Auf dem Weg zur automatisiert verifizierbaren Spezifikation – Software-Testautomatisierung in der Medizintechnik

Stefan Trapp

Palmerstraße 31
20535 Hamburg

stefan.trapp@stefan-trapp-consulting.de
www.stefan-trapp-consulting.de

1 Einleitung

Testautomatisierung und testgetriebene Entwicklung (*Test-Driven Development*, TDD) sind seit Jahren „Megatrends“ der Softwareentwicklung. Da gerade die Softwareentwicklung für die Medizintechnik – wie später noch erläutert wird – durch hohe Testaufwände gekennzeichnet ist, hat dieses Thema in dieser Branche hohe Relevanz.

Dieses Arbeitspapier beschreibt einen Ansatz zur **Automatisierung des Softwaretests**, der für einen **großen Medizinprodukte-Hersteller** implementiert wurde. Die Vorgehensweise umfasst **mehrere Stufen des automatisierten Testens**. Ein zentraler Aspekt der Strategie sind **Akzeptanztests mit dem Tool „FitNesse“**. Aus den Ergebnissen dieser kontinuierlich ausgeführten Tests (*Continuous Integration*) können bei Bedarf (z. B. bei einem Software-Release) **automatisch Dokumente zur Erfüllung regulatorischer Auflagen generiert werden**. Auf diese Weise werden – gegenüber den bislang manuell durchzuführenden Tests – erhebliche Aufwände eingespart, somit Kosten reduziert, die Durchlaufzeit verringert sowie die Zuverlässigkeit von Software und Test verbessert.

2 Spezielle Anforderungen an medizinische Software

Die Medizintechnik-Branche ist – außer durch komplexe Produkte – durch hohe juristische und regulatorische Anforderungen gekennzeichnet, da Medizinprodukte unmittelbar auf den menschlichen Organismus Einfluss nehmen. Beispiele zu erfüllender Ansprüche sind:

- Einhaltung der Anforderungen des deutschen Medizinproduktegesetzes (MPG)
- CE-Zertifizierung
- Notwendigkeit klinischer Tests vor Markteinführung
- Einhaltung allgemeiner und medizintechnikspezifischer internationaler Normen

Die Erfüllung all dieser Ansprüche bedingt einen hohen formalen Aufwand (z. B. Dokumentations- und Testaufwand) und dadurch tendenziell langsame Releasezyklen.

Folgende Normen sind für Medizinprodukte von besonderem Interesse:

- DIN EN ISO 14971:2009-10, „Medizinprodukte – Anwendung des Risikomanagements auf Medizinprodukte“ (ISO 14971:2007, korrigierte Fassung 2007-10-01); Deutsche Fassung EN ISO 14971:2009
- DIN EN 60601-1:2006 (3. Edition) „Medizinische elektrische Geräte – Teil 1: Allgemeine Festlegungen für die Sicherheit einschließlich der wesentlichen Leistungsmerkmale“ (IEC 60601-1:2005); Deutsche Fassung EN 60601-1:2006[1]
- DIN EN 62304:2006 „ Medizingeräte-Software – Software-Lebenszyklus-Prozesse“ (IEC 62304:2006); Deutsche Fassung EN 62304:2006

[1] Zur Familie der DIN EN 60601-1 gehören noch rund 10 Kollateralstandards (Ergänzungsnormen, EN 60601-1-x) und rund 60 Partikulärstandards (besondere Festlegungen, EN 60601-2-y). Ein spezifisches Medizinprodukt hat die Anforderungen der EN 60601-1 sowie die der anwendbaren Kollateral- und Partikulärstandards einzuhalten. Letztere können einzelne Anforderungen der Kollateralstandards und des Basisstandards aufheben, verändern oder ergänzen. Der Inhalt der Partikulärstandards bestimmt somit die Anforderungen für ein Medizinprodukt und ihre Verfügbarkeit, ob für ein Medizinprodukt schon die 3. Edition (also EN 60601-1:2006) angewendet werden muss/kann, oder ob noch die 2. Edition verwendet werden muss/darf. In der EU dürfen seit dem 1. Juni 2012 Medizinprodukte im Grundsatz nur noch in Verkehr gebracht werden, wenn sie der 3. Edition entsprechen. Per EU-Amtsblatt ist geregelt, dass die 2. Edition (EN 60601-1:1990) ab diesem Datum nicht mehr zur Erfüllung der sog. grundlegenden Anforderung der Medizinprodukte-Richtlinie und damit zur In-Verkehr-Bringung herangezogen werden kann. Für Produkte mit Partikulärstandards legt das EU-Amtsblatt Übergangstermine fest, wie lange noch auf der 2. Edition basierende Produkte in Verkehr gebracht werden dürfen. [6], [7]

