oder auch in CSV exportiert werden. Die BIRT-Berichte-Designs werden als XML-Dateien gespeichert und können auf zahlreiche Datenquellen zugreifen (vgl. BIRT 2015b).

BIRT bietet eine breite Palette an Reports für die Software (Abbildung 11). Listen als einfachste Methode zur Übersicht der Daten können vielfältig gruppiert werden sowie mathematische Funktionen wie Summierung oder Durchschnitt zur Verfügung stellen. Die interaktiven Grafiken bieten die Möglichkeit, numerische Daten leichter zu interpretieren. Unter anderem werden auch Kontingenztafeln, Nachrichten, Dokumente, Notizen und Reports-Kombinierung angeboten (vgl. BIRT 2015c).

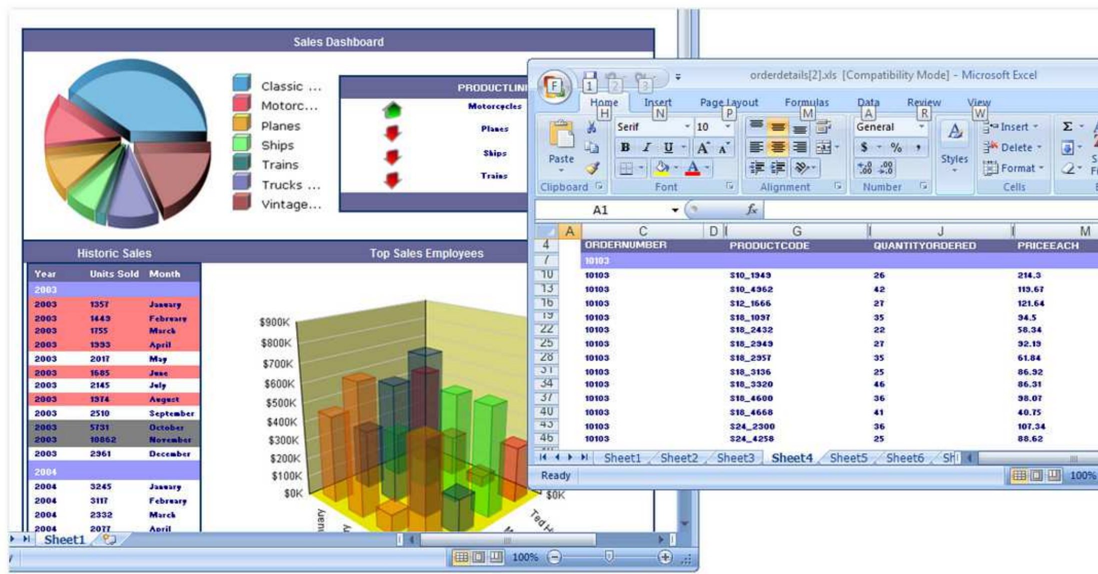


Abbildung 11: BIRTs verschiedene Dashboards Ansichten (BIRTWORLD 2015)

4.4.2.1 Festlegung des Konzepts

BIRT ist ein sehr professionelles und umfangreiches Werkzeug. Es kann mehr als durch die Dashboard-Anforderungen gestellten Aufgaben erledigen. Der Einsatz von BIRT würde zweifelsfrei viele Vorteile für mögliche zukünftige Erweiterungen des Dashboards bringen. BIRT erfordert eine nicht zu vernachlässigbare Einarbeitung in die Technologie und Funktionen. Es ist wichtig zu betonen, dass die Berichtsgenerierung aus Datenbanken erfolgt. Die Testergebnisse des Test-Editors werden aber nicht in Datenbanken, sondern, wie bereits beschrieben, in einzelnen Testresultatdateien im XML-Format gespeichert. Als schnelle Möglichkeit zur Nutzung von BIRT wurde eine XML-Datenbank aus den existierenden Testresultatdateien erstellt. Man könnte die Testdaten entweder nach jeder Testausführung sofort in der XML-Datenbank speichern oder erst bei der Berichtserstellung aus allen existierenden Testresultatdateien auslesen. Mit bestimmten Auslesealgorithmen kann man verschiedene Datenbanken für die Dashboard-Elemente herstellen. Auf Dauer sind aber solche XML-Datenbanken unpraktisch, denn diese werden immer mehr Speicher in Anspruch nehmen, und es müssen weitere Funktionen zur Verwaltung der Datenbanken implementiert werden. Der Einsatz einer richtigen, zum Beispiel SQL-Datenbank, für die effektive BIRT-Nutzung ist somit unumgänglich. Es ist auch wichtig zu beachten, dass BIRT einen Speicherplatz von ca. 280 MB benötigt. Verglichen damit ist der Test-Editor selbst 117 MB groß. So würde der Einsatz von BIRT den Softwarespeicherbedarf mehr als

verdreifachen und sehr verschwenderisch wirken, da nur ein kleiner Teil der BIRT-Funktionen für den Test-Editor-Dashboard vorerst in Betracht käme.

Unter Beachtung solcher Aspekte wie Speicherplatzökonomie und unumgängliche algorithmische Implementierung der Auslesefunktionen für die Testresultatdateien scheint eine selbständige Implementierung des Dashboards als Plug-In die effizienteste Lösung zu sein. Solche Entwicklung würde dem Bedarf der Zielgruppe und den Vorstellungen der Softwareentwickler entsprechen, Speicherressourcen sparen und eine komplette Integrierung durch die Ausnutzung der bereits in der Test-Editor-Entwicklung verwendeten Technologien garantieren.

4.4.3 Designentwurf und Funktionen

Nach der groben Festlegung der Entwicklungsvorgehensweise muss ein passendes interaktives Design der visuellen Oberfläche des Dashboards erstellt werden. Ein endgültiges Design muss mit dem Test-Editor-Design übereinstimmen und alle priorisierten Anforderungen abdecken.

Folgender Designentwurf zeigt eine mögliche Dashboard-Oberfläche. Die Oberfläche wurde im Laufe des Implementierens mehrmals optimiert und überarbeitet. Im Folgenden werden die einzelnen Elemente sowie die im Entwicklungsprozess entstandenen Erweiterungen und Änderungen im Sinne der Funktionalität erläutert. Abbildung 12 zeigt den ersten Designentwurf, und die endgültige Designentwicklung ist auf der Abbildung 13 zu sehen.

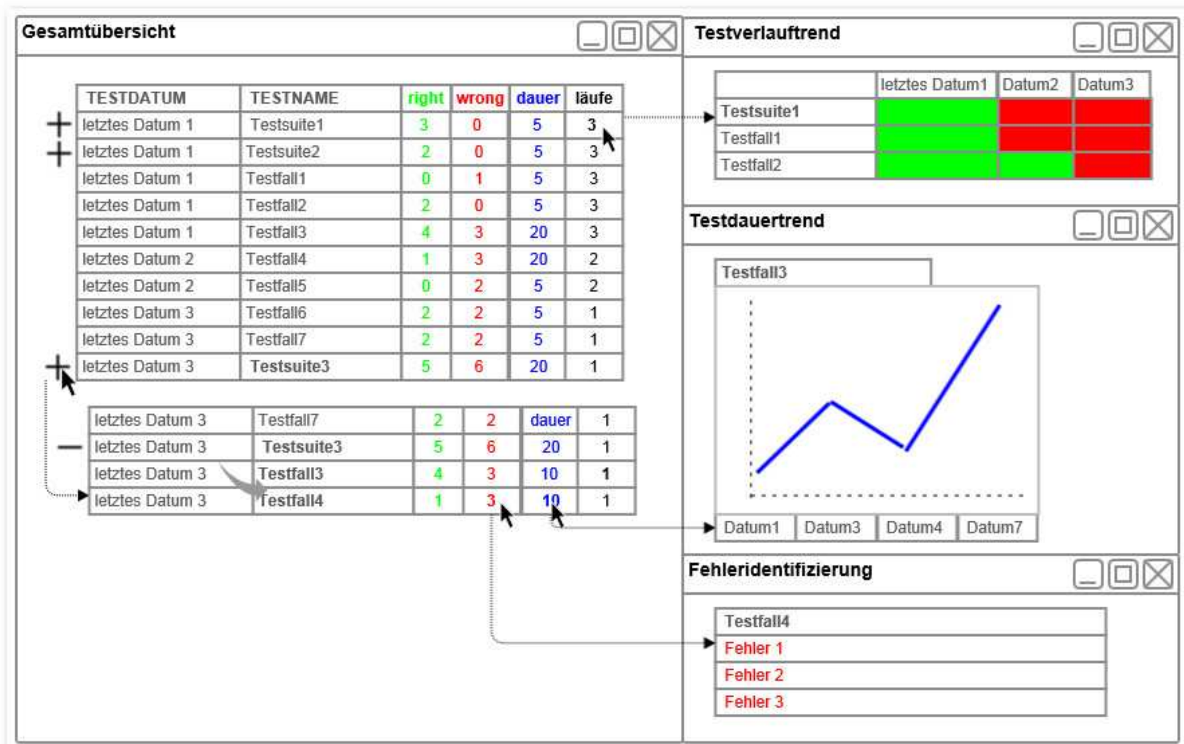


Abbildung 12: Erster Designentwurf

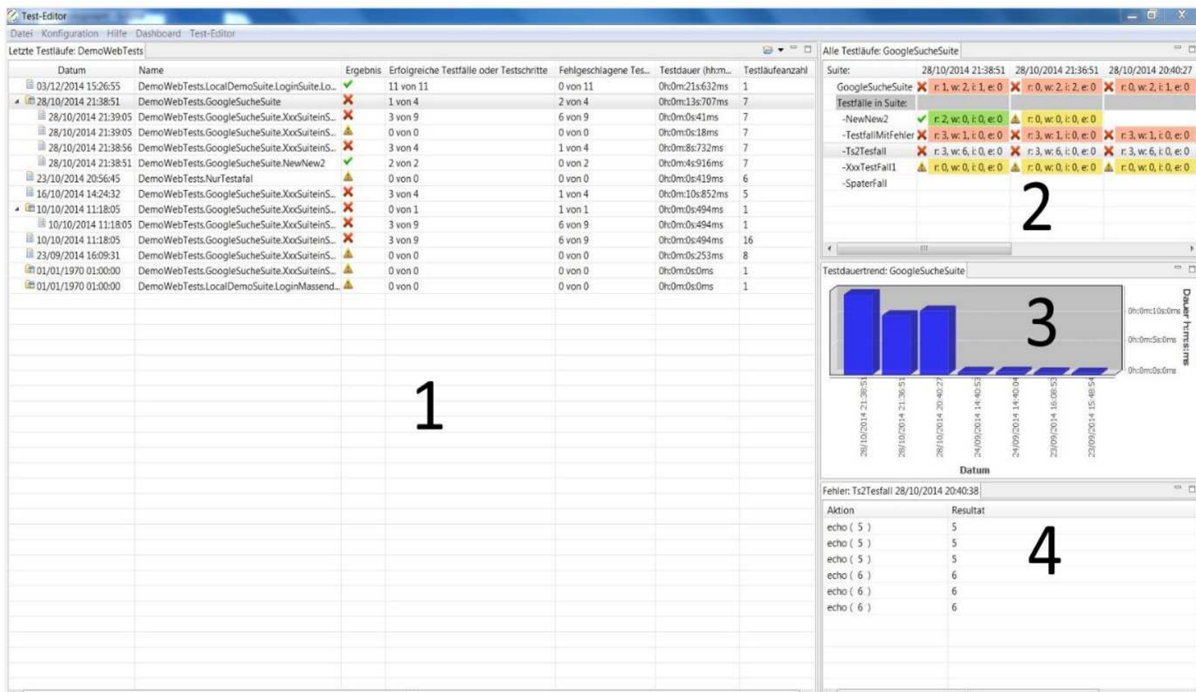


Abbildung 13: Endgültiges Design

Dashboards-Oberfläche

Für die Ausnutzung des gesamten Programm-Fensters wird das Dashboard (Abbildung 13) in einer eigenen Ansicht konstruiert. Somit wird ein Umschalter

benötigt, mit dem man zwischen dem Test-Editor- und den Dashboard-Ansichten wechseln kann. Die Umschaltung wird mit Hilfe des „Test-Editor“- „Dashboard“-Buttons (Abbildung 14) in dem Menübereich des Test-Editors umgesetzt.

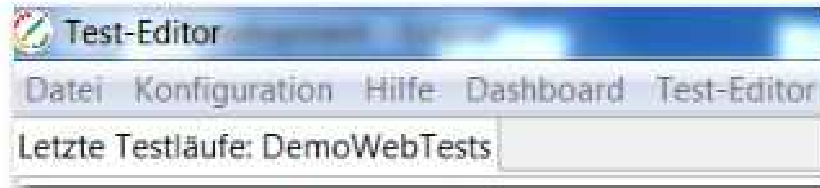


Abbildung 14: Umschaltung der Ansichten

Die Oberfläche ist in vier Bereiche (Abbildung 13) aufgeteilt: „Letzte Testläufe“, „Alle Testläufe“, „Testdauertrend“ und „Fehler“. Alle Tabellen besitzen eine Scroll-Bar, die eine freie Bewegung über die angezeigten Daten garantiert. Die Tabellen-Spalten sowie alle Bereiche sind für den individuellen Überblick in der Größe anpassbar. Zusätzlich sind die Bereichs- (Abbildung: 15) und Spalten-Überschriften (Abbildung: 16) mit Tooltips versehen. Tooltips sind kleine Pop-up-Fenster, die bei Mouseover auf ein Element die entsprechenden Schnellinfos anzeigen.

