Bachelorarbeit

Fabio von Hertell

Automatisierte Software-Qualitätssicherung am Beispiel eines Frameworks für automatisiertes Testen

Fakultät Technik und Informatik

Faculty of Engineering and Computer Science

Department of Computer Science

Fabio von Hertell

Automatisierte Software-Qualitätssicherung am Beispiel eines Frameworks für automatisiertes Testen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Jörg Raasch
Zweitgutachterin: Prof. Dr. Bettina Buth

Abgegeben am: 23.01.2009

Fabio von Hertell

Thema der Bachelorarbeit

Automatisierte Software-Qualitätssicherung am Beispiel eines Frameworks für automatisiertes Testen

Stichworte

Qualitätssicherung, Qualitätsmanagement, Softwaretest, Testautomatisierung, verteilte Systeme, Windows Communication Foundation

Kurzzusammenfassung

Diese Arbeit befasst sich mit der Qualitätssicherung von Softwareprodukten und geht dabei insbesondere auf die Qualitätsprüfung durch Softwaretests und die Automatisierung derselben ein. Sie bezieht sich hierzu auf die Erstellung eines Frameworks zur Automatisierung von Softwaretests.
Dieses System soll den Softwareentwicklungsprozess dahingehend unterstützen, dass Qualitätsanforderungen, die durch Tests beschrieben sind, vollständig auf eine Software angewendet werden können und die Tätigkeiten, die von Testern oder Entwicklern durchgeführt werden müssen auf ein Minimum reduziert werden. Hierdurch sollen Entwicklungskosten reduziert, Entwickler/Tester entlastet und die Softwarequalität verbessert werden.

Fabio von Hertell

Title of the paper

Automated software quality assurance and the creation of a software testing automation framework

Keywords

Quality assurance, quality management, testing, testing automation, distributed systems, Windows Communication Foundation

Abstract

This work addresses the quality assurance of software products, particularly software testing and automation of testing. For this purpose it refers to the creation of a framework which allows the automation of software testing.
This system is intended to support the software development process in a way that quality requirements, specified by automated tests or by test cases for manual testing, can be applied to a software-system as a whole and the necessary testing activities of testers or developers can be reduced to a minimum. This should reduce development costs, ease the work for developers and testers and improve the software quality.

Inhalt

1 Einführung

Qualitätssicherungsmaßnahmen spielen bei der Entwicklung von Software eine entscheidende Rolle. Schon bei sehr kleinen Softwarelösungen, Programmen mit wenigen Eingabe- und Konfigurationsmöglichkeiten, zeigt sich, dass durch die Kombination der Eingabeparameter eine große Vielfalt an unterschiedlichen Eingabemöglichkeiten besteht. Von diesen sollten möglichst viele (im Idealfall alle) sinnvolle Ergebnisse oder aber aussagefähige Fehlermeldungen erzeugen. Zur Veranschaulichung: Ein einfaches Programm, welches drei Parameter entgegennimmt, die jeweils 10 unterschiedliche Werte annehmen können, kann bereits mit 1000 (10^3) unterschiedlichen Parameterkombinationen aufgerufen werden. Größere Softwareapplikationen mit vielen Eingabemöglichkeiten besitzen eine vielfach höhere Komplexität. Hierdurch wird auch deutlich, dass eine nicht triviale Softwarelösung aufgrund der Komplexität niemals vollständig getestet werden kann. Ziel kann es nur sein, eine möglichst hohe Qualität zu erreichen, indem möglichst viele Funktionen einer Applikation spezifiziert und getestet werden (Testabdeckung).

Das Automatisieren von Softwaretests kann hierbei große Vorteile bieten: Alle Tests die einmal in automatisierbarer Form spezifiziert wurden, können jederzeit, mit vertretbarem Aufwand, vollständig auf die Software angewendet werden. So kann nach jeder Änderung an der Software geprüft werden, ob alle Testfälle noch bestanden werden (regressives Testen), was ohne Automatisierung, wegen des hohen Aufwandes, nicht praktikabel wäre.

1.1 Motivation

Zum Automatisieren von Softwaretests existieren zahlreiche Werkzeuge: Unit-Test Frameworks, Systeme zur Automatisierung von GUI-Tests, Lasttest-Systeme, Werkzeuge zum Erstellen von Code-Metriken, etc. Da oftmals mehrere Werkzeuge von verschiedenen Herstellern eingesetzt werden, sind diese auf unterschiedliche Weise zu bedienen und liefern ihre Ergebnisse ebenfalls in unterschiedlichen Formaten zurück. Wünschenswert wäre eine Möglichkeit, alle in einem Projekt eingesetzten Testwerkzeuge über die gleiche Oberfläche zu bedienen und die Ergebnisse aller Tests in einem ausführlichen Test-/Qualitätsreport zusammenzufassen.

1.2 Ziel der Arbeit

Ziel der Arbeit ist es, ein System zu entwerfen, welches die Möglichkeit bietet, verschiedene Testwerkzeuge zu integrieren. Die Ergebnisse dieser Werkzeuge sollen in einem einheitlichen Qualitätsreport zusammengefasst werden. Bestandteil soll es sein, einen Prototyp zu implementieren, der die Funktionsweise dieses Systems demonstriert und die Machbarkeit belegt.

1.3 Inhalt der Arbeit

Kapitel 1: Einführung

Dieses Kapitel enthält eine kurze Einführung in die Arbeit und vermittelt einen Überblick über die in der Arbeit behandelten Themen.

Kapitel 2: Grundlagen

In diesem Kapitel werden einige grundlegende Themen bezüglich der Qualitätssicherung von Software beschrieben.

Kapitel 3: Vision

Dieses Kapitel beschreibt, was das Framework leisten soll, welche Anforderungen an das System gestellt werden und was dabei zu beachten ist.

Kapitel 4: Architektur

In diesem Kapitel werden die fachliche und die technische Architektur des Systems beschrieben.

Kapitel 5: Spezifikation

Dieses Kapitel enthält die Spezifikation für das Testframework. Die Funktionsweise des Systems wird beschrieben und durch UML-Diagramme verdeutlicht.

Kapitel 6: Prototyp

In diesem Kapitel wird die Erstellung des Prototyps beschrieben. Es enthält einen Überblick darüber, was der erstellte Prototyp bereits leistet und welche Funktionalitäten bis zur vollständigen Produktreife noch ergänzt werden müssen.

Kapitel 7: Ausblick

In diesem Kapitel wird darauf eingegangen, in welcher Weise das vorgestellte System noch verwendet werden kann und welche Möglichkeiten zur Erweiterung und Verbesserung bestehen.

Kapitel 8: Fazit

Dieses Kapitel enthält ein kurzes Fazit aus der Arbeit.