Mit der internationalen Norm DIN EN 60601-1:2006 (3. Edition) wurde die Bedeutung des Risikomanagements erheblich gestärkt. Risikomanagement gemäß DIN EN ISO 14971 ist damit ein wesentlicher Bestandteil des Sicherheitskonzeptes von Medizingeräten geworden. [7]

DIN EN 62304:2006 definiert den Software-Lebenszyklus, nach dem medizinische Software und Software in medizinischen Geräten zu entwickeln sind. Abbildung 1 zeigt diesen Zyklus als Teil der Entwicklung eines Gesamtsystems.

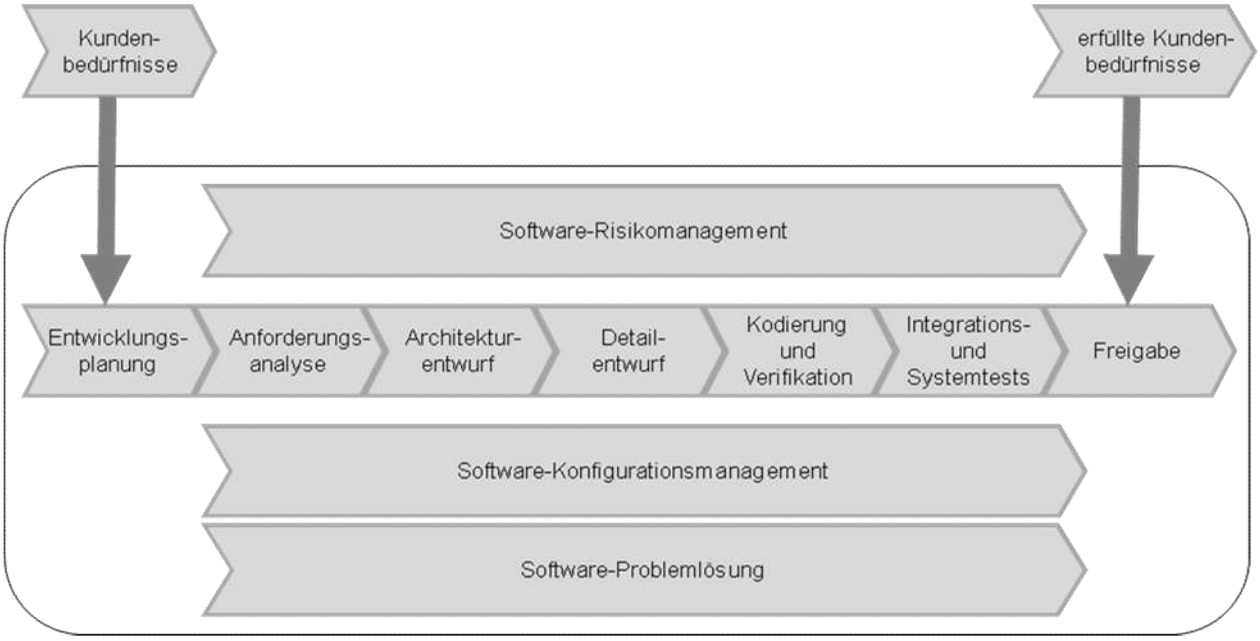

Abbildung 1: Software-Lebenszyklus gemäß DIN EN 62304:2006 [4]

Die Norm ist zwischen der Europäischen Union (EU) und den USA harmonisiert und somit für beide Märkte gleichermaßen relevant. Neben dem Prozess der Softwareentwicklung werden die entwicklungsbegleitenden Prozesse Software-Risikomanagement, Software-Konfigurationsmanagement und Software-Problemlösung definiert. [18]

Eine zentrale Anforderung der DIN EN 62304 ist die Dokumentation und **Nachverfolgbarkeit** (*Traceability*) der Erfüllung **von System-/Software-Anforderungen sowie der** in der Software **implementierten risikoreduzierenden Maßnahmen** (*Risk Reduction Measures*) durch den System-/Softwaretest. Die risikoreduzierenden Maßnahmen sind ein Ergebnis des in Abbildung 1 dargestellten Software-Risikomanagements. Aus den Erfordernissen der DIN EN 62304 ergibt sich erstens, dass der Entwicklungsprozess für medizinische Software zwingend den Risikomanagementprozess umfassen muss. Zweitens **muss jede risikoreduzierende Maßnahme in der Software mit mindestens einem Testfall verknüpft sein**, damit sichergestellt werden kann, dass die Maßnahme korrekt implementiert wurde. Dies gilt für alle Software-Medizinprodukte unabhängig von ihrer Klassifizierung. [18]