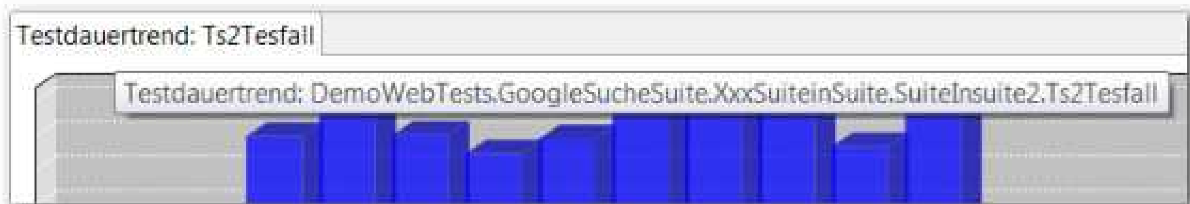


Abbildung 15: Tooltip für Bereichs Überschrift

Fehlgeschlagene Testfälle oder Testschritte | Testdauer (hh:mm:ss:ms) | Testläufeanzahl

Für Testfälle werden Testschrittanzahlen angezeigt ("2 von 4" bedeutet 2 Schritte von 4 Schritten gesamt).

Für Suiten werden Testfallanzahlen angezeigt ("2 von 4" bedeutet 2 Testfälle von 4 Testfällen gesamt).

0 von 0 | 0h:0m:0s:18ms | 7

Abbildung 16: Tooltip für Spalten Überschrift

Bereich 1: Tabelle „Letzte Testläufe“

Datei Konfiguration Hilfe Dashboard Test-Editor

Letzte Testläufe: DemoWebTests

Datum	Name	Ergebnis	Erfolgreiche Testfälle oder Testschritte	Fehlgeschlagene Testfälle oder Testschritte	Testdauer (hh:mm:ss:ms)	Testläufeanzahl
03/12/2014 15:26:55	DemoWebTests.LocalDemoSuite.LoginSuite.Lo...		11 von 11	0 von 11	0h:0m:21s:632ms	1
28/10/2014 21:38:51	DemoWebTests.GoogleSucheSuite		1 von 4	2 von 4	0h:0m:13s:707ms	7
28/10/2014 21:39:05	DemoWebTests.GoogleSucheSuite.XxxSuiteinS...		3 von 9	6 von 9	0h:0m:0s:41ms	7
28/10/2014 21:39:05	DemoWebTests.GoogleSucheSuite.XxxSuiteinS...		0 von 0	0 von 0	0h:0m:0s:18ms	7
28/10/2014 21:38:56	DemoWebTests.GoogleSucheSuite.XxxSuiteinS...		3 von 4	1 von 4	0h:0m:8s:732ms	7
28/10/2014 21:38:51	DemoWebTests.GoogleSucheSuite.NewNew2		2 von 2	0 von 2	0h:0m:4s:916ms	7
23/10/2014 20:56:45	DemoWebTests.NurTestafal		0 von 0	0 von 0	0h:0m:0s:419ms	6
16/10/2014 14:24:32	DemoWebTests.GoogleSucheSuite.XxxSuiteinS...		3 von 4	1 von 4	0h:0m:10s:852ms	5
10/10/2014 11:18:05	DemoWebTests.GoogleSucheSuite.XxxSuiteinS...		0 von 1	1 von 1	0h:0m:0s:494ms	1
10/10/2014 11:18:05	DemoWebTests.GoogleSucheSuite.XxxSuiteinS...		3 von 9	6 von 9	0h:0m:0s:494ms	16
23/09/2014 16:09:31	DemoWebTests.GoogleSucheSuite.XxxSuiteinS...		0 von 0	0 von 0	0h:0m:0s:253ms	8
01/01/1970 01:00:00	DemoWebTests.GoogleSucheSuite.XxxSuiteinS...		0 von 0	0 von 0	0h:0m:0s:0ms	1
01/01/1970 01:00:00	DemoWebTests.LocalDemoSuite.LoginMassend...		0 von 0	0 von 0	0h:0m:0s:0ms	1

Abbildung 17: Tabelle „Letzte Testläufe“ – Bereich 1

Gemäß den definierten Anforderungen ist es notwendig, für das Dashboard eine große Menge der Testergebnisse zu bewältigen. Für solch eine Anzahl der Daten ist die Verwendung einer tabellarischen Ansicht sinnvoll (Abbildung 17). Für den besseren Überblick beansprucht dieser Bereich den größten Teil der Ansicht des Dashboards, ist aber wie schon erwähnt in der Größe veränderbar.

Die Testergebnisse können sowohl von einem Testsuitelauf, als auch von einem Testfalllauf stammen. In Anbetracht dieser Information bringt eine Baumansicht (Tree-View) eine bessere, kompakte Übersicht. Die Kind-Elemente (Testfälle), falls diese innerhalb einer Testsuite ausgeführt wurden, sind in die zugehörigen Testsuiten gruppiert. Diese sind ebenfalls auf- und zu-klappbar. Die Ergebnisse der Testfälle, die unabhängig von einer Testsuite ausgeführt wurden, besitzen eigene Zeilen. In der zugeklappten Tree-View wird der aktuellste Einzellauf einer Testsuite oder eines Testfalls angezeigt. Zum Beispiel, werden ein Testfall A am 01.01.2015 und 02.01.2015 und einer Testsuite, die die Testfälle A und B beinhaltet, am 03.01.2015 und 04.01.2015 ausgeführt, so werden schließlich zwei Zeilen (Testsuite vom 04.01.2015 und Testfall A vom 02.01.2015) in der Tabelle stehen. Dabei wird die Testsuite-Zeile aufklappbar sein und die Zeilen mit Testfall A und B vom 04.01.2015 beinhalten. Da die Testfälle und Testsuiten nicht unbedingt sofort aus deren Namen erkennbar sind, werden für die Identifizierung die bereits im Test-Editor verwendeten zugehörigen Symbole (Icons) benutzt (Abbildung 18).

10/10/2014 11:18:05 DemoWebTests.GoogleSuche
10/10/2014 11:18:05 DemoWebTests.GoogleSuche
10/10/2014 11:18:05 DemoWebTests.GoogleSuche

Abbildung 18: Icons: Testfall, Testsuite

Die Informationen über die Testläufe sind in Spalten aufgeteilt. Es werden Datum der Testausführung, hierarchisch aufgebauter Testfall- oder Testsuite-Name, Ergebnis, Anzahl der erfolgreichen und fehlgeschlagenen Testfälle in einer Testsuite oder Testschritte in einem Testfall, Testdauer und gesamte Anzahl der bislang ausgeführten Tests jeweils für jeden Testlauf angezeigt. Alle Spalten können entsprechend alphabetisch, numerisch oder nach Datum sortiert werden.

Für eine schnelle Analyse der Ergebnisse wurden die farblichen Markierungen in der Ergebnis-Spalte verwendet: rot für *fehlerhaft*, grün für *erfolgreich* und gelb für *Warnung*. Anstatt nur eine farbliche Markierung zu verwenden werden die Ergebnisse zur barrierefreien Auswertung mit leicht verständlichen Icons in zugehöriger Farbe erweitert: Kreuz für *fehlerhaft*, Häkchen für *erfolgreich* und Dreieck mit Ausrufezeichen für *Warnung*. (Abbildung 19)

Abbildung 19: Icons: ok, fail, warning

Die Tabelle trägt als Überschrift den Projektnamen. Weiterhin ist es möglich, unter Verwendung eines Dropdown-Menus (Abbildung 20) zwischen anderen existierenden Projekten umzuschalten. Bei der Auswahl eines anderen Projektes wird die Tabelle mit den zugehörigen Testdaten aktualisiert und die Bereiche 2, 3 und 4, die weiter unten beschrieben werden, werden geleert.

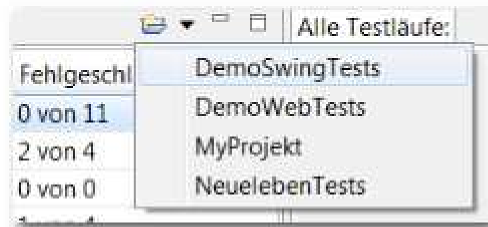


Abbildung 20: Projekte in Dropdown Menü

Beim Selektieren mittels Doppel-Klicks einer beliebigen Zelle in der Tabelle wird der entsprechende Testlauf ausgewählt, gleichzeitig werden die Bereiche 2 und 3 mit den zugehörigen Daten aktualisiert beziehungsweise Bereich 4 geleert.

Bereich 2: Tabelle „Alle Testläufe"

Die Ansicht des Bereichs 2 hängt von dem im Bereich 1 ausgewählten Testlauf ab. Bei der Auswahl einer Testsuite wird eine Testsuite-Tabelle angezeigt (Abbildung 21).

Test-Editor

Datei Konfiguration Hilfe Dashboard Test-Editor

Alle Testläufe: GoogleSucheSuite

Suite:	28/10/2014 21:38:51	28/10/2014 21:36:51	28/10/2014 20:40:27	24/09/2014 14:40:53	24/09/2014 14:40:04	23/09/2014 16:08:53	23/09/2014 15:48
GoogleSucheSuite	r: 1, w: 2, i: 1, e: 0	r: 0, w: 2, i: 2, e: 0	r: 0, w: 2, i: 1, e: 0	r: 0, w: 0, i: 3, e: 0	r: 0, w: 0, i: 3, e: 0	r: 0, w: 0, i: 3, e: 0	r: 0, w: 0, i: 3, e
Testfälle in Suite:							
-NewNew2	r: 2, w: 0, i: 0, e: 0	r: 0, w: 0, i: 0, e: 0					
-TestfallMitFehler	r: 3, w: 1, i: 0, e: 0	r: 3, w: 1, i: 0, e: 0	r: 3, w: 1, i: 0, e: 0				
-Ts2Tesfall	r: 3, w: 6, i: 0, e: 0	r: 3, w: 6, i: 0, e: 0	r: 3, w: 6, i: 0, e: 0	r: 0, w: 0, i: 0, e: 0	r: 0, w: 0, i: 0, e: 0	r: 0, w: 0, i: 0, e: 0	r: 0, w: 0, i: 0, e
-XxxTestFall1	r: 0, w: 0, i: 0, e: 0	r: 0, w: 0, i: 0, e: 0	r: 0, w: 0, i: 0, e: 0	r: 0, w: 0, i: 0, e: 0	r: 0, w: 0, i: 0, e: 0	r: 0, w: 0, i: 0, e: 0	r: 0, w: 0, i: 0, e
-SpaterFall				r: 0, w: 0, i: 0, e: 0	r: 0, w: 0, i: 0, e: 0	r: 0, w: 0, i: 0, e: 0	r: 0, w: 0, i: 0, e

Abbildung 21: Tabelle „Alle Testläufe": Testsuitelauf – Bereich 2

Die Tabelle enthält den Testsuitenamen als Überschrift. In diesem Beispiel wurde eine *GoogleSucheSuite,* die fünf Testfälle beinhaltet, ausgewählt. Die erste Spalte (Namen-Spalte) enthält den Testsuitenamen sowie Namen aller beinhalteten Testfälle. Die Testfallnamen sind wie in einer Baumstruktur hierarchisch eingerückt. Andere Spalten (Ergebnis-Spalten) beinhalten die Testergebnisse einzelner Testausführungen: Eine Spalte entspricht genau einem Testlauf. Als Überschrift haben diese Spalten das Datum und die Uhrzeit des Testlaufs. Die Sortierung ist analog der FitNesse-Testhistory aufgebaut: Sie ist von links nach rechts zu lesen und beginnt mit dem aktuellsten Testlauf.

Die erste Zeile (Testsuiten-Zeile) fängt mit dem Testsuite-Namen an und beinhaltet die Testsuitelauf-Ergebnisse. Außerdem enthalten die Zellen (Testsuiten-Zellen) in

der Testsuiten-Zeile die Anzahlen von richtigen mit einem „r“, fehlerhaften mit einem „f“, ignorierten mit einem „i“ oder ausnahmebehafteten mit einem „e“ gekennzeichnete Testfälle. Die zweite Zeile (Trenn-Zeile) dient zur Abgrenzung der Testsuite von den Testfällen und ist in grau hinterlegt. Die nachfolgenden Zeilen (Testfall-Zeilen) fangen mit dem Namen der Testfälle an und präsentieren jeweils zu jedem Testfall die zugehörigen Testfalllauf-Ergebnisse. Die Zellen (Testfall-Zellen) in den Testfall-Zeilen stellen die gleichen Informationen für die Testschritte des zugehörigen Testfalls dar. Allgemein sind die Ergebnisse in den Zellen durch entsprechende farbliche Markierung und Icons, wie schon im Bereich 1 oben beschrieben wurde, zu erkennen.

Diese Tabelle ermöglicht nicht nur die Anzeige aller Testläufe einer Testsuite, sondern hilft auch die Entwicklung der Testsuite zu analysieren. Die leeren Zellen zeigen, dass ein zugehöriger Testfall zum Ausführungszeitpunkt ausgeschlossen, noch nicht erstellt oder entfernt wurde.

Im Einzelnen zeigt beispielweise, wie in der Abbildung 21 dargestellt ist, der Testlauf am 28.10.2014 um 21:38:51 (erste Ergebnis-Spalte) einen fehlerhaften Lauf der *GoogleSucheSuite.* Vier Testfälle waren in dieser Testsuite ausgeführt. Ein Testfall war erfolgreich, zwei waren fehlerhaft und einer wurde ignoriert. Der fünfte Testfall, der kein Ergebnis in der Zelle hat, wurde aus dem Lauf ausgeschlossen oder gar gelöscht.

Bei der Auswahl eines Testfalls im Bereich 1 wird die Darstellung im Bereich 2 etwas einfacher (Abbildung 22).