2 Grundlagen

2.1 Qualitätsmanagement

2.1.1 Definition

Nach [ISO8402] wird Qualitätsmanagement (QM) folgendermaßen definiert: „Alle Tätigkeiten der Gesamtführungsaufgabe, welche die Qualitätspolitik, Ziele und Verantwortungen festlegen, sowie diese durch Mittel der Qualitätsplanung, Qualitätslenkung, Qualitätssicherung und Qualitätsverbesserung im Rahmen des Qualitätsmanagements verwirklichen“.

Danach ist Qualitätsmanagement ein Oberbegriff und bezieht sich auf Planung, Lenkung, Prüfung und Verbesserung der Qualität eines Produktes, Prozesses oder einer Dienstleistung. QM kann und sollte in jeder Branche und jedem Betrieb eine Rolle spielen, da „Im Wettbewerb […] Produktqualität oft der zentrale Erfolgsfaktor [ist]“ [Wallmüller95 / S. 1].

2.1.2 Qualitätssicherung

Qualitätssicherung im Speziellen umfasst „alle geplanten und systematischen Tätigkeiten, die innerhalb des Qualitätsmanagementsystems verwirklicht sind, und die wie erforderlich dargelegt werden, um angemessenes Vertrauen zu schaffen, dass eine Einheit die Qualitätsanforderung erfüllen wird“ [ISO8402].

Der Begriff des Vertrauens bezieht sich hier sowohl auf den Kunden, als auch auf den Hersteller selbst [Balzert98 / S. 278].

2.2 Softwarequalität

2.2.1 Der Qualitätsbegriff

Die Bedeutung des Begriffes *Qualität* kann sehr unterschiedlich aufgefasst werden. [Balzert98 / S. 256] unterscheidet fünf verschiedene Ansätze, den Begriff aufzufassen:

- Der transzendente Ansatz
- Der produktbezogene Ansatz (Entwicklung)
- Der benutzerbezogene Ansatz (Marketing / Vertrieb)
- Der prozessbezogene Ansatz (Fertigung)
- Der Kosten-/Nutzen-bezogene Ansatz (Finanzen)

[DIN55350/11] definiert den Qualitätsbegriff entsprechend des produkt- und prozessbezogenen Ansatzes als „die Gesamtheit von Eigenschaften und Merkmalen eines Produktes oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse bezieht.“

Nach [ISO9126] wird Softwarequalität als „die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen“ definiert.

2.2.2 Produktorientiertes vs. prozessorientiertes QM

[Balzert98 / S. 279] unterscheidet außerdem zwischen *produktorientiertem Qualitätsmanagement*, welches sich mit dem Testen und Beurteilen der Qualität eines Softwareproduktes befasst, und *prozessorientiertem Qualitätsmanagement*, welches sich auf die angewendeten Methoden, verwendeten Werkzeuge, Standards und Richtlinien bezieht.

2.2.3 Qualitätsmodelle

Da allgemeine Definitionen des Softwarequalitätsbegriffs für praktische Anwendungen nicht anwendbar sind, schlägt [Balzert98 / S. 257 ff.] die Anwendung von *Qualitätsmodellen* vor. Der allgemeine Qualitätsbegriff wird mit Hilfe eines Qualitätsmodells „durch Ableiten von Unterbegriffen operationalisiert“ [Balzert / S. 257]. Ein Qualitätsmodell legt bestimmte Qualitätsmerkmale fest, deren Grad durch Qualitätsindikatoren (Metriken) ermittelt werden kann [Balzert98 / S. 268]. Beispiele solcher Qualitätsmodelle sind DIN ISO 9126, S. 3ff oder das von Hewlett-Packard entwickelte FURPS [Balzert98 / S. 260ff].

2.2.4 Qualitätssicherung

Analog zu prozess- und produktorientiertem QM unterscheidet [Balzert98] zwischen *konstruktiven* und *analytischen* Qualitätssicherungs(QS)-Maßnahmen. Bei konstruktiven Maßnahmen handelt es sich um „Methoden, Sprachen, Werkzeuge, Richtlinien, Standards und Checklisten“ [Balzert98 / S. 279], welche dafür sorgen, dass das Produkt bzw. der Entwicklungsprozess schon vor Beginn der Entwicklung bestimmte Eigenschaften besitzen, die zur Qualität des Produktes beitragen.

Analytische Maßnahmen haben die „Prüfung und Bewertung der Qualität“ [Balzert98 / S. 280] zum Ziel. [Balzert98] unterscheidet weiter zwischen *Analysierenden Verfahren* und *Testenden Verfahren*. Analysierende Verfahren „sammeln gezielt Informationen […] mit analytischen Mitteln“ [Balzert98 / S. 281]. Das zu testende Softwaresystem wird dabei nicht ausgeführt sondern nur analysiert. Zu diesen Verfahren gehört zum Beispiel das *Code-Review*, die *Statische Analyse* oder der *Walkthrough.* Bei testenden Verfahren wiederum wird die Software ausgeführt und geprüft, ob sich das System der Spezifikation entsprechend verhält.

Das System, welches in dieser Arbeit spezifiziert wird, kann als konstruktive QS-Maßnahme aufgefasst werden. Bei den Prüfmethoden, die von dem System angewendet werden können handelt es sich aber ausschließlich um analytische QS-Maßnahmen.

3 Vision / Anforderungsanalyse

3.1 Problemsituation

Zum automatisierten Testen von Software stehen einem Entwicklungsteam viele Werkzeuge zur Verfügung, die unterschiedliche Aspekte eines Softwaresystems testen. Üblicherweise werden in einem Softwareentwicklungsprojekt mehrere dieser Werkzeuge eingesetzt. Ein Tester oder Entwickler muss jedes dieser Werkzeuge einzeln, auf unterschiedliche Weise, bedienen und erhält mehrere Ausgaben, die sich in ihren Formaten unterscheiden. Die Erstellung eines zusammengefassten Qualitätsreports mit den Ergebnissen aller durchgeführten Tests gerät damit zu einer Aufgabe, die in dem zeitlich engen Rahmen der meisten Projekte meist nicht durchführbar ist. Weiterhin wird die regelmäßige Durchführung bestimmter Tests von Entwicklern oftmals vernachlässigt, wenn für einen vollständigen Testlauf beispielsweise fünf verschiedene Werkzeuge bedient werden müssen. Der Projektleitung ist es nicht mit vertretbarem Aufwand möglich, aus der Fülle an unterschiedlichen Ausgabedateien einen Überblick über den aktuellen Qualitätsstatus und die Qualitätsentwicklung des Projektes zu erhalten. Auch die Einführung neuer Testwerkzeuge stellt ein Problem dar, da jeder Entwickler und Tester in der Bedienung des neuen Werkzeuges geschult werden muss.