3 Softwaretests und testgetriebene Entwicklung

3.1 Softwaretests

Tests sind ein wichtiges Element der Qualitätssicherung von Software. Sie erlauben es, das Testobjekt hinsichtlich der Erfüllung der definierten Anforderungen zu prüfen und zu bewerten. Während der Entwicklung dienen Tests der späteren möglichst fehlerfreien Inbetriebnahme der Software. Fehlerfreiheit ist insbesondere dann eine wichtige Qualitätsanforderung, wenn aus Fehlern eines Software-Systems eine Gefährdung menschlichen Lebens resultieren

kann, wie es z. B. in der Luftfahrt oder der Medizintechnik der Fall ist. Allerdings kann durch Tests nicht die Fehlerfreiheit einer Software nachgewiesen werden, sondern es ist nur eine Aussage über die ausgeführten Testfälle möglich. Erkannte Softwarefehler können im Rahmen der Software-Problemlösung (vgl. Abbildung 1) bearbeitet und behoben werden. [1]

3.2 Dynamische Softwaretests und Testautomatisierung

Ihrer großen Bedeutung im Entwicklungsprozess entsprechend, sind Tests oft mit erheblichem Zeit- und Ressourcenaufwand bei der Entwicklung und Wartung von Software verknüpft. Der Testaufwand hängt ganz wesentlich von den unter dem Begriff der **Testbarkeit** zusammengefassten Eigenschaften des zu testenden Produkts ab. Neben anderen qualitätsverbessernden Maßnahmen sind daher die verbesserte Testbarkeit von Software und in der Folge die Möglichkeit zur **Automatisierung von Tests** wichtige Elemente für eine höhere Effektivität und Effizienz der Testaktivitäten. [11]

Softwaretests können in dynamische und statische[1] Verfahren eingeteilt werden. Im Fokus der **Testautomatisierung** stehen die **dynamischen Tests**, die auf der Ausführung des zu testenden Programmteils beruhen. Für einen solchen Test(-fall) werden die Eingaben und die erwarteten Ausgaben spezifiziert. Letztere werden nach Testausführung mit dem tatsächlichen Resultat verglichen. Außerdem wird festgelegt, wie das System initialisiert werden muss, um den Test durchführen zu können. Dynamische Tests können in verschiedene **Teststufen** eingeteilt werden, die sich an den Stufen des V-Modells der Softwareentwicklung orientieren, nämlich

- Unit-Tests (auch Komponenten- oder Modultests),
- Integrationstests,
- Systemtest und
- Akzeptanztests.

Testgegenstand von **Unit-Tests**, die meistens durch die Softwareentwickler selbst geschrieben und mit Hilfe spezieller Entwicklungstools (***xUnit***, ***xMocks*** und **Tools zur kontinuierlichen Integration**, *Continuous Integration*[2]) automatisiert ausgeführt werden, ist die Funktionalität einzelner, abgrenzbarer Teile der Software (z. B. Klassen, Units oder Module). Ziel der Unit-Tests ist der Nachweis der Lauffähigkeit und korrekter fachlicher (Teil-)Ergebnisse. [9], [12], [15], [17]

Integrationstests prüfen die Zusammenarbeit voneinander abhängiger Komponenten. Sie können mit den gleichen Tools wie Unit-Tests (s. o.) implementiert werden, sollen aber die Korrektheit der beteiligten Komponenten-Schnittstellen und von komponentenübergreifenden Abläufen nachweisen.

Im **Systemtest** wird – i. d. R. durch die realisierende Organisation und in einer Testumgebung – geprüft, ob das gesamte System seine Spezifikation (funktionale und nicht-funktionale Anforderungen) erfüllt. Der Systemtest kann beispielsweise mit Testtools wie QF-Test (http://www.qfs.de/de/index.html) für die grafische Benutzerschnittstelle eines Softwaresystems (GUI-Testautomatisierung, „*Record-and-Play Tools*“) automatisiert werden.