Test-Editor
Datei Konfiguration Hilfe Dashboard Test-Editor
Alle Testläufe: Ts2Tesfall

28/10/2014 21:37:01	28/10/2014 20:40:38	10/10/2014 11:18:05	24/09/2014 15:58:12	24/09/2014 15:57:41	24/09/2014 15:54:56	24/09/2014 15:54:12	24/09/2014 15:53:51
r: 3, w: 6, i: 0, e: 0	r: 3, w: 6, i: 0, e: 0	r: 3, w: 6, i: 0, e: 0	r: 3, w: 6, i: 0, e: 0	r: 3, w: 0, i: 0, e: 0	r: 0, w: 0, i: 0, e: 0	r: 0, w: 0, i: 0, e: 0	r: 0, w: 0, i: 0, e: 0

Abbildung 22: Tabelle „Alle Testläufe“: Testfalllauf – Bereich 2

Die Tabelle bekommt den Testfallnamen als Überschrift. Da Testfälle keine Kinderelemente besitzen können, beinhalten hier alle Spalten, im Unterschied zur oben beschrieben Testsuite-Darstellung, die Ergebnisse von einzelnen

Testausführungen. Die Zellen zeigen entsprechend nur die testfallrelevanten Informationen.

Allgemein gilt Folgendes: Beim Doppel-Klick auf eine beliebige Testfall-Zelle werden die fehlerhaften Testschritte aus dem Testlauf des zugehörigen Testfalls, falls in der Zelle vorhanden, im Bereich 4, der nachfolgend beschrieben wird, auf Fehler ausgewertet.

Bereich 3: Diagramm „Testdauertrend"

Bei der beliebigen Auswahl eines Testlaufes in dem Bereich 1 wird im Bereich 3 ein Testdauertrend-Diagramm in der Form eines Balken-Diagramms erstellt (Abbildung 23).

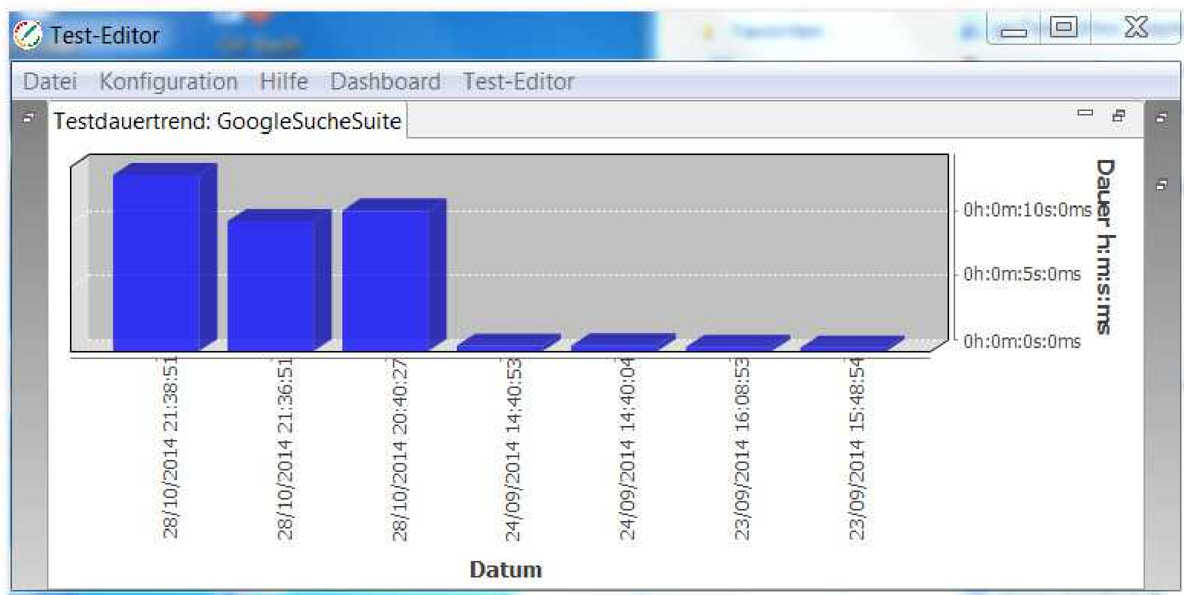


Abbildung 23: Diagramm „Testdauertrend" – Bereich 3

Die X-Achse beinhaltet die Ausführungszeitpunkte der bisherigen Testläufe. Diese sind auch wie im Bereich 2 beginnend mit dem aktuellsten Testlauf-Datum von links nach rechts sortiert. Die Y-Achse beinhaltet die Testdauer-Messungen im Format hh:mm:ss:ms. Bei einer Testsuite wird die gesamte Testlaufdauer aus den Dauern der ausgeführten Testfälle summiert. Jeder Balken zeigt somit die Dauer eines einzelnen Testlaufs.

Bereich 4: Tabelle „Fehler"

In diesem Bereich (Abbildung 24) werden die übergegebenen fehlerhaften Schritte in einer Tabelle angezeigt; Andernfalls bleibt die Tabelle leer.

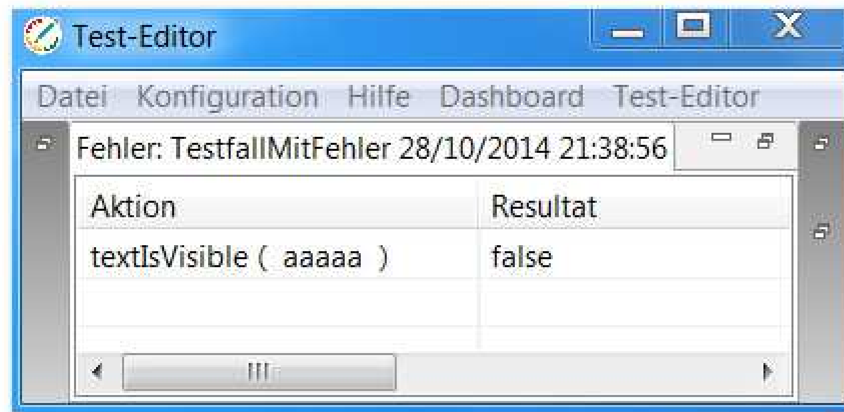


Abbildung 24: Tabelle „Fehler" – Bereich 4

Als Überschrift beinhaltet die Tabelle den Testfallnamen kombiniert mit dem Ausführungszeitpunkt des Testlaufs. Die erste Spalte beinhaltet die Abfragen (Aktionen) oder die Daten (Soll-Werte), die an das zu testende System gesendet wurden. Die zweite Spalte zeigt die zugehörigen System-Reaktionen oder die Ist-Werte (Resultate) an. Die fehlerhaften Schritte werden in den Zeilen, in der Reihenfolge der Ausführung beginnend, mit dem zuerst gestarteten Testschritt von oben nach unten sortiert.

4.5 Realisierung

4.5.1 Technologien

Für die Herstellung des Dashboards gelten die gleichen Entwicklungsumgebungen wie auch für den Test-Editor. Vor der Beschreibung der Implementierung werden zuerst die im Entwicklungsprozess verwendeten Technologien erläutert.

Die Implementierung erfolgt in einem Eclipse 4 Integrated Development Environment (IDE) Framework. „Eines der Kernkonzepte von Eclipse ist seine modulare Architektur. Sie erlaubt das Hinzufügen und Entfernen von Features. Diese Features werden als Komponenten in Eclipse „Plug-Ins" oder „Bundles" genannt – bereitgestellt" (TEUFEL u. HEIMING 2012, S. 20). Der Einsatz von Plug-Ins ermöglicht eine nahezu beliebige Erweiterung.

Die Abbildung 25 zeigt die beinhalteten Komponenten am Beispiel des Eclipse 4.2 Software Development Kits (SDK).

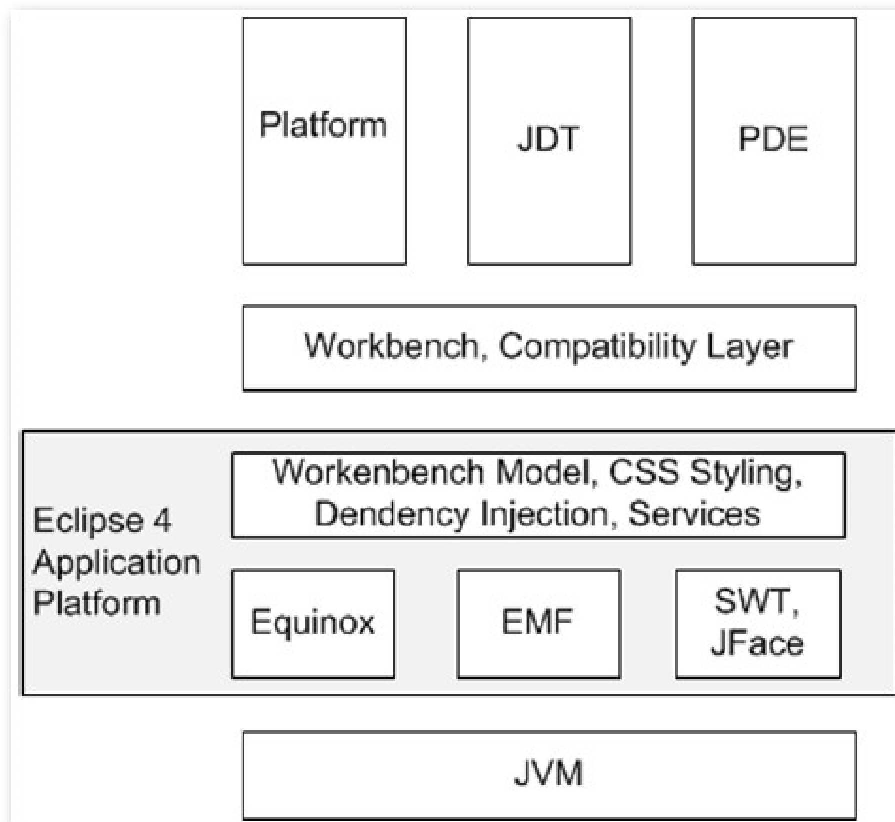


Abbildung 25: Das Eclipse 4.2 SDK besteht aus unterschiedlichen Komponenten (TEUFEL u. HEIMING 2012, S. 30)

Die unterste Schicht bildet die Java Virtual Machine (JVM). Darauf baut das SDK auf.

Direkt auf der JVM sitzt die Eclipse 4 Application Platform. Die Basiskomponenten dieser Plattform sind Equinox, EMF und SWT.

Das Eclipse Modeling Framework (EMF) wird verwendet, um das Application-Model im Speicher abzubilden. Die Änderungen der Bestandteile dieses Modells erfolgen zuerst im Modell und danach bringen die speziellen Renderer die Änderungen an die Oberfläche (vgl. TEUFEL u. HEIMING 2012, S. 29).

Ab der Version 3.0 wurde OSGi-Technologie (Open Services Gateway initiative) eingeführt. Den Kern von Eclipse bildet eine Komponente namens Equinox (OSGi-Framework), die Basis für alle Plug-Ins ist und Rich Client Platform (RCP) Paradigma bedient. Equinox verwaltet Abhängigkeiten zwischen Plug-Ins und kann diese je nach Bedarf laden, starten, beenden und sogar nachinstallieren. So erlaubt es die

Zusammenarbeit der verschiedenen Komponenten (vgl. TEUFEL u. HEIMING 2012, S. 21-22).

Das Standard Widget Toolkit (SWT) ist eine Komponentenbibliothek, die eine Verwendung von systemeigenen Widgets wie Textboxen, Comboboxen, Tables usw. ermöglicht. Das Toolkit JFace erweitert die Funktionen von SWT und kann somit aus den von SWT gelieferten Basiskomponenten komplexere Widgets zusammenstellen. Für eine leichtere Oberflächenerstellung stellt JFace Viewer und Wizards dar, die wiederum SWT-Widgets verwenden. Diese Kernkomponenten bilden die Basis für das Application-Model, den Dependency-Injection-Mechanismus, die Service-Schicht und das CSS-Styling (vgl. TEUFEL u. HEIMING 2012, S. 30).

Dependency Injection (DI) oder Inversion of Control (IoC) ist ein Konzept, bei dem nicht die Anwendung für die Bereitstellung eines Objektes verantwortlich ist, sondern die Fabrikmethoden des zugrundeliegenden IoC-Containers. Dies erfolgt über die Annotation *@Inject*, die man in Konstruktoren, auf Methoden und auch auf Felder setzen kann (vgl. TEUFEL u. HEIMING 2012, S. 32).

Weitere wichtige Bausteine von Eclipse 4 Application Platform sind unter anderem IEclipseContext, Core Services, das Application-Model, die Rendering Engine, Declarative Styling usw.

Der hierarchisch organisierte IEclipseContext (EclipseContext) steuert das Registrieren und Finden von Services, das Zwischenspeichern von Selections usw. (vgl. TEUFEL u. HEIMING 2012, S. 32).

Es ist möglich, in Eclipse über Dependency Injection auf unterschiedliche Standard-Services (Core Services) zuzugreifen. Es werden Services für das Logging, Eventhandling, zum Zugriff auf das Application-Model und vieles mehr angeboten.

Ein nicht vernachlässigbares Teil der Implementierung wird das Application-Model sein. Die Bestandteile wie Fenster, Buttons oder Menus sind in diesem Modell definiert und werden zur Laufzeit durch entsprechende Renderer erzeugt. Mit speziellen vom Framework bereitgestellten Services, beispielsweise dem *PartService* und den *ModelService*, lässt sich zur Laufzeit auf das interne Modell zugreifen und dieses bei Bedarf auch verändern.