3.2 Grundsätzliche Anforderungen (Muss-Anforderungen)

Ein System, welches die oben beschriebenen Probleme löst, sollte folgenden Anforderungen genügen:

3.2.1 Einfache Integration neuer Werkzeuge (G001)

Die Integration neuer Werkzeuge sollte mit möglichst geringem Aufwand erfolgen können, damit die Einführung neuer Testarten oder die Umstellung auf andere Produkte keine größere Hürde darstellt und flexibel auf neue Qualitätsrichtlinien oder spezielle Qualitätsanforderungen für bestimmte Projekte reagiert werden kann.

3.2.2 Geringe Anforderungen an Testwerkzeuge (G002)

Es sollen unterschiedliche Werkzeuge in das System integriert werden können. Die Anforderungen an Werkzeuge, die integriert werden können, sollten möglichst gering sein, so dass bei der Auswahl von Werkzeugen nur wenige Einschränkungen bestehen.

3.2.3 Erstellung eines zusammenfassenden Qualitätsreports (G003)

Das System muss eine Ausgabe erzeugen, welche die Ergebnisse aller Werkzeuge in einem einzigen Dokument zusammenfasst, so dass es möglich ist, einen Qualitätsreport zu erhalten, der den Qualitätsstatus eines Softwareprojektes für einen bestimmten Zeitpunkt beschreibt.

3.2.4 Flexibles Format des Qualitätsreports (G004)

Der Qualitätsreport sollte in einem möglichst flexiblen, maschinenlesbaren Format erstellt werden, so dass die Weiterverarbeitung, Aufbereitung oder Analyse des Reports mit geringem Aufwand erfolgen kann. Eine Ausgabe als HTML-Dokument oder als einfache, unstrukturierte Textdatei wäre ungeeignet.

3.2.5 Einfache Bedienung durch Entwickler und Tester (G005)

Die Bedienung des Systems durch Entwickler und Tester sollte möglichst einfach gestaltet sein. Im Idealfall sollten diese Benutzer nur das Projekt auswählen müssen, welches getestet werden soll und den Testlauf starten, ohne dass weitere Informationen erfasst werden müssen. Die Testkonfiguration für ein bestimmtes Projekt, die Auswahl der angewendeten Testarten bzw. Testwerkzeuge und gegebenenfalls die Konfiguration bestimmter Parameter (z.B. Quellpfade oder Testfälle), kann durch einen einzigen Qualitätsbeauftragten oder Testleiter erfolgen. Dadurch würden Entwickler entlastet und wären dennoch in der Lage, vollständige, den Qualitätsanforderungen entsprechende Tests durchzuführen, ohne über die Details der Testdurchführung geschult zu werden.

3.3 Marktanalyse

Mit den oben beschriebenen grundsätzlichen Anforderungen als Entscheidungsgrundlage wurde eine Marktanalyse durchgeführt, um zu ermitteln, ob bereits eine Software auf dem Markt existiert, welche diese Anforderungen erfüllt. Hierzu wurden diverse Produkte untersucht, von denen die folgenden drei der Idee am nächsten kommen.

1. TechExcel DevSuite.net	
Produktname	DevSuite.net
Hersteller	TechExcel Inc.
Website	http://www.devsuite.net
Beschreibung	DevSuite.net von TechExcel ist eine Web-basierte Plattform zum Tracken und Verwalten von Fehlern, Testfällen und anderen anfallenden Aufgaben in einem Softwareprojekt. Schwerpunkt liegt hier auf der Verwaltung manueller Tests, es ist aber eine Integration des Produktes *TestComplete* (Siehe unten) möglich.
Erfüllung der Anforderungen	
G001	**Nein**, das System kann nicht durch Integration von Testwerkzeugen um Tests erweitert werden.
G002	**Nein**, das System kann nicht durch Integration von Testwerkzeugen um Tests erweitert werden.
G003	**Nein**, das System erstellt keinen zusammenfassenden Qualitätsreport.
G004	**Nein**, das System erstellt keinen zusammenfassenden Qualitätsreport.
G005	**Nein**, das System bietet für alle Benutzer eine komplexe Schnittstelle zum Hinzufügen und Bearbeiten von Testfällen.

2. Seapine TestTrack Studio	
Produktname	TestTrack Studio 2008
Hersteller	Seapine Software Inc.
Website	http://www.seapine.com/ttstudio.html
Beschreibung	TestTrack Studio ist eine Applikation ´zur Verwaltung von manuellen Testfällen und zur Fehlerverwaltung.
Erfüllung der Anforderungen	
G001	**Nein**, das System kann nicht durch Integration von Testwerkzeugen um Tests erweitert werden.
G002	**Nein**, das System kann nicht durch Integration von Testwerkzeugen um Tests erweitert werden.
G003	**Ja**, das System kann beliebige, konfigurierbare Reports erstellen.
G004	**Ja**, die Reports werden im XML-Format erstellt und können beliebig weiterverarbeitet oder transformiert werden.
G005	**Nein**, das System bietet für alle Benutzer eine komplexe Schnittstelle zum Hinzufügen und Bearbeiten von Testfällen.

3. AutomatedQA TestComplete	
Produktname	TestComplete 6
Hersteller	AutomatedQA Corp.
Website	http://www.automatedqa.com/products/testcomplete/
Beschreibung	Das Produkt TestComplete von AutomatedQA bietet eine Fülle unterschiedlicher Testarten an. Darunter manuelle Tests, Unit-Tests und Lasttests. Über eine Scriptsprache können unterschiedlichste Tests automatisiert werden.
Erfüllung der Anforderungen	
G001	**Ja**, das System bietet die Möglichkeit Plugins zu verwenden, mit denen neue Testarten definiert werden.
G002	**Ja**, prinzipiell kann jedes Werkzeug integriert werden, welches automatisierbar ist.
G003	**Ja**, das System erstellt einen zusammenfassenden Qualitätsreport.
G004	**Ja**, das System erstellt einen Qualitätsreport im XML-Format, der entsprechend weiterverarbeitet oder transformiert werden kann.
G005	**Nein**, das System bietet für alle Benutzer eine hochkomplexe Benutzerschnittstelle, die nur von ausgiebig geschulten Benutzern verwendet werden kann.

3.3.1 Zusammenfassung

Wie aus den Tabellen sichtbar wird, erfüllt das Produkt *TestComplete* die meisten Anforderungen. Leider ist dieses System nicht dafür ausgelegt, die definierten Testfälle autonom, ohne Benutzerinteraktion durchzuführen. Es bietet zwar die Möglichkeit über die Kommandozeile mit Startparametern aufgerufen zu werden, aber auch hierbei öffnet sich grundsätz-

lich die grafische Benutzeroberfläche, welche durchaus komplex ist und damit der Idee einer möglichst simplen Oberfläche für Tester und Entwickler, wiederspricht. Eine ausführliche Schulung zum Umgang mit dem System wäre notwendig. Gefordert wurde aber, dass ein einzelner, versierter Qualitätsbeauftragter die Testfälle konfiguriert und jeder Entwickler / Tester über eine einfache Start-Schaltfläche diese Tests durchführen kann.