---

[1] Statische Tests erfolgen ohne Programmausführung, z. B. durch Reviews, statische Code-Analyse oder formale Verifikation. [2]

[2] Das „*x*“ steht im Java-Umfeld beispielsweise für JUnit (http://www.junit.org/) und JMock (http://jmock.org/), im .NET-Umfeld für Rhino Mocks (http://hibernatingrhinos.com/oss/rhino-mocks) oder NMock (http://nmock3.codeplex.com/). Beispiele für Tools zur kontinuierlichen Integration sind CruiseControl (http://cruisecontrol.sourceforge.net/) oder Jenkins (http://jenkins-ci.org/).

Akzeptanztests (auch Abnahmetests oder *User Acceptance Tests*) sind Tests einer gelieferten Software durch den Auftraggeber bzw. Kunden.[1] Neben kompletten Systemen können Akzeptanztests auch für **andere** (kleinere) **Software-Artefakte** (z. B. **Software-Komponenten** oder **–Subsysteme**) **mit eigenem Software-Lebenszyklus** (Releasezyklus) **und eigener Anforderungsdefinition** realisiert werden. Ein verbreitetes Open Source Tool für automatisierte Akzeptanztests ist **FitNesse** (http://fitnesse.org/), basierend auf dem „*Framework for Integrated Test*“ (Fit, http://fit.c2.com/). FitNesse gestattet die browserbasierte Definition von Testfällen in Form von Entscheidungstabellen in einem Wiki. Da hierfür keine Programmierkenntnisse erforderlich sind, kann das Testfall-Wiki **durch** das **Anforderungsmanagement oder** die **Domänen-Experten** (nicht die Softwareentwicklung) erstellt werden. Es stellt im Idealfall die vollständige **ausführbare Spezifikation eines Software-Artefakts** dar. Nach Ausführung der Tests wird das Testergebnis im Wiki – ähnlich wie bei den xUnit-Tools – durch „Ampelfarben“ sichtbar gemacht. [1]

3.3 Testgetriebene Entwicklung

Bei testgetriebener Entwicklung (*Test-Driven Development*, TDD) wird eine Software so entwickelt, dass die automatisierten Testfälle **vor** der Implementierung der zugehörigen Funktionalität (d. h. vor Implementierung des zugehörigen Produkt-Codes, der die per Testfall definierten Anforderungen erfüllen soll) geschrieben werden. Da die benötigte Funktionalität bereits durch die ausführbaren Testfälle spezifiziert ist, kann ein gesondertes Dokument zur Anforderungsdefinition im Prinzip entfallen. TDD gibt den Entwicklern und Kunden früh Feedback über Probleme bei der Implementierung und der Ausführung der Tests und zielt so auf eine umfassende Testbarkeit schon in den frühen Phasen des Entstehungsprozesses eines Softwaresystems. [3], [10]

Testgetriebene Entwicklung treibt die Automatisierung des Softwaretests „auf die Spitze“, erfordert aber zur ihrer Realisierbarkeit einige Voraussetzungen/Praktiken:

Kontinuierliche Integration
Code-Änderungen werden frühzeitig (normalerweise mindestens täglich) in der Versionsverwaltung hinzugefügt („eingecheckt“). Anschließend wird das Gesamtsystem automatisch durch ein Tool zur *Continuous Integration* neu gebaut und getestet, um mögliche Probleme infolge der Änderung sofort erkennbar zu machen. [5]

Einhaltung der Prinzipien des objektorientierten Designs
Die Prinzipien des objektorientierten Designs, z. B. Single Responsibility Prinzip, Open-Closed Prinzip, Liskovsches Substitutionsprinzip, Interface Segregation Prinzip oder Dependency Inversion Prinzip, sollen zu einer verbesserten Wartbarkeit sowie Testbarkeit und damit erhöhten Qualität und Lebensdauer von Software führen. Sie zielen hierzu auf die **Verringerung von Abhängigkeiten** (d. h. auf eine **geringere Koppelung**[2] oder, positiv ausgedrückt, auf eine **verbesserte Isolierbarkeit** der Systemelemente) und der **Komplexität** von Komponenten, Klassen oder anderen Software-Artefakten. [11], [13], [14], [16]

Refactoring
Refactoring bezeichnet die manuelle oder automatisierte Veränderung der Struktur eines Softwaresystems, ohne die Funktionalität oder die Schnittstellen nach außen zu verändern. Ziel ist es, die Lesbarkeit, Wartbarkeit und Erweiterbarkeit der Software zu verbessern. Automatisierte Tests, die eine notwendige Voraussetzung für zuverlässiges Refactoring sind, dienen hierbei als Regressionstests: Bei jedem Refactoring der Software soll durch die Ge-