Wie schon oben beschrieben, übernimmt die Rendering Engine zur Laufzeit das Rendern und Zeichnen der Oberfläche aus dem im Speicher gelegenen abstrakten Application-Model. Standardweise wird eine Rendering Engine für SWT mitgeliefert. Dieses Konzept macht das Eclipse 4 Framework von Oberflächenbibliotheken unabhängig (vgl. TEUFEL u. HEIMING 2012, S. 33).

Die Styling Engine, die keinen Bezug zum Application-Model hat, kann verwendet werden, um Schriften, Farben und andere Teile der Widget-Darstellung zu beeinflussen (vgl. TEUFEL u. HEIMING 2012, S. 34).

Nach der Application Platform kommt die Workbench mit den sogenannten Compatibility Layer. Dieser hilft die Aufrufe des Application Programming Interface (API), die noch auf 3.x basieren, zu übersetzen und somit ältere Plug-Ins lauffähig zu machen.

In der obersten Schicht des SDKs befinden sich die entsprechenden Plug-Ins, die die jeweiligen Entwicklungsumgebungen bereitstellen und für die Plug-In-Entwicklung benötigt werden. Für die Java-IDE werden die „Java Development Tools“ (JDT) und für die Plug-In-Entwicklung wird das „Plugin Development Environment“ (PDE) verwendet (vgl. TEUFEL u. HEIMING 2012, S. 31).

4.5.2 Implementierung

Die Abbildung 26 zeigt die Zusammenhänge der implementierten Klassen und die Dashboard-Elemente für diese, die Klassen verantwortlich sind. Im Weiteren wird die Implementierung der wichtigsten Elemente erläutert. Die kompletten Klassen-Codes sind im Anhang zu finden.

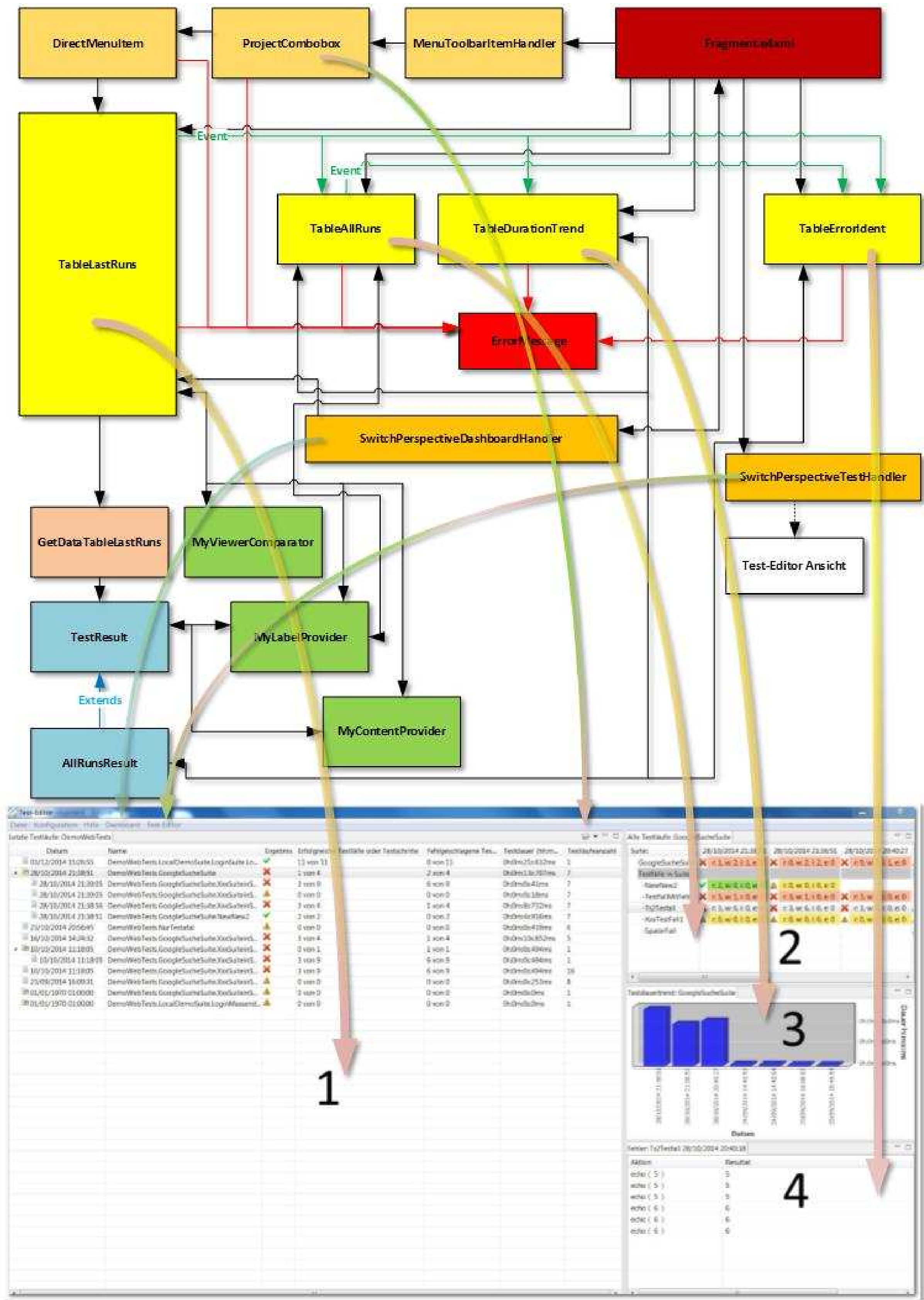


Abbildung 26: Zusammenhänge der Klassen und Dashboard-Elementen

4.5.2.1 Plug-In-Projekt und Fragment Model

Da das Dashboard als Plug-In implementiert wird, wurde ein neues Plug-In-Projekt mit dem Packagenamen *org.testeditor.dashboard* angelegt. Die Struktur des Test-Editors wird im Application-Model in der Datei *„Application.e4xmi“* definiert, das sich im Projekt *org.testeditor.ui* befindet. Für das Dashboard wird dazu im zugehörigen Projekt ein neues Fragment *fragment.e4xmi* erzeugt (Abbildung 27)

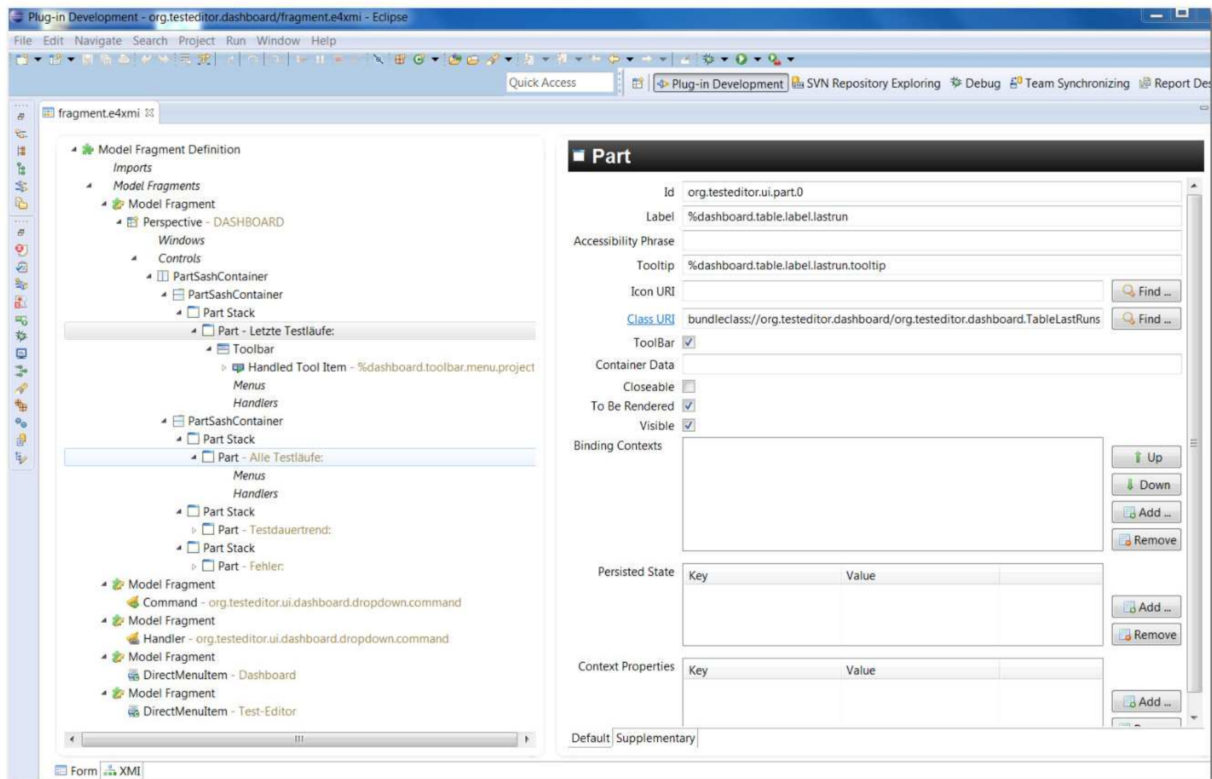


Abbildung 27: Fragment.e4xmi

In dieser Fragment-Datei werden die visuellen Bereiche des Dashboards festgelegt. Das Dashboard soll eine eigene Ansicht bekommen, dazu wird eine *Perspective* namens *DASHBOARD* angelegt.

In dem Test-Editor-Menü-Bereich werden die Umschalt-Buttons für die Dashboard- und Test-Editor-Ansicht zur Verfügung gestellt. Dies erfolgt mit den entsprechenden *DirectMenuItem*-Elementen. Die Buttons sind aktiv (*enabled*) und sind vom Typ *Push*, so dass man auf diese klicken kann. In dem *Class-URI*-Feld wird die zugehörige, beim Betätigten des Buttons anzusprechende Klasse definiert.

```
Dashboard-Perspective:
bundleclass://org.testeditor.dashboard/org.testeditor.dashboard.SwitchPerspectiveDashboardHandler
```

Test-Editor-Perspective: bundleclass://org.testeditor.dashboard/org.testeditor.dashboard.SwitchPerspectiveTestHandler

Die Bereiche des Dashboards müssen dem definierten Design entsprechend in der *Perspective* platziert werden. Dazu wird in der *Perspective* ein *PartSashContainer* (PSC1) mit vertikaler Aufteilung, der wiederum zwei weitere *PartSashContainer* PSC2 und PSC3 mit horizontaler Aufteilung beinhaltet, erstellt. So wird die Ansicht zuerst in zwei Bereiche geteilt. In dem PSC2 wird ein *Part Stack* erzeugt und dort wird ein *Part* für die Anzeige der Tabelle „Letzte Testläufe“ angelegt. In dem *Part Stack* des PSC3 werden in entsprechender Reihenfolge drei *Parts* für die Anzeigen der Tabelle „Alle Testläufe“, des Diagramms „Testdauertrend“ und der Tabelle „Fehler“ erstellt. Die Größen der *Parts* werden prozentual angepasst. Außerdem bekommen die *Parts* automatisch bei Bedarf einen Scroll-Bar und sind zur Laufzeit beliebig bezüglich der Größe und Position veränderbar.

Alle *Parts* enthalten das Class-URI-Feld (Uniform Resource Identifier-Feld). Diese Felder bekommen die entsprechenden Klassen, welche zur Laufzeit aufgerufen werden, um die Dashboard-Bereiche mit zugehörigen Elementen und Daten zu befüllen.

```
Tabelle “Letzte Testläufe”:
bundleclass://org.testeditor.dashboard/org.testeditor.dashboard.TableLastRuns
Tabelle “Alle Testläufe”:
bundleclass://org.testeditor.dashboard/org.testeditor.dashboard.TableAllRuns
Diagramm “Testdauertrend”:
bundleclass://org.testeditor.dashboard/org.testeditor.dashboard.TableDurationTrend
Tabelle “Fehler”:
bundleclass://org.testeditor.dashboard/org.testeditor.dashboard.TableErrorIdent
```

In dem Bereich „Letzte Testläufe“ wird in der Toolbar ein dynamisches Drop-Down-Menu (Abbildung 28) erstellt.

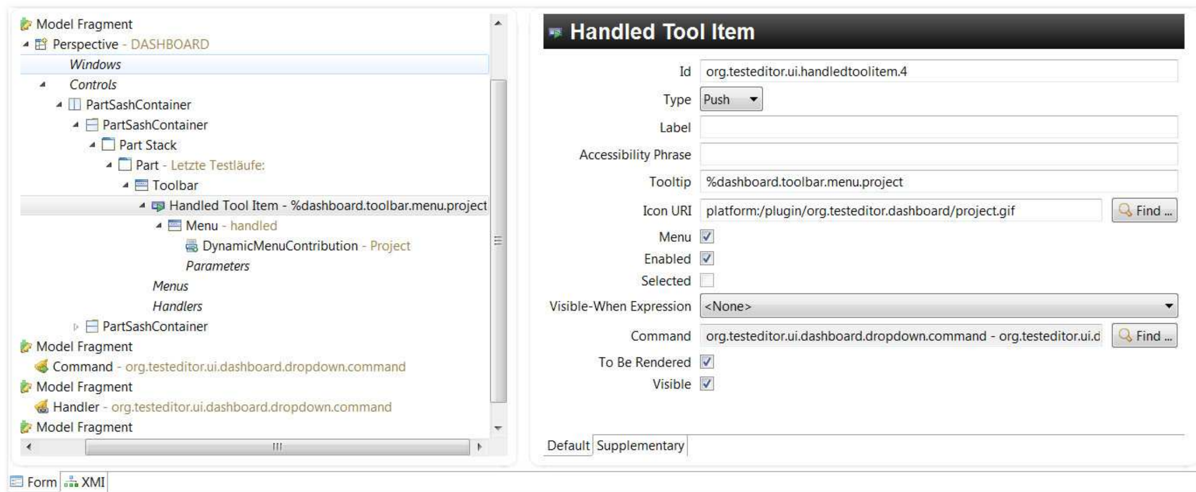


Abbildung 28: Drop-Down-Menu für Projekte fragment.e4xmi

Dieses Menu bekommt im Feld-Icon-URI einen zugehörigen Projekt-Icon zugewiesen, das im Dashboard angezeigt wird. Über ein *Command* wird ein *Handler* angestoßen, der wiederum die zugehörige Klasse aufruft. *Command* und *Handler* sind ebenfalls in der *fragment.e4xmi* hinterlegt.

```
bundleclass://org.testeditor.dashboard/org.testeditor.dashboard.MenuTollbarItemHandler
```

Jedes Element in der *fragment.e4xmi* bekommt eine Kennung - *ID*. Über die *IDs* können die Elemente, zum Beispiel in den Klassen, gefunden und angesprochen werden.