3.4 Zusätzliche Anforderungen

Da sich aus der Marktanalyse kein Produkt ergeben hat, welches den geforderten, grundsätzlichen und zwingend erforderlichen Anforderungen genügt, ist die Notwendigkeit zur Erstellung eines solchen Systems gegeben. Da ein neues System entworfen und realisiert werden muss, ist es möglich weitere Anforderungen zu definieren, die den Wert eines solchen Systems im alltäglichen Testbetrieb und die Wiederverwendbarkeit der erstellten Softwaremodule erhöhen.

3.5 Plugin-Schnittstelle (F001)

Das zu erstellende System muss grundsätzlich eine einfache Integration neuer Werkzeuge ermöglichen (G001). Hierfür eignet sich eine Plugin-Architektur (*Steckbarer Adapter* [Gamma96 / S. 176]). Die Plugin-Schnittstelle soll für alle Testwerkzeuge die Möglichkeit zur Verfügung stellen, ihre Testergebnisse in einem einheitlichen Format in den Qualitätsreport zu schreiben. Die einzelnen Plugins dienen somit dazu, die Testwerkzeuge auszuführen und die Ausgaben in ein einheitliches Format zu konvertieren. Jedes Modul verfügt über eine Ausführungsmethode, welche ein bestimmtes Werkzeug ausführt und die Ausgabe des Werkzeuges in einem einheitlichen Format zurückliefert.

3.5.1 Ausführungsparameter (F002)

Die einzelnen Plugins benötigen zur Ausführung verschiedene Informationen, zum Beispiel die Angabe, unter welchem Dateisystempfad ein Testobjekt zu finden ist, oder ähnliches. Diese Informationen sollen beim Start eines Testlaufs an das System übergeben werden können.

3.5.2 Nicht-Test Plugins (F003)

Um das System möglichst flexibel zu machen und eine einfache Möglichkeit zur Erweiterung oder Anpassung zu geben, soll es sich bei den Plugins nicht gezwungenermaßen um Testplugins handeln, die ein bestimmtes Testwerkzeug aufrufen. So wäre es beispielsweise denkbar, ein Plugin zu implementieren, welches lediglich Informationen über die Testmaschine (Name, CPU, Speicher, OS und ähnliches) ermittelt und in das Protokoll schreibt. Auch könnten Plugins erstellt werden, die eine Workflow-Engine zur Abbildung von komplexeren Abläufen verwenden.

3.5.3 Pluginabhängigkeiten (F004)

Plugins können von einander abhängig sein, was die Reihenfolge der Ausführung betrifft. So könnte es zum Beispiel vorkommen, dass ein Plugin A existiert, welches den aktuellen Sourcecodestand kompiliert und ein anderes Plugin B, welches die Kompilate testet. In diesem Fall muss das Plugin A vor dem Plugin B ausgeführt werden.

3.5.4 Datenaustausch zwischen Plugins (F005)

Es kann vorkommen, das Plugins Informationen beschaffen, die von anderen Plugins verwendet werden müssen. Beispielsweise könnte ein Plugin integriert werden, welches den Buildvorgang für ein bestimmtes Projekt durchführt und die erzeugten Binärdateien in einem bestimmten Verzeichnis ablegt. Ein weiteres Plugin, beispielsweise ein Unit-Test-Plugin müsste dieser Pfad bekannt gemacht werden, damit es die Unit-Tests auf den erstellten Kompilaten durchführen kann. Hierzu muss ein einfacher Mechanismus geschaffen werden, der es ermöglicht, Informationen von einem Plugin an ein anderes weiterzugeben.

3.5.5 Nebenläufigkeit der Plugins (F006)

Es sollen auch langlebige Testmodule, wie zum Beispiel Workflows mit Benutzerinteraktion unterstützt werden. So wäre es denkbar, einen manuellen Test zu integrieren, der einem verantwortlichen Tester ein Testformular mit Testfällen zur Verfügung stellt, welches dieser Tester dann im Laufe einiger Tage bearbeiten soll. Falls die Plugins rein sequentiell ausgeführt werden würden, könnte es vorkommen, dass ein Test, der nur eine halbe Stunde Laufzeit hat, mehrere Tage warten müsste, bevor er ausgeführt wird. Um diesen Fall zu verhindern, sollen alle Plugins nebenläufig ausgeführt werden. Hierbei ist besonders im Hinblick auf die Anforderung F004 (Pluginabhängigkeiten) gegebenenfalls eine Synchronisation der voneinander abhängigen Plugins nötig.

3.6 Nebenläufigkeit der Testläufe (F007)

Um Entwicklungsteams sinnvoll bei ihrer Arbeit zu unterstützen, muss das System es ermöglichen, mehrere Testläufe gleichzeitig auszuführen: Einerseits arbeiten eventuell mehrere Teams an unterschiedlichen Projekten und sollen sich nicht gegenseitig ausbremsen und andererseits führen unterschiedliche Entwickler oder Tester möglicherweise verschiedene Testläufe für dasselbe Projekt aus. Das System soll die nebenläufige Ausführung von mehreren Testläufen ermöglichen, wobei für jeden Benutzer zu jedem Zeitpunkt immer nur ein Testlauf pro Projekt möglich ist.

3.7 Der Testserver (F008)

Verschiedene Überlegungen führen zu der Anforderung, dass eine Verteilung des Systems auf einen Testserver und mehrere Testclients vorteilhaft ist:

- **Langlebige Testserien –** Es kann vorkommen, dass einige Tests sehr lange brauchen, bis sie beendet sind. Der Client des Testers dürfte während dieser Zeit nicht heruntergefahren werden.
- **Lastenverteilung –** Auf einem Entwicklungsrechner laufen üblicherweise viele unterschiedliche Applikationen, die vom Entwickler verwendet werden. Einerseits kann der Entwickler in seiner Arbeit behindert werden, wenn er lokal einen Testlauf ausführt, andererseits wird die Durchführung der Testserie verzögert, wenn andere Programme zusätzlich ausgeführt werden.

- **Dienstverteilung –** Durch die Verteilung wird es möglich den Testserver als unabhängigen Dienst im Netzwerk anzubieten. Hierdurch könnte beispielsweise eine Weboberfläche als Benutzerschnittstelle verwendet werden, ohne dass der Testserver auch auf dem Webserver installiert werden muss.

Aufgrund dieser Überlegungen soll das System verteilt realisiert werden: Die Tests werden auf einem dedizierten Testserver ausgeführt und die Ergebnisse der Tests können von dort abgerufen werden.