[1] Handelt es sich um ein komplettes Softwaresystem, werden diese Tests i. d. R. in der Produktivumgebung und mit Kopien von „Echtdaten“ durchgeführt.
[2] Zur Messung der Isolierbarkeit bzw. Koppelung der Teile eines Systems existieren diverse Koppelungs-/Abhängigkeitsmetriken. [11]

samtheit der Testfälle sichergestellt werden, dass trotz der durchgeführten Änderungen die implementierte Funktionalität intakt bleibt.[1] [8], [15]

4 Implementierte Teststrategie bei einem Medizingeräte-Hersteller

In diesem Abschnitt wird die bei einem großen Medizingeräte-Hersteller implementierte Teststrategie für Systeme, die aus Hard- und Software bestehen (z. B. diagnostische Röntgensysteme), vorgestellt. Grundsätzlich werden dabei Verifikations- und Validationstests unterschieden. Während durch die Verifikation festgestellt wird, ob eine Software (bzw. ein Produkt/System) ihrer (seiner) Spezifikation entspricht (funktionales Testen), wird mittels Validation geprüft, ob eine Software (bzw. ein Produkt/System) den (klinischen) Benutzeranforderungen entspricht. Der Fokus liegt im Folgenden auf der Verifikation, nicht der Validation.

Die Systemverifikation erfolgt gemäß der Darstellung in Abbildung 2. Input für alle (Verifikations-)Tests (linke „Spalte“ in Abbildung 2) sind erstens die Systemanforderungen und die aus dem Risiko-Management resultierenden risikoreduzierenden Maßnahmen (RRM). Diese Anforderungen umfassen als Teilmenge auch die vom Software-Subsystem zu erfüllenden Anforderungen. Da das Software-Subsystem aus mehreren Software-Komponenten besteht, werden zweitens die System-Anforderungen und die RRM den Softwarekomponenten zugeordnet und auf Komponentenebene weiter verfeinert.