In der *MANIFEST.MF*-Datei werden unter anderem die Abhängigkeiten zu den integrierten Bundles und Packages definiert. Dort befinden sich zum Beispiel Packages wie *testeditor.ui* (für die Benutzer-Oberfläche des Test-Editors), *jfreechart* (für die Grafik in dem Bereich „Testdauertrend“), *w3c.dom* oder *jdom* (die für XML-Verarbeitung) usw. Alle Abhängigkeiten, die in *MANIFEST.MF* definiert sind, sind in dem Anhang 8 zu finden.

4.5.2.2 Umschaltung der Perspektiven

SwitchPerspectiveDashboardHandler, SwitchPerspectiveTestHandler

Wie bereits in der Datei *fragment.e4xmi* definiert, sind die Klassen *SwitchPerspectiveDashboardHandler* und *SwitchPerspectiveTestHandler* für die Umschaltung zwischen den Dashboard- und Test-Editor-Ansichten verantwortlich. Beim Umschalten zur Dashboard-Perspektive muss der Dashboard-Button im Menu-Bereich angeklickt werden. Zuerst wird mit Hilfe des *modelService* nach der

Perspektive-ID *„org.testeditor.ui.perspective.dashboard“* gesucht und, wenn diese Perspektive nicht bereits aktiv ist, wird es mit dem Befehl *„partservice.switchPerspective(dashboard);“* aktiviert. Dabei wird die Methode *„refresh“* zum Aktualisieren der Tabelle „Letzte-Testläufe“ ausgeführt. Diese Methode wird in der Klasse *TableLastRuns* genauer erläutert.

```
@Execute
public void execute(MApplication app, EPartService partService,EModelService modelService,
MWindow window,IEclipseContext context, @Named(IServiceConstants.ACTIVE_SHELL) final Shell
                    shell) throws JDOMException,IOException, ParseException, SystemException {
        MPerspective dashboard = (MPerspective)modelService.find
                                        ("org.testeditor.ui.perspective.dasboard" , app);
        if(!modelService.getActivePerspective(window).getLabel().equals("DASHBOARD")) {
                partService.switchPerspective(dashboard);
                TableLastRuns tab = (TableLastRuns) partService.findPart("org.testeditor.ui.part.0")
                                                                        .getObject();
                tab.refresh(null, modelService, window, app, context);
        }
}
```

Beim Umschalten zur Test-Editor-Perspektive wird nach dem Betätigen des Test-Editor-Buttons nach der Perspektive-ID *„org.testeditor.ui.perspective.testeditor“* gesucht und anschließend ebenfalls mit der *switchPerspective*-Anweisung umgeschaltet. Diese Umschalt-Klassen sind in dem Anhang 14 und 15 zu finden.

4.5.2.3 Projekt-Dropdown-Menu

MenuToolbarItemHandler

In der Eclipse Umgebung gibt es ein Problem mit der Aktivierung des Dropdown-Menüs: Obwohl es im *fragment.e4xmi* als *„Enabled“* eingestellt ist, bleibt es deaktiviert und kann nicht bedient werden. Um dieses Problem zu beheben, wird eine Umgehungslösung mit dem Handler *MenuToolbarItemHandler* (Anhang 9) und dem entsprechenden Befehl (Command), die bereits im 4.5.2.1 Plug-In-Projekt und Fragment Model erwähnt wurden, benötigt. Mit solch einem Handler wird das Dropdown-Menu aktiv.

```
@Execute
public void execute(final MApplication application, final EModelService modelService,
                        @Named("com.at.dropdowntoolbaritem.itemid") final String itemid) {
```

```
        MUIElement element = modelService.find(itemid, application);
        if (element != null) {
                ToolItem toolItem = (ToolItem) element.getWidget();
                Event event = new Event();
                event.type = SWT.Selection;
                event.detail = SWT.ARROW;
                toolItem.notifyListeners(SWT.Selection, event);
        }
}
```

Nach der Betätigung des dynamischen Dropdown-Menüs wird die Klasse *ProjectComboBox* aufgerufen.

ProjectComboBox

Die *ProjectComboBox*-Klasse (Anhang 13) ist für die Beschaffung der auswählbaren Elemente (Projekte) zuständig. Mit Hilfe des *testProjectService* werden die Namen aller existierenden Projekte mit dem *getProjects*-Befehl ausgelesen und in dem Auswahlbereich des Dropdown-Menüs angezeigt. Wenn keine Projekte vorhanden sind, wird eine entsprechende Fehlermeldung (Error-Message) angezeigt. Fehler- oder Warnung-Messages stellt die Klasse *ErrorMessage*, die später erläutert wird, zur Verfügung. Außerdem ist hier die *ContributionURI* gesetzt. Der durch den Benutzer ausgewählte Projekt-Name wird an diese Klasse zur weiteren Verarbeitung übergeben.

```
@AboutToShow
public void aboutToShow(List<MMenuElement> items, IEclipseContext context) {
        try {
                List<TestProject> projects = testProjectService.getProjects();
                for (int i = 0; i < projects.size(); i++) {
                        MDirectMenuItem element =
                                        MMenuFactory.INSTANCE.createDirectMenuItem();
                        String projectName = projects.get(i).getName();
                        element.setLabel(projectName);
                        element.setElementId(projectName);
                        element.setContributionURI
                  ("bundleclass://org.testeditor.dashboard/org.testeditor.dashboard.DirectMenuItem");
                        items.add(i, element);
                }
        } catch (SystemException e) {
```

```
                    ErrorMessage error = ContextInjectionFactory.make(ErrorMessage.class, context);
                    error.errorProject();
          }
}
```

DirectMenuItem

Die *DirectMenuItem*-Klasse (Anhang 5) baut den kompletten Pfad zum ausgewählten Projekt und überprüft, ob dieses Testfall- oder Testsuite-Ordner beinhaltet. Falls solche Ordner vorhanden sind und die Testergebnisse von diesem Projekt nicht bereits in der Tabelle „Letzte Testläufe“ dargestellt sind, wird der Projektname an die Methode *refresh* aus der *TableLastRuns*-Klasse übergeben. Andernfalls ist das Projekt leer, die *refresh*-Methode wird nicht ausgeführt und es wird eine entsprechende Message aus der *ErrorMessage*-Klasse ausgegeben.

```
File dir=new File(Platform.getLocation().toFile()+"\\"+projectName+
                                                        "\\FitNesseRoot\\files\\testResults");
if (GetDataTableLastRuns.searchFile(dir)) {
          if (!mPart.getLabel().equals(translationService.translate ("%dashboard.table.label.lastrun",
                                                        CONTRIBUTOR_URI) + " "+ projectName)) {
                    tab.refresh(projectName, modelService, window, app, context);
          }
} else {
          ErrorMessage error = ContextInjectionFactory.make(ErrorMessage.class, context);
          error.errorProjectEmpty();
}
```

4.5.2.4 Tabelle „Letzte Testläufe“

TableLastRuns

Die *TableLastRuns*-Klasse (Anhang 19) baut und befüllt die Tabelle „Letzte Testläufe“ mit den Testergebnissen der letzten Testläufe.

Mit dem *swt.widgets.Tree*-Objekt wird die Tabelle für die anzuzeigenden Testergebnisse in einer Baumstruktur definiert.

```
final Tree tree = new Tree(parent, SWT.BORDER | SWT.FULL_SELECTION | SWT.H_SCROLL |
                                                        SWT.V_SCROLL);
tree.setHeaderVisible(true);
tree.setLinesVisible(true);
```

Die Spalten (swt.widgets.TreeColumn-Objekte) werden in einer *for*-Schleife erzeugt und bekommen die Namen, Tooltips, Größen und die Layouts zugewiesen.

```
for (int i = 0; i < titles.length; i++) {
        TreeColumn column = new TreeColumn(tree, SWT.CENTER);
        column.setText(titles[i]);
        column.setAlignment(SWT.LEFT);
        if (i == 0) {
                column.setWidth(200);
                column.addSelectionListener(getSelectionAdapter(column, 0)); // new
                column.setToolTipText(translationService.translate(
                        "%dashboard.table.label.lastrun.column.tooltip.date",CONTRIBUTOR_URI));
        }
}
```

Für die komfortable Nutzung von *Content Provider* und *Label Provider*, die die Daten an die angezeigte Tabelle übermitteln, wird ein Objekt der TreeViewer-Klasse injiziert.

```
@Inject
private TreeViewer v;
```

Unter anderem macht der *TreeViewer* auch eine Anbindung von Sortern und Filtern möglich. Die dem Inhalt entsprechende Sortierung der Spalten-Elemente wird über die *MyViewerComparator*-Klasse realisiert (Anhang 12).

```
comparator = new MyViewerComparator();
v.setComparator(comparator);
```

In der Methode *getSelectionAdapter* werden beim Anklicken des Headers einer beliebigen Spalte die Sortierungen in dem *MyViewerComparator* durchgeführt und die Baumansicht anschließend mit der neuen Sortierreihenfolge aktualisiert.

```
private SelectionAdapter getSelectionAdapter(final TreeColumn column, final int index) {
        SelectionAdapter selectionAdapter = new SelectionAdapter() {
                @Override
                public void widgetSelected(SelectionEvent e) {
                        comparator.setColumn(index);
                        int dir = comparator.getDirection();
                        v.getTree().setSortDirection(dir);
                        v.getTree().setSortColumn(column);
                        v.refresh();
```

```
                }
        };
        return selectionAdapter;
}
```

Über die *MyContentProvider*-Klasse (Anhang 10) werden die *TestResult*-Objekte als Parent- oder Child-Elemente an den *TreeViewer* gereicht. Die *MyLabelprovider*-Klasse (Anhang 11) setzt die Testergebnisse als *Strings*, die zugehörigen Icons usw. in die Zellen ein. Diese Klassen werden später beschrieben.

```
v.setLabelProvider(new MyLabelProvider());
v.setContentProvider(new MyContentProvider());
```

Die Tabelle benötigt einen *MouseEvent*-Listener, damit bei Auswahl einer Zeile, die einen Testlauf beinhaltet, der Name des Testfalls oder der Testsuite an die Tabelle „Alle Testläufe“ geleitet werden kann.

```
public void mouseDoubleClick(MouseEvent e) {
        String fileName = ((Tree) e.getSource()).getSelection()[0].getText(1);
        eventBroker.send("FileName", fileName);
```

Dieser Event wird mit Hilfe eines injizierten *EventBrokers* gesendet. Dabei wird der Testlauf-Name genommen und als Event mit dem eindeutigen Event-Namen gesendet. Die Klassen, die auf dieses Event warten, empfangen es und führen die zugehörigen Operationen nach dem Observer Pattern (Beobachter-Muster) aus.

Zur Ladung oder Aktualisierung der Daten in der Tabelle wird in der Methode *refresh* die Klasse *GetDataTableLastRuns* aufgerufen. Die Aktualisierung der Tabelle geschieht immer bei der Erstladung und bei dem Projektwechsel, dabei müssen die anderen Dashboard-Bereiche geleert werden, da diese sonst mit dem neuen Projekt nicht übereinstimmen werden. Deswegen werden gleichzeitig die Events an die *TableAllRuns*-, *TableDurationTrend*-, *TableErrorIdent*-Klassen gesendet und dort entsprechend verarbeitet.

```
public void refresh(String projectName, EModelService modelService, MWindow window,
MApplication app,IEclipseContext context) throws JDOMException, IOException, ParseException,
                                                                        SystemException {
        String string = "x";
        GetDataTableLastRuns x = ContextInjectionFactory.make(GetDataTableLastRuns.class,
                                                                                context);
        v.setInput(x.getData(projectName, modelService, window, context, app));
```

```
        eventBroker.send("DisposeAllRunsResultTable", string);
        eventBroker.send("DisposeErrorTable1", string);
        eventBroker.send("DisposeChartTable", string);
}
```

GetDataTableLastRuns

In der *GetDataTableLastRuns*-Klasse (Anhang 7) werden die Daten der Test-ergebnisse ausgelesen und an den *TreeViewer* zur Darstellung in der Tabelle „Letzte Testläufe" übergeben.