3.8 Versionierung (F009)

Da die Konfiguration des Systems und die einzelnen Testfälle eines Entwicklungsprojektes ständigen Änderungen unterworfen sind, ist es unabdingbar, dass der Qualitätsreport auch Informationen darüber enthält, auf welchen Stand der Testkonfiguration sich der Report bezieht. Hierzu muss jedes Plugin drei Versionsnummern erhalten:

- Die Version des eigentlichen Plugins
- Die Version des vom Plugin ausgeführten Werkzeuges bzw. die Version des vom Plugin verwendeten Subsystems.
- Die Version der verwendeten Testfälle

3.9 Isolation (F010)

Falls ein Plugin bei seiner Ausführung einen Fehler erkennt oder selbst fehlerhaft ist und einen Fehler auslöst, wird die Ausführung des Plugins unerwartet abgebrochen. Dies darf keinen Einfluss auf andere Plugins haben, die von dem Plugin unabhängig sind. Der Testlauf soll, falls in einem Plugin ein Fehler auftritt oder erkannt wird, weiterlaufen und alle Plugins ausführen, die nicht von diesem Plugin abhängig sind.

3.10 Fehlerbehandlung (F011)

Wenn ein Plugin während seiner Ausführung einen Fehler erkennt oder auslöst und nicht ordnungsgemäß beendet wurde, muss sichergestellt sein, dass der Benutzer nach Beendigung des Testlaufs darüber informiert und auch die Fehlermeldung an den Benutzer weitergegeben wird.

3.11 Tabelle der Anforderungen

Nr	Titel	Beschreibung	Klassifizierung
Grundsätzliche Anforderungen			
G001	Einfache Integration von Testwerkzeugen	Die Integration neuer Werkzeuge in das System soll mit minimalem Aufwand erfolgen können.	Must
G002	Geringe Anforderungen an Testwerkzeuge	Die Anforderungen an Testwerkzeuge, die verwendet werden können, sollen möglichst gering sein.	Must
G003	Erstellung eines zusammenfassenden Qualitätsreports	Das System soll einen Qualitätsreport erstellen, der die Ergebnisse aller Tests zusammenfasst.	Must
G004	Flexibles Format für den Qualitätsreport	Der Qualitätsreport soll in einem flexiblen, maschinenlesbaren Format erstellt werden.	Must
G005	Einfache Benutzerschnittstelle	Die Benutzerschnittstelle zum Starten und Verwalten von Testserien soll möglichst einfach gestaltet sein.	Must
Zusätzliche Anforderungen			
F001	Plugin-Schnittstelle	Das Hinzufügen neuer Testmodule soll erfolgen können, ohne dass das bestehende System in irgendeiner Weise verändert werden muss.	Should
F002	Ausführungsparameter	Es soll eine Möglichkeit geben, beim Starten eines Testlaufs beliebige Parameter an das System zu übergeben, welche von den Modulen als Ausführungsargumente verwendet werden können.	Should
F003	Nicht-Test Plugins	Es soll möglich sein, Plugins zu entwickeln, die keine Testwerkzeuge ausführen sondern beispielsweise Workflows über eine Workflow-Engine starten oder nur Informationen über die verwendete Testmaschine ermitteln.	Should
F004	Pluginabhängigkeiten	Es soll möglich sein, Abhängigkeiten zwischen Testmodulen zu berücksichtigen, so dass die Reihenfolge in der die Module ausgeführt werden, bestimmt werden kann.	Should
F005	Datenaustausch zwischen Plugins	Es soll möglich sein, dass Plugins Informationen an andere Plugins weitergeben, so dass ein Informationsaustausch zwischen den Plugins ermöglicht wird.	Should
F006	Nebenläufigkeit der Plugins	Die einzelnen Plugins eines Testlaufs sollen nicht sequenziell, sondern nebenläufig ausgeführt werden.	Should
F007	Nebenläufigkeit der Testläufe	Es soll möglich sein, mehrere Testläufe gleichzeitig durchzuführen.	Should

F008	Der Testserver	Es soll einen dedizierten Testserver geben, auf dem alle Testläufe durchgeführt werden.	Should
F009	Versionierung	Die Version der Plugins soll durch drei Versionsnummern beschrieben werden: • Die Versionsnummer des verwendeten Werkzeuges oder Subsystems. • Die Versionsnummer des Plugins • Die Versionsnummer der Testfälle / Testkonfiguration	Should
F010	Isolierung	Plugins, die voneinander unabhängig sind, dürfen sich nicht gegenseitig durch Fehler beeinträchtigen	Must
F011	Fehlerbehandlung	Fehler, die bei der Ausführung von Plugins aufgetreten sind, sollten dem Benutzer gemeldet werden.	Should

4 Architektur

4.1 Grundsätzliche Architekturelemente

Die im Kapitel 3 festgelegten Anforderungen ermöglichen es, folgende Module für die Architektur der Anwendung zu identifizieren:

- **Grafische Benutzeroberfläche**
  Das System soll eine grafische Benutzeroberfläche zur Verfügung stellen, über die ein Entwickler oder Tester auf einfache Weise bereits definierte Testfälle auf das zu testende System anwenden kann. (Anforderung G003 – „Einfache Benutzerschnittstelle“).
- **Plugin-Verwaltung**
  Gemäß der Anforderung F001 (Plugin-Schnittstelle) muss ein Modul existieren, welches die einzelnen Plugins lädt und verwaltet.
  Es existieren einige Plugin-Frameworks auf dem Markt, welche eine umfangreiche Plattform für Plugin-basierte Systeme zur Verfügung stellen. Im .NET-Bereich sei vor allem auf das Open-Source Projekt *Plux* hingewiesen. Nach eingehender Analyse stellte sich aber heraus, dass der Einsatz eines solchen, mächtigen Frameworks das System unnötig kompliziert macht. Plugin-Frameworks wie *Plux* sind darauf ausgelegt, als Basis für vollständig Plugin-basierte Systeme verwendet zu werden. Sie ermöglichen es, beliebig viele unterschiedliche Plugin-Schnittstellen zu definieren und Plugins aufeinander aufzubauen. Also Plugins, die wiederum Plugins verwenden usw. Da für das hier spezifizierte System nur ein einziger Plugin-Typ benötigt wird, ist es im Hinblick auf die Komplexität und Wartbarkeit des Systems pragmatischer, eine einfache Plugin-Schnittstelle zu implementieren.
- **Ausführungsverwaltung**
  Die Anforderungen F003-F007 (Nebenläufigkeit, Pluginabhängigkeiten, Datenaustausch zwischen Plugins), sowie F010 und F011 (Isolation und Fehlerbehandlung), legen ein Modul nahe, welches die Ausführung der Testserien und Plugins verwaltet und alle Anforderungen an die Ausführungslogik erfüllt.
- **Protokollverwaltung**
  Die Anforderungen G003 (Erstellung eines zusammenfassenden Qualitätsreports) und G004 (Flexibles Format für den Qualitätsreport) suggerieren ein Modul, welches die Verwaltung des globalen Protokolls realisiert und es den einzelnen Plugins ermöglicht, ihre Ausgabe einheitlich zu protokollieren.