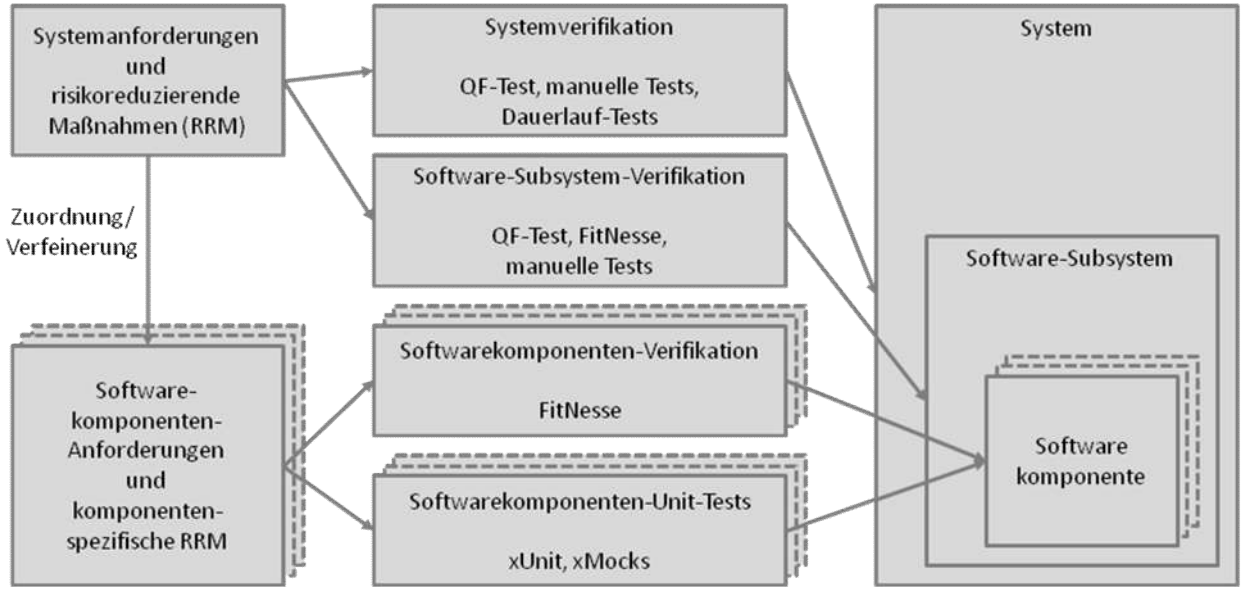

Abbildung 2: Teststufen und zugehörige Testmethoden bzw. -werkzeuge

Die Verifikation findet auf Basis dieser Anforderungen auf den Ebenen der **Softwarekomponente**, des **Software-Subsystems** und des **Gesamtsystems**, bestehend aus Hard- und Software, statt (mittlere und rechte „Spalte“ in Abbildung 2). Bei den Komponenten-Tests kann weiter in Softwarekomponenten-Unit-Tests (mit xUnit und xMocks) und die Softwarekomponenten-Verifikation (mit FitNesse, vgl. hierzu Abschnitt 5) unterschieden werden. Auf den Ebenen „Software-Subsystem“ und „System“ werden – neben automatisierten Tests mit FitNesse bzw. QF-Test – weitere Testmethoden, z. B. manuelle Tests und Dauerlauf-Tests, angewendet.

[1] Regressionstests sind für *Bugfixes* oder die Implementierung zusätzlicher Funktionalität ebenfalls von Bedeutung.

5 Automatisierte Software-Verifikation und Dokumentation der Verifikation mit FitNesse

Dieser Abschnitt beschäftigt sich eingehender mit der **Softwarekomponenten-Verifikation** und der **Software-Subsystem-Verifikation** unter Einsatz des Tools „FitNesse“ [1]. Ziel der beschriebenen Vorgehensweise ist nicht nur die **automatisierte Ausführung der Verifikation**, sondern **zusätzlich die automatische Erzeugung der erforderlichen Dokumentation** in Form eines Verifikations-Ergebnisberichts, z. B. zur Erfüllung der Nachweispflicht gemäß DIN EN 62304. Mit anderen Worten ist es die langfristige Intention, zu einer **„automatisiert ausführ- bzw. verifizierbaren Spezifikation“** zu gelangen, die die wiederholte, aufwändige, manuelle Durchführung und Dokumentation der Verifikationstests weitgehend überflüssig macht. Dies ist umso interessanter, als im vorliegenden Fall Software-Releases häufig, nämlich

- für eine komplette Produktfamilie und
- auf mehreren Software-Branches parallel (Neuprodukt-Entwicklung und Produkt-Lebenszyklus-Management, *Life-Cycle Management*, LCM),

erfolgen.

Abbildung 3 (Seite 7) verdeutlicht die Vorgehensweise. FitNesse besteht aus einem Webserver (der Block „FitNesse“ in Abbildung 3), einem Wiki und einer *Test Engine* („SLIM“ in Abbildung 3). Tests werden, wie schon in Abschnitt 3.2 erwähnt, durch Domänen-Experten (*Domain Experts*) oder Requirements-Ingenieure im FitNesse Testfall-Wiki (browserbasiert) erzeugt, nicht durch Software-Entwickler. Diese Testfälle beschreiben in Tabellenform die Geschäftslogik in verständlichem Klartext. FitNesse erreicht so die **Trennung der (ausführbaren) Anforderungen an die Geschäftslogik von der softwaretechnischen Umsetzung**.

Die Anbindung der Testfälle an das zu testende System (*System Under Test*, SUT) wird durch die Entwicklung in Form sogenannter „*Testfixtures*“ (Test-Code) implementiert. Das Wiki gestattet anschließend unter Benutzung von FitNesse und der zugehörigen *Test Engine* SLIM (*Simple List Invocation Method*)[1] browserbasiert die Ausführung der Tests. Die Testergebnisse werden durch „Ampelfarben“ auf den Wiki-Seiten angezeigt. Dieses Testergebnis kann (z. B. in Form eines PDF-Dokuments) manuell (rote Pfeile in Abbildung 3) in die Dokumentvorlage des Verifikations-Ergebnisberichts von Softwarekomponente bzw. Software-Subsystem eingefügt werden und so unmittelbar zur Erfüllung regulatorischer Dokumentations-Anforderungen verwendet werden. Nur dieser geringe manuelle Aufwand ist bei einem Software-Release zu leisten.

Neben dieser „**manuellen Auslösung**“ der automatisierten Verifikation zeigt Abbildung 3 zusätzlich die „**automatisierte Auslösung**“ im Rahmen der **kontinuierlichen Integration**. Die Verifikation mit FitNesse wird in diesem Fall regelmäßig durch das verwendete Werkzeug zur kontinuierlichen Integration gestartet. Das Testergebnis (z. B. in Form einer HTML- oder XML-Datei) kann anschließend – im Fall eines Software-Releases und ebenfalls durch das *Continuous Integration Tool* automatisiert – in die zugehörige Dokumentvorlage (Verifikations-Ergebnisbericht von Softwarekomponente oder Software-Subsystem) eingefügt werden.