Als erstes wird es in der Methode *getData* durch das Abfragen sichergestellt, dass das zu untersuchende Projekt die Testläufe mit den zugehörigen Ergebnissen enthält. Wenn keine Probleme auftreten, wird ein neues Label mit dem entsprechenden Projekt-Namen für die Tabelle gesetzt. Im anderen Fall kommt eine Message, dass das Projekt leer ist. Beim Öffnen des Dashboards wird die Tabelle automatisch mit den Daten aus dem ersten Projekt geladen. Dazu sind alle Projekte in einer Liste gespeichert.

```
List<TestProject> projects = testProjectService.getProjects();
```

Beim Öffnen eines neuen Projektes über das Dropdown-Menu wird, wie schon beschrieben, der ausgewählte Projekt-Name genommen und, wenn dieser nicht leer ist, als Label für die Tabelle verwendet.

```
if (projectname != null) {
        testResultDirectroy = new File(Platform.getLocation().toFile() + "\\" + projectname
                                                  + "\\FitNesseRoot\\files\\testResults");
        mPart.setLabel(translationService.translate("%dashboard.table.label.lastrun",
                                                  CONTRIBUTOR_URI) + " "+ projectname);
}
```

Alle Testfälle und Testsuiten werden alphabetisch sortiert, in einem Array *testResultFoldersContent* vom Typ *File* gespeichert und an die Methode *getDataFromXMLResultFiles* übergeben.

```
File[] testResultFoldersContent = testResultDirectroy.listFiles();
Arrays.sort(testResultFoldersContent);
root = getDataFromXMLResultFiles(root, testResultDirectroy, testResultFoldersContent);
```

In der Methode *getDataFromXMLResultFiles* wird in einer *for*-Schleife aus dem *testResultFoldersContent*-Array der aktuellste Testlauf von jedem Testfall und jeder

Testsuite genommen und abhängig davon, ob es ein Testfall oder eine Testsuite ist, an die Methoden *suiteGetResults* bzw. *getPreconditionToRetrieveTestCaseData* gesendet.

Laut den Anforderungen werden nur letzte Testläufe der Testsuiten und Testfällen angezeigt. Bei einer Suite es ist einfacher, die aktuellste Datei auszuwählen. Die Testsuiten-Ergebnisdateien werden dazu in der Methode *getLastModified* sortiert und die aktuellste Datei wird sofort zur Auswertung in *suiteGetResults* übergeben. Ein Testfall erfordert mehr Aufwand. Es muss sichergestellt werden, dass sein aktuellster Testfalllauf zu keinem Testsuitelauf gehört, nur dann können die Ergebnisse aus diesem Testfall in die Tabelle übernommen werden. Ist dies nicht der Fall, so wird der vorgehende Testfalllauf analysiert. Der genaue Algorithmus wird weiter unten beschrieben.

In *getPreconditionToRetrieveTestCaseData* wird zuerst überprüft, an welcher Stelle der Testfall im *testResultFoldersContent* steht. Wie bereits erklärt, sind die Namen hierarchisch aufgebaut, was bedeutet, dass der erste Testfall im Array zu keiner Testsuite gehört. Seine aktuellste Testergebnisdatei kann man sofort an die Methode zur Testfall-Datenauswertung *testfallGetResults* übergeben. Alle anderen Dateien muss man weiter in der Methode *compareLastRunParentSuiteLastRunTestCase* untersuchen, um mögliche Parent-Testsuiten zu finden.

In der Methode *compareLastRunParentSuiteLastRunTestCase* werden die Ordner-namen verglichen.

```
if (fileName.startsWith(testResultFoldersContent[k].getName())
                                        && !fileName.equals(testResultFoldersContent[k].getName())) {
parentSuite = true;
```

Existiert eine Parent-Testsuite nicht, so wird die aktuellste Testfall-Ergebnisdatei in der Methode-*testfallGetResults* ausgewertet. Wenn eine Parent-Testsuite gefunden wurde, werden die Testfall- und Parent-Testsuite-Ergebnisdateien weiter in der *compareTimestamps*-Methode verarbeitet.

In der *compareTimestamps*-Methode werden die Testergebnisdateien von Parent-Testsuite und Testfall in die entsprechenden Arrays nach Datum sortiert abgelegt und in einer Schleife mit einander bezüglich des Ausführungsdatums verglichen. Die Ausführungszeitpunkte sind im Dateinamen sowie in der Datei selbst gespeichert. Es

ist wichtig zu bemerken, dass der Zeitpunkt eines Testfalls sich beim Vergleich der Ausführungszeitpunkte aus den Dateinamen von dem Zeitpunkt des Testsuitelaufs unterscheiden kann, obwohl diese zusammen ausgeführt wurden. Die Startzeit der Testsuite wird bei dem Testsuitelauf im Dateinamen gespeichert Die exakten Teststartzeiten der mit ausgeführten Testfälle befinden sich dabei in der Datei und unterscheiden sich untereinander sowie von der Testsuite Startzeit, da die Testfälle nicht gleichzeitig ausgeführt wurden. Mit einem XML-Parser wird die Datei ausgelesen und die entsprechenden Daten werden verglichen. Die daraus folgende Übereinstimmung der Zeitangaben bedeutet, dass der analysierte Testfalllauf in dieser Parent-Suite ausgeführt wurde und nicht in der Tabelle angezeigt werden sollte. Die Testfall-Ergebnisdateien, die aus einem Testsuitelauf stammen, werden dabei nach und nach aus dem Array gelöscht.

```
for (int m = 0; m < docChildren.size() && e < docpageHistoryReference.size(); m++) {
        Element pageHistoryReference = (Element) docChildren.get(m);
        // compare testcase from suite with searched testcase
        if (pageHistoryReference == docpageHistoryReference.get(e)) {
                String nameinxml = pageHistoryReference.getChild("name").getValue();
                String nametestfall = dirSearch.getName();
                // test case in suite found
                if (nameinxml.equals(nametestfall)) {
                        int timestampXMLlength = pageHistoryReference.getChild
                                                        ("pageHistoryLink").getValue().length();
                        String timestampXML = pageHistoryReference.getChild  ("pageHistoryLink").
                              getValue().substring(timestampXMLlength - 14, timestampXMLlength);
                        // test case time stamp equals suite time stamp, delete it from list
                        if (timestampXML.equals(testfallDate)) {
                                lastRunFileList[t] = null;
                                s = filesByDateParentSuiteLength;
                                break;
                        }
                }
                e++;
        }
}
```

Der gefilterte *lastRunFileList*-Array, der nun nur die von einer Testsuite unabhängigen Testlaufergebnisdateien beinhaltet, wird zurück an die Methode

compareLastRunParentSuiteLastRunTestCase übergeben. Dort wird aus den zurückgebliebenen Dateien die aktuellste ausgewählt und zur Auswertung an die Methode *testfallGetResults* geleitet.

In der Methode *testfallGetResults* erfolgt schlussendlich die Auslesung der Testergebnisse und testrelevanten Daten. Es wird ein Objekt der *TestResult*-Klasse erzeugt und dort werden die Daten mittels *set*-Befehls gesetzt.

```
String testfallDate = lastModFile.getName().substring(0, 14);
Date date = new SimpleDateFormat("yyyyMMddHHmmss").parse(testfallDate);
String formattedDate = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss").format(date);
String right = docChildResult.getChild("counts").getChild("right").getValue();
int rightint = Integer.parseInt(right);
…
TestResult testResult = new TestResult();
testResult.setTestcase(true);
testResult.setDate(formattedDate);
testResult.setQuantityRight(rightint);
…
```

Folgende Daten werden ausgelesen bzw. gesetzt:

- Testfall Name;
- Formatierte Testausführungsdatum und Testlaufdauer;
- Anzahl von fehlerhaften, richtigen, ignorierten Testschritten, Ausnahmen und allen bisherigen Testläufen;
- Zusammensetzung der Ergebnisse für die Spalten „Erfolgreiche Testschritte“ und „Fehlgeschlagene Testschritte“;
- Testresultat.

Das Testresultat wird für die Ergebnis-Spalte nach folgendem Muster berechnet.

```
if (wrongInt + exceptionsInt + ignoresInt == 0 && rightint > 0) {
        testResult.setResult("ok");
}
if (wrongInt > 0 || exceptionsInt > 0 || ignoresInt > 0) {
        testResult.setResult("failed");
}
if (rightint + wrongInt + ignoresInt + exceptionsInt == 0 || wrongInt + exceptionsInt + rightint == 0
                                                                        && ignoresInt > 0) {
```

```
        testResult.setResult("warning");
}
```

Die oben beschriebenen Testsuite-Ergebnisdateien werden in der Methode *suiteGetResults* ausgewertet. Die Auswertung erfolgt in ähnlicher Weise wie in der Methode *testfallGetResults*. Hier werden zusätzlich zu den Testsuite-Testlaufdaten die Daten der beinhalteten Testfälle ausgewertet und zur Testsuite als Child-Element hinzugefügt. Zusätzlich wird noch die Testsuite-Testlaufdauer aus den einzelnen Testfalllaufzeiten summiert.

Die ausgelesenen Daten werden über die *MyContentProvider*- und *MyLabelProvider*-Klassen in den *TreeViewer* geleitet und dem Benutzer in den entsprechenden Zellen angezeigt.

MyViewerComparator

In der Klasse *MyViewerComparator* (Anhang 12) wird die Sortierung der Spalten-elemente realisiert. Es wird ein Index der ausgewählten Spalte übergeben und der Komparator sortiert die Elemente bezüglich deren Typ alphabetisch oder numerisch nach Datum.

MyLabelProvider

Die Klasse *MyLabelProvider* (Anhang 11) ist zuständig für die Übergabe von Bildern und Texten an den *TreeViewer*. Hier werden die im Design definierten Icons für die Spalten „Datum“ und „Ergebnis“ zur Verfügung gestellt. Außerdem übergibt *MyLabelProvider* die Daten von den *TestResult*-Objekten an die zugehörigen Spalten und definiert die Färbung der Zeilen. Die Testlaufdauer ist in den Ergebnisdateien im Millisekunden-Format gespeichert und wird für die Dashboard-Anzeige in das Format hh:mm:ss:ms umgewandelt.

```
private String formatDuration(int runTimeInMillisInt) {
        Time time = new Time(runTimeInMillisInt);
        Calendar cal = Calendar.getInstance();
        cal.setTime(time);
        String durationFormat = cal.get(Calendar.HOUR) - 1 + "h:" + cal.get(Calendar.MINUTE) + "m:"
                        + cal.get(Calendar.SECOND) + "s:" + cal.get(Calendar.MILLISECOND) + "ms";
        return durationFormat;
}
```

MyContentProvider

Der *TreeViewer* benötigt einen strukturierten *ContentProvider* (Anhang 10), der typischerweise das *ITreeContentProvider*-Interface implementiert, und dient dazu, die TestResult-Objekte in die Baumstruktur zu transformieren. Es definiert zum Beispiel, welche Elemente auf und zu klappbar sind und steuert dies auch.

AllRunsResult

In der *AllRunsResult*-Klasse (Anhang 1) werden die Attribute der Testergebnis-Objekte definiert. Es sind zum Beispiel: Test-Name, Pfad zur Testergebnisdatei, Testlaufdauer, Anzahl der Testläufe usw.

TestResult

Die *TestResult*-Klasse (Anhang 20) erbt von der *AllRunsResult*-Klasse und definiert noch weitere Attribute: das Testergebnis-*result* und eine *childs*-Liste für den *MyContentProvider*, welche die Testlaufdaten beinhaltet.

4.5.2.5 Tabelle „Alle Testläufe“

TableAllRuns

In der *TableAllRuns*-Klasse (Anhang 16) wird die Tabelle für den Dashboard-Bereich „Alle Testläufe“ gebaut und mit Daten befüllt.

Beim Umschalten der Projekte wird an die *TableAllRuns*-Klasse ein Event *DisposeAllRunsResultTable* gesendet. Dementsprechend wird die Tabelle, falls die etwas bereits anzeigt, geleert.

Beim Doppel-Klick auf einen beliebigen Testfall oder eine Testsuite in der Tabelle „Letzte Läufe“ wird ein Event *FileName* an diese Klasse gesendet. Zusätzlich wird der Ordner-Name, der die Testergebnisdateien beinhaltet, mit übergeben. Es ist generell zwischen einer Tabelle für die Testfallläufe (TF-Tabelle) und für Testsuiteläufe (TS-Tabelle) zu unterschieden. Die Aufgabe ist es, alle zugehörigen Testläufe zu analysieren und anzuzeigen. Nach der Prüfung, dass Ordner und Testergebnisse existieren, wird Bereichs Label mit dem zugehörigen Testlaufnamen und dem Tooltip versehen, und die Tabelle wird erstmals geleert. Bei nicht erfolgreicher Prüfung werden die entsprechenden Fehlermeldungen aus der *ErrorMessage*-Klasse angezeigt und die Tabelle behält den ursprünglichen Zustand.

Es werden zwei Listen angelegt, die die *AllRunsResult*-Objekte beinhalten werden. Für die Testfälle wird die *objektlist*-Liste und für die Testsuiten die zweidimensionale *objektListSuiteTests*-Liste verwendet. Bei den Testsuiten werden in der *objektListSuiteTests*-Liste zusätzlich zu den Testsuiteläufen auch jeweils die zugehörigen Testfallläufe gespeichert.

```
final List<AllRunsResult> objektList = new ArrayList<AllRunsResult>();
final ArrayList<List<AllRunsResult>> objektListSuiteTests = new ArrayList<List<AllRunsResult>>();
```

Entsprechend dem Design und abhängig von der Auswahl in der Tabelle „Letzte Läufe“ wird die TF-Tabelle unterschiedlich zur TS-Tabelle aufgebaut. In die TS-Tabelle wird eine extra Spalte zum Anzeigen der Namen eingefügt und es werden unterschiedliche Tooltips verwendet.

In der Methode *setAllRunResultData* fängt die Befüllung der Listen an. Es werden zuerst Pfad und Name des Testlaufs in den *AllRunsResult*-Objekt sowie die formatierten Testlaufszeitpunkte von allen zugehörigen Testläufen in die Titelliste für die Spalten gesetzt.