4.2 Fachliche Architektur

In Abbildung 1 ist die fachliche Architektur des Frameworks mit den oben beschriebenen Modulen dargestellt. Prinzipiell ist das System aus drei Schichten aufgebaut: Eine grafische Benutzerschnittstelle, die eigentliche Testverwaltung, welche die Ausführungsverwaltung und die Protokollverwaltung enthält, sowie die Pluginverwaltung mit den verschiedenen Testmodulen (Plugins).

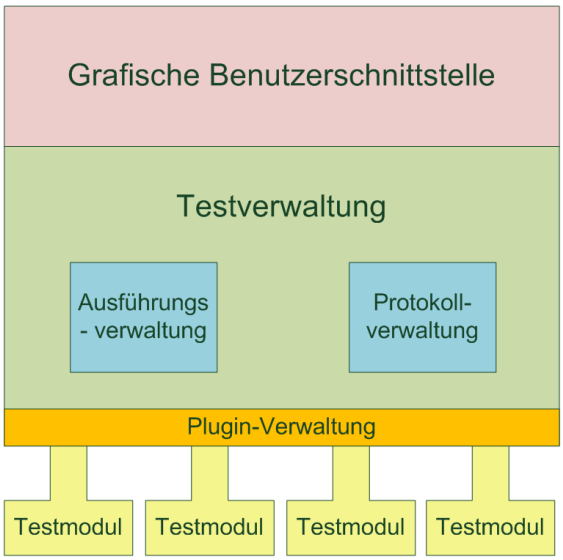


Abbildung 1: Fachliche Architektur

4.2.1 Die grafische Benutzerschnittstelle

Die grafische Benutzerschnittstelle ermöglicht dem Benutzer einen komfortablen Zugriff auf das System und stellt eine Oberfläche zur Verfügung, über die Testserien verwaltet und Testergebnisse abgerufen werden können.

4.2.2 Testverwaltung

Die Testverwaltung stellt das eigentliche Framework dar und besitzt verschiedene Verantwortlichkeiten:

- **Laden und Ausführung der Testmodule (Plugins)** – Die Testverwaltung lädt die verschiedenen Testmodule und führt diese aus.
- **Realisierung der Nebenläufigkeit** – Die Testverwaltung muss für die Module und für die Testserien einen Ausführungskontext zur Verfügung stellen, in dem die Nebenläufigkeit der Testserien und der voneinander unabhängigen Module sichergestellt wird.
- **Verwaltung der Abhängigkeiten** – Die Testverwaltung ist dafür zuständig, die Module in der Reihenfolge auszuführen, in der sie voneinander abhängig sind und dabei den Austausch von Informationen zwischen den Modulen zu ermöglichen.
- **Isolation / Fehlerbehandlung** – Die Testverwaltung muss sicherstellen, dass die Module isoliert voneinander ausgeführt werden, so dass ein Fehler innerhalb eines Moduls keinen Einfluss auf die Ausführung der gesamten Testserie haben kann. Zusätzlich muss die Testverwaltung Fehler, die in Modulen aufgetreten sind, in soweit behandeln, dass die Fehlerdetails nach der Beendigung der Testserie zur Verfügung stehen.
- **Erstellung und Verwaltung des Qualitätsreports** – Die Testverwaltung muss eine Datenstruktur zur Verfügung stellen, in welche die Module ihre Testergebnisse schreiben können.

4.3 Technische Architektur

Die in Abbildung 2 dargestellte technische Architektur ergänzt die fachliche Architektur um einige Komponenten, die aus technischer Sicht nötig sind. Sie führt außerdem die Abstraktionsschicht *Protokollengine* ein, welche weiter unten beschrieben ist und erweitert die Architektur um eine Verteilungsfunktionalität, um die Möglichkeit eines dedizierten Testservers bereitzustellen.

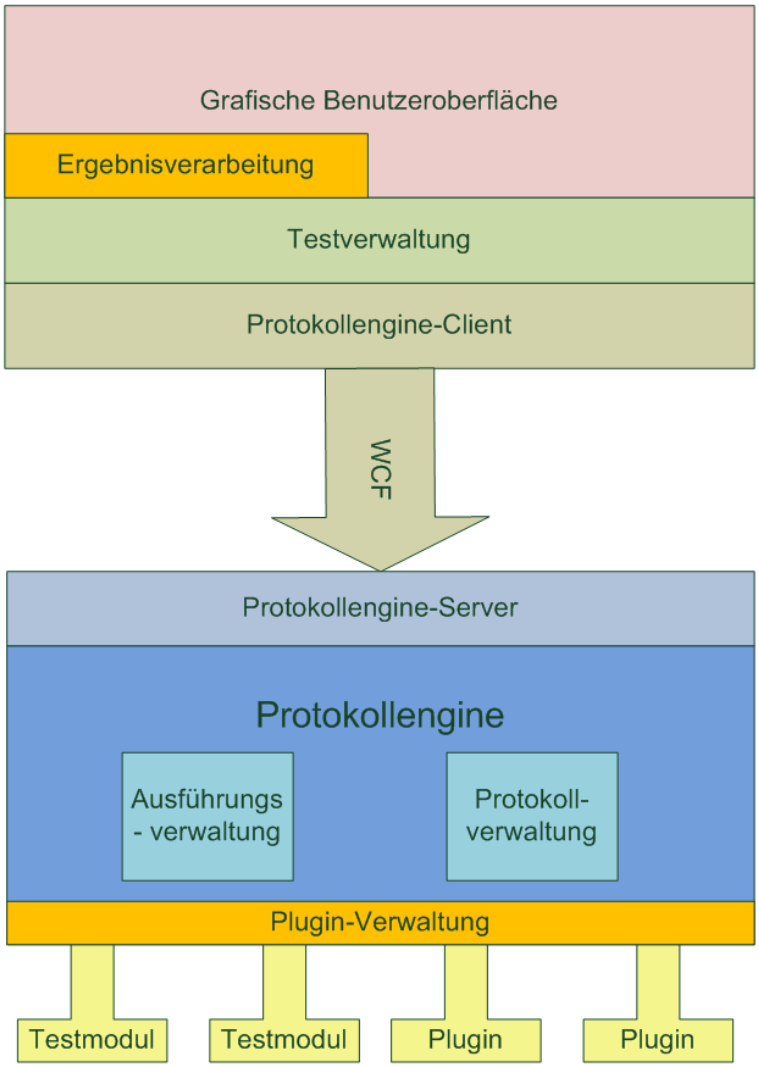


Abbildung 2: Technische Architektur

4.3.1 Ergebnisverarbeitung

Durch die Nebenläufigkeit der Testserien ist es nötig, die Ergebnisse derselben explizit aus dem System abzurufen, sobald eine Testserie beendet wurde. Die Ergebnisverarbeitung ist hierfür zuständig. Sie prüft in regelmäßigen Abständen, ob beendete Testserien vorhanden sind und ruft, falls dies der Fall ist, das erstellte Protokoll aus dem System ab. Anschließend legt sie dieses in einem geeigneten Speicher ab (Webserver, Datenbank, Netzwerklaufwerk). Der Benutzer, welcher den Test gestartet hat, kann dann jederzeit auf die Ergebnisse des Testlaufs zugreifen.