[1] SLIM ist neben FIT (*Framework for Integrated Testing*) eine der beiden *Test Engines* des FitNesse Frameworks. In der Regel sollte bevorzugt das neuere SLIM, nicht das ältere FIT, eingesetzt werden.

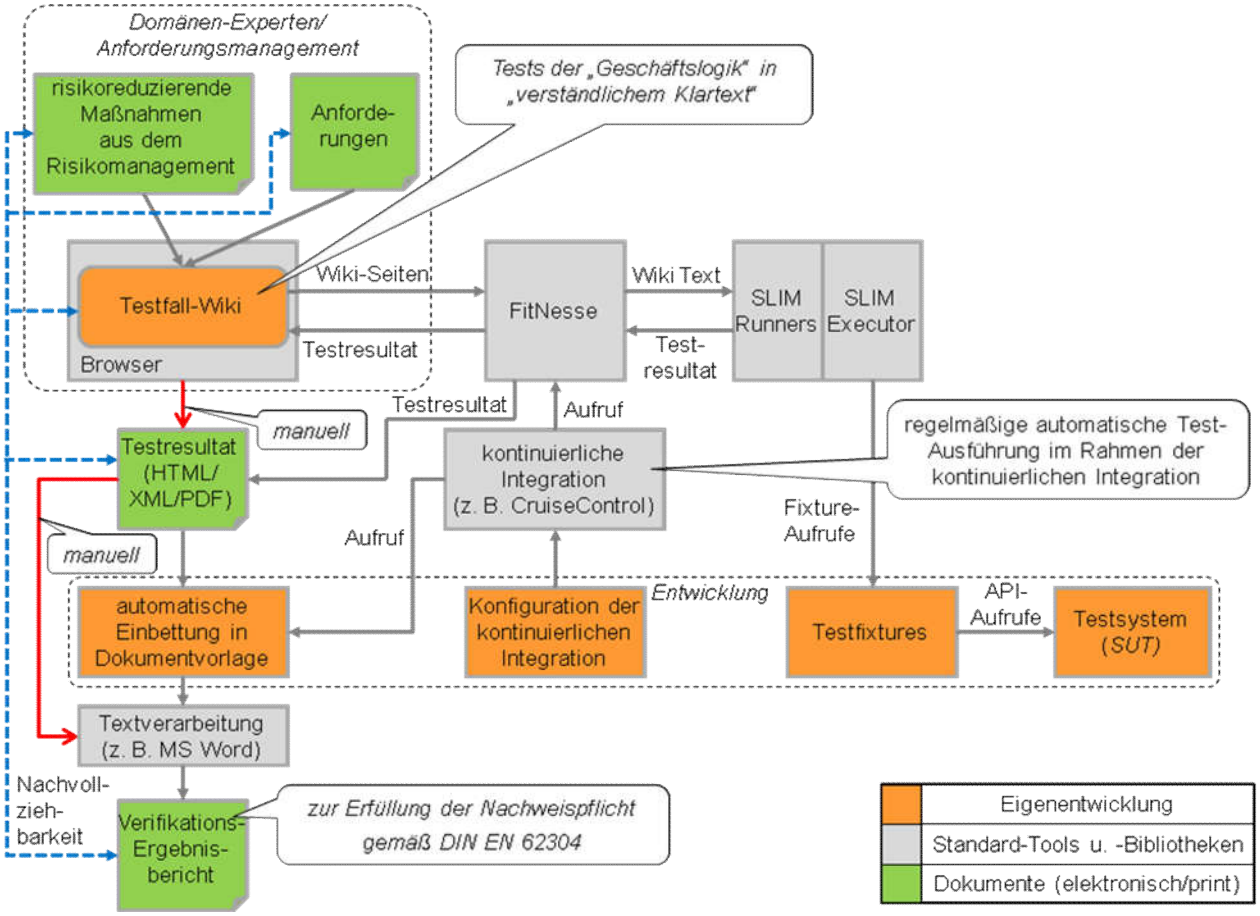

Abbildung 3: Automatisierte Erzeugung von Verifikations-Ergebnisberichten mit Fit-Nesse zur Erfüllung der Nachweispflicht gemäß DIN EN 62304