Nach der Identifizierung wird die Testsuite zur Auswertung an die Methode *retrieveSuiteData* übergeben, der Testfall an *retrieveTestCaseData*. Dort werden alle zugehörigen Testläufe, ähnlich wie in der *GetDataTableLastRun*-Klasse, untersucht und als Objekte mit allen testrelevanten Daten in den entsprechenden Listen gespeichert.

Die Methode *setFirstRowResults* wird für die Befüllung der TF-Tabelle verwendet. Die Methode *setSuiteResultsInTable* befüllt die TS-Tabelle. In der TF-Tabelle werden die Testergebnisse mit dem zugehörigen Icon in einer *for*-Schleife in die einzige Zeile geladen. Für die Icons wird der gleiche *MylabelProvider* verwendet.

```
// SUITE Failed
if (objektList.get(m).isFailed()) {
        item.setBackground(j + 1, new Color(Display.getDefault(), 255, 182, 153));
        item.setText(j + 1, objektList.get(m).getresultSummary());
        item.setImage(j + 1, MyLabelProvider.getImage("/failed.png"));
}
```

In der Methode *setSuiteResultsInTable* wir die erste Zeile für die Testsuite-Ergebnisse mit der Methode *setFirstRowResults* befüllt. Die zweite Zeile bekommt

eine Überschrift in der ersten Spalte und wird grau gefärbt. Das ist die Trennlinie zwischen den Testsuite-Ergebnissen und den beinhalteten Testfall-Ergebnissen.

Aus der ersten Testergebnisdatei werden die Namen der Testfälle in die erste Spalte und die zugehörigen Ergebnisse über die *setResultsSuiteTestCases*-Methode in die zweite Spalte gesetzt.

```
if (j == 0) {
        TableItem subitem = new TableItem(table, SWT.NONE);
        subitem.setText(j, "  -" + getLastName(objektListSuiteTests.get(j).get(i).getTestCaseName()));
        setResultsSuiteTestCases(objektListSuiteTests, j, i, subitem);
}
```

Bei den nächsten Spalten, die für die anderen Testläufe stehen, muss jetzt immer wieder überprüft werden, ob die Testfallnamen aus den anderen Testläufen bereits in der ersten Spalte aufgelistet sind oder nicht.

```
String testname = getLastName(objektListSuiteTests.get(j).get(i).getTestCaseName());
boolean testnameExists = false;
int k = 0;
for (k = 0; k < table.getItemCount(); k++) {
        if (table.getItem(k).getText().substring(3).equals(testname)) {
                testnameExists = true;
                break;
        }
}
```

Wenn der Testfallname noch nicht eingetragen ist, wird eine neue *subitem*-Zeile für diesen Testfall erzeugt. Der Name wird eingetragen und die Ergebnisse werden über die Methode *setResultsSuiteTestCases* in die entsprechende Zelle übergeben.

```
if (!testnameExists) {
        TableItem subitem = new TableItem(table, SWT.NONE);
        subitem.setText(0, "  -" + getLastName(objektListSuiteTests.get(j).get(i).getTestCaseName()));
        setResultsSuiteTestCases(objektListSuiteTests, j, i, subitem);
}
```

Wenn der Testfallname schon existiert, wird das Testlaufergebnis in seiner Zeile in die zum Testlauf zugehörige Zelle eingetragen.

```
if (testnameExists) {
        if (objektListSuiteTests.get(j).get(i).isFailed()) {
```

```
                table.getItem(k).setBackground(j + 1, new Color(Display.getDefault(), 255, 182, 153));
                table.getItem(k).setImage(j + 1, MyLabelProvider.getImage("/failed.png"));
                table.getItem(k).setText(j + 1, objektListSuiteTests.get(j).get(i).getresultSummary());
        }
}
```

Nachdem alle Daten in die Tabelle geladen wurden, werden die Events an andere Dashboard-Bereiche versendet. Der Bereich für den Testdauertrend wird zunächst geleert und dann bekommt die *TableDurationTrend*-Klasse die hier bereits verwendete Liste mit den *AllRunsResult*-Objekten, die die Daten von allen Testläufen beinhalten. Die Tabelle „Fehler“ bekommt ebenfalls ein Event und wird geleert, da sonst die eventuell dort angezeigten Fehler nicht mit der aktuellen Anzeige der Tabelle „Alle Testläufe“ übereinstimmen.

Die Methode *sendDataToErrorTable* implementiert für die Tabelle einen *MouseDoubleClick-Listener.* So wird die Selektierung eines Testfallergebnisses in einer Zelle ermöglicht. Bei der TS-Tabelle werden nur die Klicks auf die Testfallergebnis-Zellen angenommen, danach wird ein Pfad zum ausgewählten Testfall gebaut und anschließend als Event an die Tabelle „Fehler“ gesendet. In der TF-Tabelle wird aus der *AllTestResult*-Objekt-Liste das der Spaltennummer entsprechende Element genommen und ebenfalls als Event an die Tabelle „Fehler“ gesendet.

```
if (i >= 0 && objektListSuiteTests.isEmpty()) {
        if (objektList.get(i).getTestResultFilePath().exists()) {
                eventBroker.send("Testobject", objektList.get(i));
                objectSend = true;
                break;
        }
}
```

4.5.2.6 Diagramm „Testdauertrend“

TableDurationTrend

Die Klasse *TableDurationTrend* (Anhang 17) ist für die Darstellung der Testdauertrends als Balken-Diagramm zuständig. Die Anzeige wird beim Projektwechsel geleert. Beim Doppelklick auf einen Testfall oder eine Testsuite in der Tabelle „Letzte Testläufe“ werden die Daten in der *TableAllrunsTable*-Klasse

verarbeitet. Danach wird die Liste mit den *AllRunsResult*-Objekten mit dem Event an die *TableDurationTrend*-Klasse übergeben. Diese Liste enthält dann alle bisherigen Testläufe und zugehörigen Daten von dem ausgewähltem Testfall bzw. Testsuite. Wie auch in anderen Dashboard-Bereichen wird der Header mit dem Testlaufnamen versehen.

Für die Realisierung eines Diagramms wird eine Open-Source-JFreeChart-Bibliothek verwendet, welche eine Unterstützung für verschiedene Diagramme, Tabellen, Grafiken usw. bietet. In der *createControls*-Methode werden die Grafikeigenschaften definiert wie zum Beispiel Achsenbeschriftung, Tooltips, Orientierung der Grafik, Achsenposition usw.

```
chart = ChartFactory.createBarChart3D(null, // chart title
translationService.translate("%dashboard.table.label.duration.axis.dates", CONTRIBUTOR_URI),
                                                              // domain X axis label
translationService.translate("%dashboard.table.label.duration.axis.duration", CONTRIBUTOR_URI)
                                                    + " h:m:s:ms", // rangeY axis label
createDataset(objektList), // data
PlotOrientation.VERTICAL, // orientation
false, // include legend
true, // tooltips?
false // URLs?
);
// get a reference to the plot for further customisation...
final CategoryPlot plot = chart.getCategoryPlot();
plot.setRangeAxisLocation(0, AxisLocation.BOTTOM_OR_RIGHT); // y axis right
```

Die Reihenfolge der Testläufe in dem Diagramm muss mit der Reihenfolge derer in Tabelle „Alle Testläufe“ übereinstimmen. Die Daten werden, beginnend mit dem Aktuellsten, von rechts nach links angezeigt. Deswegen ist die Y-Achse nach rechts verlegt und zeigt so den Anfang des Diagramms, wo der älteste Testlauf steht. Um dies zu erreichen, müssen die Daten für die Achsen in der richtigen Reihenfolge eingelesen werden. Dies geschieht in der Methode *createDataset*.

Die Methode *createDataset* bezieht die Daten aus der übergebenen Liste und stellt diese in der richtigen Reihenfolge in die zugehörigen Arrays. In dem *durationValues*-Array wird die Testdauer einzelner Testläufe gespeichert und in dem *dates*-Array werden die Testlauf-Zeitpunkte gespeichert.

```
for (int i = 0; i < datesToLoad; i++) {
        durationValues[i] = objektList.get(i).getDuration();// Duration column keys
        dates[i] = objektList.get(i).getDate();// Date column keys
        dataset.addValue(durationValues[i], series, dates[i]); // create the dataset...
}
return dataset;
```

Dementsprechend werden die Daten später in die zugehörigen Achsen geladen. Die Y-Achse bekommt den *dates*-Array, die X-Achse – den *durationValues*-Array.

Für die Färbung nutzt der JFreeChart eine *awt.color*-Klasse *(Abstract Windowing Toolkit)*. Es müssen aber die SWT-Farben verwendet werden. Damit die verwendete SWT-Farbe „Blue" nicht verfälscht wird, wird diese Farbe in der Methode *toAwtColor* in Rot-, Grün- und Blau-Anteile zerlegt, aus denen dann die übereinstimmende SWT-Farbe zusammengesetzt wird.

```
Color color = toAwtColor(ColorConstants.COLOR_BLUE);
…
public static java.awt.Color toAwtColor(org.eclipse.swt.graphics.Color color) {
        return new java.awt.Color(color.getRed(), color.getGreen(), color.getBlue());
}
```

4.5.2.7 Tabelle „Fehler"

TableErrorIdent

In der *TableErrorIdent*-Klasse (Anhang 18) werden nach dem Selektieren eines Testfall-Ergebnisses die zugehörigen fehlerhaften Schritte in der Tabelle ausgegeben. Wie bereits beschrieben, wird dazu ein Pfad zu der zu untersuchenden Datei von der *TableAllruns*-Klasse in einem Event bereitgestellt. Die Tabelle wird unter zwei Bedingungen durch das Empfangen eines Events geleert, und zwar durch den Projektwechsel oder das Testlauf-Selektieren in der Tabelle „Letzte Testläufe".

Nach dem Empfangen der Testlaufdatei wird in der Methode *getEvent* eine Überschrift für den zugehörigen Bereich generiert, welche aus dem Testlauf-Namen und -Datum besteht. Es wird eine Tabelle in der gleichen Art wie in der *TableAllRuns*-Klasse mit zwei Spalten „Aktion" und „Resultat" erzeugt.

Diese Tabelle wird in der Methode *searchErrorInXML* mit den Ein- und Ausgaben des Systems befüllt. Die XML-Testergebnisdatei beinhaltet alle Aktionen und

Reaktionen des Systems. Da für die Anzeige nur die fehlerhaften Schritte benötigt werden, wird diese Datei in einer *for*-Schleife nach dem *wrong*-Status durchsucht. Es werden die gefilterten Instruktionen (Aktion) und Resultate in den entsprechenden Zellen angezeigt.

```
if (nodeStatus.getTextContent().equals("wrong")) {
        TableItem item = new TableItem(table, SWT.NONE);
        String[] arr = instructionResultNodeList.item(1).getTextContent().split("\\,");
        List<String> list = new ArrayList<String>(Arrays.asList(arr));
        list.remove(0);
        list.remove(0);
        list.remove(0);
        if (list.get(0).equals(" scriptTableActor")) {
                list.remove(0);
        }
        list.add(1, "(");
        list.add(list.size(), ")");
        String message1 = list.toString();
// replace starting "[" and ending "]" and ","
        message1 = message1.substring(1, message1.length()).replaceAll(",", "");
        message1 = message1.substring(1, message1.length()).replaceAll("]", " ");
        item.setText(0, message1);
        item.setText(1, instructionResultNodeList.item(3).getTextContent());
        errorFound = true;
}
```

4.5.2.8 Andere Komponente

ErrorMessage

Die *ErrorMessage*-Klasse (Anhang 6) realisiert für jede anzuzeigende Meldung ein neues Fenster, in dem die entsprechenden Hinweise angezeigt werden.

```
public void errorProjectEmpty() {
        Shell shell = new Shell();
        MessageDialog.openInformation(shell, translationService.translate("%dashboard.errorlabel",
                CONTRIBUTOR_URI),translationService.translate("%dashboard.errorProjectEmpty",
                                                                CONTRIBUTOR_URI));
}
```

Die Fehlermeldungen werden an den entsprechenden Stellen in die zugehörigen Klassen injiziert.

```
ErrorMessage error = ContextInjectionFactory.make(ErrorMessage.class, context);
error.errorProjectEmpty();
```

TranslationService

Der *TranslationService* wird eingesetzt, da der Test-Editor eine mehrsprachige Benutzeroberfläche bieten soll. So sind auch alle Texte der Dashboard-Elemente zunächst in Deutsch und in Englisch vorhanden. Die Übersetzungen sind in den Dateien *bundle_de.properties* und *bundle.properties* (Anhang 3 und 4) gespeichert. So könnte man jede weitere Sprache einbinden, indem man die deutsche Übersetzung in die jeweilige Sprache übersetzen kann. Da in der Datei ein Mapping zwischen einer eindeutigen ID und dem übersetzten Term steht. Dadurch wird die ID im Code mit der zugehörigen Übersetzung während des Ablaufs ersetzt und so die Mehrsprachigkeit realisiert. Dieser Service wird in die entsprechenden Klassen injiziert und abgerufen.

```
@Inject
private static TranslationService translationService;
// path to translation file.
public static final String CONTRIBUTOR_URI = "platform:/plugin/org.testeditor.dashboard";
…
mPart.setLabel(translationService.translate("%dashboard.table.label.lastrun", CONTRIBUTOR_URI) +
                                                                                " " + projectname);
```

5. Schlussbetrachtung

Im Rahmen dieser Arbeit wurde das Dashboard für die Visualisierung der Testergebnisse im Test-Editor als Java-Plug-In entwickelt. Solche agilen Methoden wie Continuous Integration und Continuous Delivery setzen ein schnelles Feedback in allen Softwareentwicklungs- und Softwarelieferphasen voraus. Nach einer Einführung in die Konzepte des CI und des CD ist es klar geworden, dass diese die Qualitätssicherung als unverzichtbaren Bestandteil haben. Der Test-Editor, ein Werkzeug für automatisierte Tests, übernimmt eine der Qualitätssicherung-Phasen und zwar die Akzeptanztests.