4.3.2 Protokollengine

Bei der Protokollengine handelt es sich um eine Abstraktion der ursprünglichen Testverwaltung, wie sie in der fachlichen Architektur beschrieben wurde. Sie ist dafür verantwortlich, die Module zu laden, ein Protokoll zur Verfügung zu stellen und die Ausführung der Module, inklusive Nebenläufigkeit, Isolation und Modul zu Modul Kommunikation zu realisieren. Im Prinzip besitzt sie die selben Aufgaben wie die Testverwaltung aus der fachlichen Architektur, mit der Ausnahme, dass sie nicht spezifisch für (Software-)Tests ist, sondern für jede Situation eingesetzt werden kann, in der ein Protokoll durch Ausführung unterschiedlicher Module erstellt werden soll. Dies erhöht zum einen die Wiederverwendbarkeit des Systems und zum anderen wird dadurch eine klare Komponentenverantwortlichkeit geschaffen, die zum Verständnis des Systems beitragen soll.

4.3.3 Testverwaltung

Die Testverwaltung bietet eine für (Software-)Tests spezifische Sicht auf die Protokollengine und kann als konkrete, testspezifische Realisierung des abstrakten Konzeptes verstanden werden, welches durch die Protokollengine dargestellt wird. Sie bietet dieselbe Funktionalität wie die Protokollengine an, nur dass hier mit Testmodulen anstatt mit abstrakten Modulen, und mit Testserien anstatt mit Protokolljobs gearbeitet wird. Die Ergebnisse der Testmodule, die in das Protokoll geschrieben wurden, werden an dieser Stelle als Testergebnisse präsentiert und der Benutzer dieser Schicht kann daher direkt auf die Testergebnisse zugreifen, ohne eine Interpretation abstrakterer Informationen vorzunehmen.

4.3.4 Das Microsoft® .NET-Framework

Der Prototyp soll auf Basis des .NET-Frameworks von Microsoft realisiert werden. Es handelt sich hierbei um eine Softwareplattform, welche sowohl eine Laufzeitumgebung (mit Garbage-Collection, Code-Security und weiteren Features), als auch eine umfangreiche Klassenbibliothek zur Verfügung stellt.

Das .NET-Framework stellt eine Implementation des ISO/IEC/ECMA-Standards *Common Language Infrastructure* dar. Eine weitere bekannte Implementation dieses Standards ist das von Novell entwickelte *Mono*, welches es ermöglicht, .NET-Programme auch auf anderen Betriebssystemen wie Linux oder Mac OS X auszuführen. Prinzipiell ist das hier vorgestellte System somit plattformunabhängig.

4.3.5 Protokollengine-Server/Client und WCF

Die Komponenten Protokollengine-Server und Protokollengine-Client sowie die Übertragungstechnologie WCF werden für die Verteilung des Systems benötigt. WCF ist die Abkürzung für Windows Communication Foundation, ein Framework für verteilte Systeme, welches von Microsoft kostenlos zur Verfügung gestellt wird und Teil des .NET Frameworks ab der Version 3.0 ist. Es gibt ebenfalls eine Implementation der WCF für *Mono*, so dass eine Verwendung dieser Technologie nicht plattformspezifisch ist. WCF bietet eine transparente, verteilte Verwendung von .NET-Objekten über verschiedene Übertragungswege, wie zum Beispiel TCP/IP, http/SOAP(Webservices) und Interprozesskommunikation. Durch Einsatz

der WCF kann der Übertragungsweg, der für die Verteilung verwendet wird, verändert werden, ohne dass eine Anpassung des Systems nötig ist.

Zur Realisierung dieser Verteilung werden ein Server und ein Client implementiert. Der Client präsentiert sich der darüber liegenden Schicht, der Testverwaltung, als Protokollengine-Schnittstelle, so dass die Verteilung für diese Schicht völlig transparent erfolgt. Der Client nimmt die Aufrufe für die Protokollengine von der Testverwaltung entgegen und überträgt diese auf einem WCF-Kanal. Ein WCF-Kanal ist die Abstraktion eines Übertragungsweges, wobei es sich um beliebige Protokolle, wie TCP/IP, http/SOAP (Webservices) oder andere handeln kann. Im Client muss nicht festgelegt werden, welcher Übertragungsweg verwendet wird. In der Konfiguration der Anwendung kann dies beliebig angepasst werden.

Der Server nimmt die Anforderungen aus dem WCF-Kanal entgegen und ruft die entsprechenden Methoden auf der Protokollengine auf. Anschließend werden die Rückgabewerte der Protokollengine-Methode auf dem WCF-Kanal zurückübertragen.

4.3.6 Testmodule und Plugins

In dem Architekturdiagramm in Abbildung 2 werden für die Module zwei unterschiedliche Beschriftungen verwendet: Testmodul und Plugin. Dies soll verdeutlichen, dass sowohl Testmodule, die Testwerkzeuge ausführen und deren Ergebnisse zurückliefern, als auch allgemeinere Plugins, die zum Beispiel eine Kompilierung des Sourcecodes durchführen, an dieser Stelle in das System integriert werden können.

5 Systemspezifikation

5.1.1 Protokollengine

Die Protokollengine realisiert die Hauptfunktionalität des Systems, ist dabei aber nicht spezifisch für Softwaretests, sondern stellt eine Abstraktion der Protokollierungs- funktionalität dar. Aufgabe dieser Komponente ist das Laden ausführbarer Module, das Ausführen dieser Module und die Verwaltung der so erzeugten Protokolle.

Die Schnittstelle, die von diesem Subsystem angeboten wird, enthält alle Methoden zur Verwaltung von Protokollierungsläufen, die als *Protocoljobs* oder einfach nur *Jobs* bezeichnet werden.