Abschließend soll an dieser Stelle darauf hingewiesen werden, dass FitNesse ungeachtet seiner Vorzüge auch Schwachpunkte hat:

- Typische Stärken (z. B. keine Lizenzgebühren) und Schwächen (z. B. mangelnde Dokumentation) von Open Source Software
- „Einseitige" Änderungen/Refactorings (z. B. Methodennamen) im Testfall-Wiki oder den *Testfixtures* lassen Tests fehlschlagen, da es keine Verknüpfung zwischen dem Wiki und dem zugehörigen Test-Code, z. B. durch eine integrierte Entwicklungsumgebung (IDE), gibt.
- Keine Autovervollständigung („IntelliSense") bei Bearbeitung der Wiki-Testfälle
- Eingeschränkte Möglichkeiten zur Fehlerfindung/zum *Debugging*

6 Fazit

Der hier vorgestellte Ansatz zur Automatisierung der Softwareverifikation bei einem großen Medizinprodukte-Hersteller hat bereits zu erheblich verringerten Testaufwänden geführt, die Durchlaufzeit verringert sowie die Zuverlässigkeit von Software und Test verbessert. Nachdem bislang nur sämtliche Testfälle für risikoreduzierende Maßnahmen automatisiert wurden, ist zum gegenwärtigen Zeitpunkt zwar ein wichtiges Etappenziel erreicht, auf dem Weg zur Vision der „vollständig automatisiert ausführ- bzw. verifizierbaren Spezifikation" ist aber noch weitere Arbeit zu leisten.

7 Literatur

[1] Adzic, G.: Test Driven .NET Development with FitNesse, 2nd edition, Neuri, London, 2009, im Internet: http://fitnesse.s3.amazonaws.com/tdd_net_with_fitnesse.pdf

[2] Balzert, H.: Lehrbuch der Software-Technik. Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung. Spektrum, Heidelberg, 1998

[3] Beck, K.: Test-Driven development by example, Addison-Wesley, Boston, 2003

[4] DIN EN 62304; VDE 0750-101:2007-03:2007-03 Medizingeräte-Software, Software-Lebenszyklus-Prozesse (IEC 62304:2006); Deutsche Fassung EN 62304:2006, Beuth, Berlin, 2007

[5] Duvall, P. M; Matyas, S., Glover, A.: Continuous Integration: Improving Software Quality and Reducing Risk, Addison-Wesley, Upper Saddle River (NJ), 2010

[6] „EN 60601“, Artikel in der deutschen Wikipedia, im Internet: http://de.wikipedia.org/wiki/EN_60601

[7] „EN 60601-1“, Artikel in der deutschen Wikipedia, im Internet: http://de.wikipedia.org/wiki/EN_60601-1

[8] Fowler, M.: Refactoring: improving the design of existing code, Addison-Wesley, Boston, 2010

[9] Fowler, M.: Mocks Aren’t Stubs, 2007, im Internet: http://martinfowler.com/articles/mocksArentStubs.html

[10] Freeman, S.; Price, N.: Growing Object Oriented Software Guided by Tests, Addison-Wesley, Upper Saddle River (NJ), 2010

[11] Jungmayr, S.: Improving testability of object-oriented systems, Dissertation, FernUniversität Hagen, 2003, im Internet: http://www.dissertation.de/FDP/sj929.pdf

[12] Link, J.: Softwaretests mit JUnit. Techniken der testgetriebenen Entwicklung, 2. Aufl., dpunkt.verlag, Heidelberg, 2005

[13] Martin, R. C.: Agile Software Development. Principles, Patterns, and Practices, Pearson Education, Upper Saddle River (NJ), 2003

[14] Martin, R. C.: Design Principles and Design Patterns, 2000, im Internet: http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

[15] Meszaros, G.: XUnit test patterns: refactoring test code, Addison-Wesley, Upper Saddle River (NJ), 2009

[16] Meyer, B.: Object-oriented software construction, Prentice Hall, Upper Saddle River (NJ), 2009

[17] Osherove, R.: The Art of Unit Testing. With Examples in .NET, Manning, Greenwich (Conneticut), 2009

[18] Stehlik, B.: Software-Entwicklung für medizinische Geräte, Special Medizintechnik & Automation, etz – Elektrotechnik + Automation, 11/2010, VDE Verlag, S. 2-3, im Internet: http://www.etz.de/files/e01161zk_infoteam.pdf