Nach der Beschreibung des Test-Editors und Konzipierung des Dashboards wurden die relevanten Anforderungen analysiert. Der Kontakt zu den zukünftigen Benutzern des Dashboards hat sich als sehr effizient und erfolgsfördernd gezeigt. Mit Hilfe der auf die Zielgruppe orientierten Umfrage wurden die Anforderungen an das Dashboard spezifiziert. Durch die enge Zusammenarbeit der Kunden und den Entwicklern war es möglich, die den Erwartungen entsprechende Software zu konzipieren und zu realisieren.

Nach der Analyse der Umgebung und Lösungsalternativen für den Test-Editor wurde der erste Designentwurf und Entwicklungskonzept, die in der Implementierung umgesetzt wurden, erstellt.

Das Dashboard-Design wurde während des Implementierungsprozesses in mehreren Iterationen überarbeitet, um die Anforderungen optimal umzusetzen und eventuell zu erweitern. Bei Darstellungsproblemen, wie zum Beispiel der farblichen Gestaltung oder anzuzeigenden Daten, gab es genug Spielraum zur Improvisation. Eine passende Lösung im Einklang mit den Anforderungen konnte dadurch zielorientiert und für alle Beteiligten zufriedenstellend gefunden werden.

Die wichtigsten Prinzipien der agilen Softwareentwicklung, wie Teamarbeit und kontinuierliches Präsentieren der Zwischenstände, haben das Treffen der richtigen Entscheidungen durch Feedbacks, Hinweise und Lösungsvorschläge stark gefördert. Durch die schrittweise Implementierung der Dashboard-Oberfläche gewann der Entwicklungsprozess an Effizienz und Struktur. Wie für die agile Vorgehensweise

typisch, wurden auch die vorläufigen Lösungsansätze überdacht, was zum nachträglichen Überarbeiten des Quell-Codes führte.

Während der Entwicklung wurde das Dashboard manuell auf die Akzeptanzkriterien getestet. Sogenannte Unit-Tests (Modultests) wurden dabei aber nicht erstellt, sondern erst durch das Entwicklerteam während der Qualitätssicherungsphase des Dashboards durchgeführt.

Im Laufe des Entwicklungsprozesses kamen einige verbesserungswürdige Punkte zum Vorschein. Zum einen wurden die Anzeigen der Tabelle „Alle Testläufe" und des Diagramms „Testdauertrend" abweichend vom Anfangskonzept und im Widerspruch zur Namensgebung auf 30 Testläufe begrenzt. Dies war wegen der Übersichtlichkeit notwendig, da bei der großen Anzahl der Testläufe, die im Diagramm „Testdauertrend" angezeigten Elemente überlappten. Die Möglichkeit, den Wertebereich über die Zeit in der Tabelle und im Diagramm über ein Menü zu definieren, oder die Skalierung des Diagramms anzupassen, bleibt der Zukunft vorbehalten. Die Begrenzung in der Tabelle „Alle Testläufe" wurde für die Übereinstimmung mit dem Diagramm „Testdauertrend" entsprechend begrenzt.

Zum anderen ist es nach näherer Betrachtung der Anforderung für den Testdauertrend aufgefallen, dass das Testdauertrend-Diagramm falsche Aussagen über die Testläufe provozieren kann und kritisch betrachtet werden muss. Die Zunahme der Testdauer deutet nicht automatisch auf fehlerhafte Testläufe hin. Die Verlangsamung kann beispielweise durch zusätzlich eingefügte Testschritte, Testfälle oder Testsuiten, die jeweils erfolgreich ausgeführt wurden, verursacht werden. Auch die unterschiedlichen Zustände in den Systemumgebungen zur Testlaufzeit können die Dauer der Testläufe beeinflussen. Somit kann das Dauertrend-Diagramm nur als Hinweis auf mögliche Probleme verwendet werden. Um die Analyse zu verbessern, sollte der Testdauertrend gemeinsam mit der Tabelle „Alle Testläufe", die die Zusammensetzungen und fehlerhaften Testschritte bzw. Testfälle der Testläufe zeigt, betrachtet werden.

Nach der Implementierung wurde das Dashboard einigen Befragten vorgeführt und wurde positiv angenommen. Zusätzlich wurden die Vorschläge für mögliche Erweiterungen geäußert, wie zum Beispiel:

- das Neustarten eines Testlaufs oder einer Gruppe der Testläufe direkt aus der Dashboard-Ansicht;
- das Wechseln in die Dashboard-Ansicht über den Rechten-Maus-Klick auf ein Projekt in der Test-Editor-Ansicht;
- gleichzeitiges Anzeigen mehrerer unterschiedlicher Projekten, Testläufen, Testfällen usw. in den zugehörigen Registerblättern in den entsprechenden Bereichen des Dashboards;

Diese Wünsche sprechen dafür, dass das Arbeitsergebnis den aktuellen Bedürfnissen der Anwender entspricht, ein Interesse am weitern Ausblick erweckt und künftige Erweiterung aus Anwendersicht verspricht.

Durch das Dashboard wurde ein schneller Überblick über die ausgeführten Testläufe und folglich schnelles Feedback über den Verlauf der Akzeptanztests ermöglicht. Es wurde eine Basis für den weiteren Ausbau des Dashboards gelegt.

Anhangsverzeichnis

Der Anhang der Arbeit befindet sich auf der Compact Disc und ist bei der Prüferin Prof. Dr. –Ing. Karin Landenfeld oder bei dem Prüfer Dipl.-Inf.(FH) Olaf Lange anzusehen.

Anhang 1: AllRunsResult.pdf

Anhang 2: Bachelorthesis_Alexander_Lebedev.pdf

Anhang 3: bundle.pdf

Anhang 4: bundle_de.pdf

Anhang 5: DirectMenuItem.pdf

Anhang 6: ErrorMessage.pdf

Anhang 7: GetDataTableLastRuns.pdf

Anhang 8: MANIFEST.pdf

Anhang 9: MenuToolbarItemHandler.pdf

Anhang 10: MyContentProvider.pdf

Anhang 11: MyLabelProvider.pdf

Anhang 12: MyViewerComparator.pdf

Anhang 13: ProjectComboBox.pdf

Anhang 14: SwitchPerspectiveDashboardHandler.pdf

Anhang 15: SwitchPerspectiveTestHandler.pdf

Anhang 16: TableAllRuns.pdf

Anhang 17: TableDurationTrend.pdf

Anhang 18: TableErrorIdent.pdf

Anhang 19: TableLastRuns.pdf

Anhang 20: TestResult.pdf

Anhang 21: Umfrageergebnisse.pdf

Literaturverzeichnis

(AKQUINET 2015) Akquinet AG (2014). In: http://www.akquinet.de/ (Zugriff: 20.01.2015)

(BALZERT 1998) Balzert, H. (1998): *Lehrbuch der Software-Technik*. Band II, 1. Auflage. Heidelberg: Spektrum Akademischer Verlag.

(BERNER 2008) Berner & Mattner Systemtechnik GmbH (2008): *Testbarkeit und Testautomatisierung. Solide Basis für die Qualitätssicherung von Softwaresystemen*. In: http://www.berner-mattner.com/cms/upload/1_PDF/_NEWSLETTER/BernerMattner_Newsletter_Industry_8_DE.pdf (Zugriff: 15.01.2015)

(BIRK u. LUKAS 2014a) Birk, A., Lukas, C. (2014): *Eine Einführung in Continuous Delivery. Teil 1: Grundlagen*. In: http://www.heise.de/developer/artikel/Eine-Einfuehrung-in-Continuous-Delivery-Teil-1-Grundlagen-2176380.html (Zugriff: 15.01.2015)

(BIRK u. LUKAS 2014b) Birk, A., Lukas, C. (2014): *Eine Einführung in Continuous Delivery. Teil 3: Acceptance Test Stage*. In: www.heise.de/developer/artikel/Eine-Einfuehrung-in-Continuous-Delivery-Teil-3-Acceptance-Test-Stage-2457023.html (Zugriff: 15.01.2015)

(BIRT 2015a) BIRT *About* (2014). In: http://www.eclipse.org/birt/about/ (Zugriff: 15.01.2015)

(BIRT 2015b) BIRT *Architecture* (2014). In: http://www.eclipse.org/birt/about/architecture.php (Zugriff: 15.01.2015)

(BIRT 2015c) BIRT *Design* (2014). In: http://www.eclipse.org/birt/about/design.php (Zugriff: 15.01.2015)

(BIRTWORLD 2015) Abblidung: *BIRTs verschiedene Dashboards Ansichten*. (2010) In: http://birtworld.blogspot.de/2010/06/birt-excel-output.html (Zugriff: 01.03.2015)

(FEATHERS 2011) Feathers, M. C. (2011): *Effektives Arbeiten mit Legacy Code. Refactoring und Testen bestehender Software*. 1. Auflage. mitp, eine Marke der Verlagsgruppe Hüthig Jehle Rehm GmbH

(FOWLER 2013) Fowler, M. (2013): *Continuous Integration*. In: http://martinfowler.com/articles/continuousIntegration.html

(Zugriff: 15.01.2015)

(FRANZ 2007) Franz, K. (2007): *Handbuch zum Testen von Web- und Mobile-Apps: Testverfahren, Werkzeuge, Praxistipps*. Springer Verlag.

(HUMBLE 2010) Humble, J. (2010): *Continuous Delivery*. In: http://continuousdelivery.com/2010/02/continuous-delivery/ (Zugriff: 15.01.2015)

(it-agile 2015a) *Was ist agile Softwareentwicklung?* (2015) In: http://www.it-agile.de/wissen/methoden/agilitaet/ (Zugriff: 11.02.2015)

(it-agile 2015b) *Vorteile agiler Methoden.* (2015) In: http://www.it-agile.de/wissen/methoden/vorteile-agiler-methoden/ (Zugriff: 15.01.2015)

(KAMANN u. WENDT 2012) Kamann, T., Wendt, H. (2012): *On the Road to Continuous Delivery.* In: http://jaxenter.de/artikel/On-the-Road-to-Continuous-Delivery-0 (Zugriff: 15.01.2015)

(KLEUKER 2013) Kleuker, S. (2013): *Qualitätssicherung durch Softwaretests. Vorgehensweisen und Werkzeuge zum Test von Java-Programmen*. Springer Fachmedien Wiesbaden

(MARTIN 2014) Martin,R. C. (2014): *Clean Coder. Verhaltensregeln für professionelle Programmierer*. 1. Auflage. mitp, eine Marke der Verlagsgruppe Hüthig Jehle Rehm GmbH

(OSHEROVE 2010) Osherove, R. (2010): *The Art of Unit Testing. Deutsche Ausgabe*. 1. Auflage. mitp, eine Marke der Verlagsgruppe Hüthig Jehle Rehm GmbH

(RÜDIGER 2013) Rüdiger, J. et al. (2013): *Umfrage – Einführung in die Methoden der Umfrageforschung*. 3. Auflage, Oldenbourg Verlag

(SCHLUFF u. FEUSTEL 2012) Schluff S., Feustel B. (2012): *Continuous Integration in Zeiten agiler Programmierung.* In: http://heise.de/-1427092 (Zugriff: 15.01.2015)

(SWANSON 2003) Swanson, M. et al. (2003): *Security Metrics Guide for Information Technology Systems*. NIST Special Publication 800-55, July 2003. In: http://cid-7086a6423672c497.skydrive.live.com/self.aspx/.Public/NIST%20SP%20800-55.pdf (Zugriff: 15.01.2015)

(Test-Editor 2014a)	Akquinet AG (2014): *Test-Editor. Benutzer-Handbuch*. Release 1.6.1. In: http://testeditor.org/wp-content/uploads/2014/04/Test EditorUserManualDe.pdf (Stand 04/2014; Zugriff: 15.01.2015)
(Test-Editor 2014b)	Akquinet AG (2014): *Test-Editor. Administrator-Handbuch. Release 1.6.1.* In: http://testeditor.org/wp-content/uploads/2014/04 /TestEditorAdminManualDe.pdf (Stand 04/2014; Zugriff: 15.01.2015)
(Test-Editor 2014c)	Akquinet AG (2014): *Test-Editor. Features.* In: http://testeditor.org/de/features/ (Zugriff: 15.01.2015)
(TEUFEL u. HEIMING 2012)	Teufel, M., Heiming Dr. J. (2012): *Eclipse 4. Rich Clients mit dem Eclipse SDK 4.2.* [E-Book]. entwickler.press. DOI: https://entwickler.de/press/buecher/eclipse-4-120791.html

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Dieses Blatt, mit der folgenden Erklärung, ist nach Fertigstellung der Abschlussarbeit durch den Studierenden auszufüllen und jeweils mit Originalunterschrift als <u>letztes Blatt</u> in das Prüfungsexemplar der Abschlussarbeit einzubinden.
Eine unrichtig abgegebene Erklärung kann -auch nachträglich- zur Ungültigkeit des Studienabschlusses führen.

<u>Erklärung zur selbstständigen Bearbeitung der Arbeit</u>

Hiermit versichere ich,

Name: Lebedev

Vorname: Alexander Borisovič

dass ich die vorliegende Bachelorarbeit bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Entwurf und Implementierung eines Dashboards in Java zur Visualisierung von Testergebnissen

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

- die folgende Aussage ist bei Gruppenarbeiten auszufüllen und entfällt bei Einzelarbeiten -

Die Kennzeichnung der von mir erstellten und verantworteten Teile der Bachelorarbeit ist erfolgt durch:

Hamburg	12.03.2015	
Ort	Datum	Unterschrift im Original