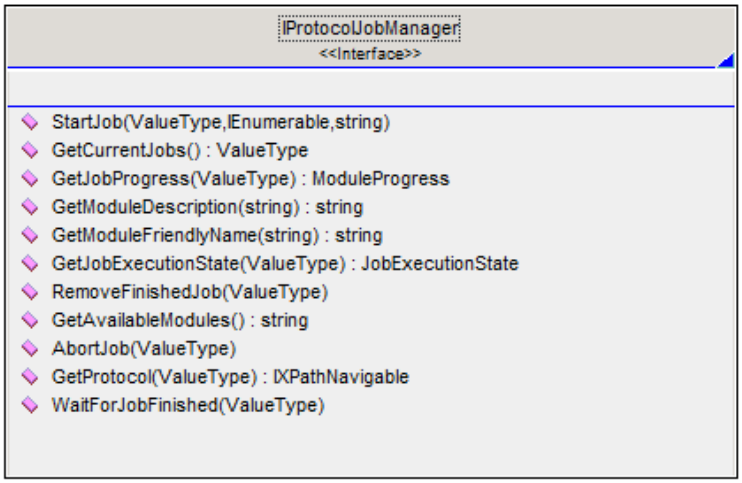


Abbildung 3: Protokollengine-Schnittstelle

Die gesamte Funktionalität, die von dieser Komponente angeboten wird, ist durch das Interface *IProtocolJobManager* definiert. Abhängige Komponenten sollten grundsätzlich nur gegen dieses Interface programmiert werden, damit eine Substitution, beispielsweise für die Verteilung des Systems, möglich ist.

Mit der `StartJob()`-Methode wird ein neuer Protokolljob gestartet, wobei eine ID angegeben wird, durch welche der Job später identifiziert werden kann. Um mehrere Jobs auf Benutzer- und Projektebene zu ermöglichen, sollte diese ID aus einer Benutzerkennung und einer Projektkennung zusammengesetzt werden, theoretisch kann aber von dem aufrufenden System eine beliebige ID vergeben werden. Ein Protokolljob kann nur gestartet werden, wenn noch kein Job mit derselben ID ausgeführt wird. Darüber hinaus muss der Methode eine Liste mit Modulen (Plugins) übergeben werden, die für diesen Job verwendet werden sollen. So kann der Benutzer entscheiden, welche Tests für den Testlauf ausgeführt werden sollen.

Die Schnittstelle verfügt über weitere Methoden zur Verwaltung der Protokolljobs. Ein aktiver Job kann durch Aufrufen der Methode `AbortJob(),` unter Angabe der Job-ID abgebrochen werden. `GetJobExexutionState()` ermöglicht es, den aktuellen Ausführungsstatus eines Jobs abzurufen. Aus dem Rückgabewert kann ermittelt werden, welche Module (Plugins) noch ausgeführt werden, der prozentuale Fortschritt einzelner Module kann abgerufen werden und für Module, die ihre Ausführung mit einem Fehler abgebrochen haben, kann die Fehlermeldung abgerufen werden.

Das Interface besteht ausschließlich aus Methoden. Auf andere Membertypen, wie Eigenschaften oder Ereignisse, die vom .NET-Framework ebenfalls für Interfaces zugelassen sind, wurde verzichtet. Diese Maßnahmen, eine schmale Schnittstelle, die Definition eines Interfaces und der Verzicht auf erweiterte Membertypen, dienen dazu, eine Substitution der Protokollengine durch eine Proxy-Komponente für die Verteilung des Systems zu ermöglichen und dabei die Anforderungen an den verwendeten Transportmechanismus möglichst gering zu halten. So kann beispielsweise die geplante Verteilung durch einen Transportmechanismus realisiert werden, der keine Callbacks (Ereignisse) unterstützt. Dies ermöglicht die Verwendung eines Webservices auf HTTP-Basis.

Durch dieses Design erfolgt der verteilte Zugriff auf die Protokollengine für abhängige Komponenten vollständig transparent, Details des Transportes bleiben verborgen. Außerdem kann die Implementierung eines beliebigen Transportweges erfolgen, ohne dass die Protokollengine oder abhängige Komponenten verändert werden müssen.

5.1.2 Protokolljobs

Ein Protokolljob kann sich in einem der drei Ausführungsstatus *NotExecuted, Running* oder *Finished* befinden. Im Status *NotExecuted* wird der Job von der Protokollengine zur Ausführung vorbereitet, im Status *Running* wird der Protokolljob gerade ausgeführt. Ein Protokolljob, der abgeschlossen wurde, bleibt solange erhalten, bis er explizit durch die Methode `RemoveFinishedJob()` entfernt wurde. Dies ermöglicht dem aufrufenden System, die Ergebnisse des Protokolljobs abzurufen und ist nötig, da zum einen die Protokolljobs nebenläufig ausgeführt werden und zum anderen auf eine Benachrichtigungsfunktion (Callback, Event) verzichtet wurde, um die Verteilung des Systems zu vereinfachen. Üblicherweise würde ein Aufrufer über die Methode `GetProtocol()` das fertige Protokoll abrufen, die Ergebnisse verarbeiten, eventuell das Protokoll speichern und anschließend durch einen Aufruf von `RemoveFinishedJob()den Job` aus der Engine entfernen. Das Starten eines neuen Protokolljobs mit einer ID, die schon von einem aktiven Job verwendet wird, ist nicht möglich, auch wenn sich der Job bereits im Zustand *Finished* befindet. Darüber hinaus kann ein Job jederzeit über die Methode `AbortJob()` abgebrochen werden.

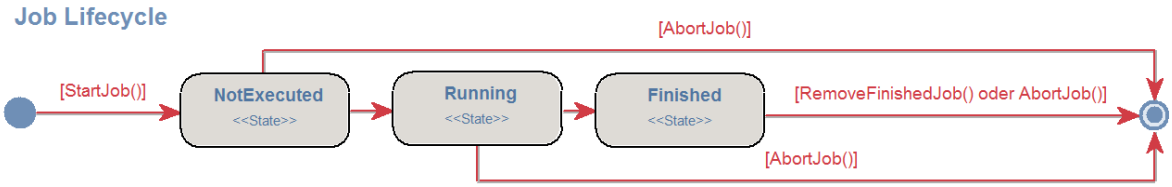


Abbildung 4: Protokolljob Lifecycle

5.1.3 Die Plugin-Schnittstelle

Zur Erstellung der Plugins erhält die Protokollengine eine Referenz auf ein Interface namens *IModuleFactory*. Dieses Interface definiert eine einzige Methode `CreateModuleSet()`, welche eine Liste mit Instanzen zurückliefert, die das `IProtocolModule`-Interface implementieren.

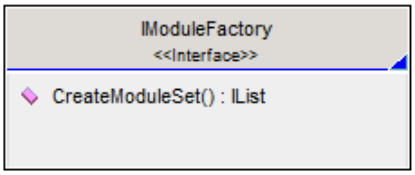


Abbildung 5: Module-Factory

Dieses Interface definiert verschiedene Eigenschaften, die das Modul beschreiben. Die Eigenschaft `ExecutionType` gibt an, unter welchen Bedingungen das Modul ausgeführt wird. Dabei wird unterschieden zwischen *Always, UserChoice* und *OnDemand*. *UserChoice* bedeutet, dass der Benutzer der Protokollengine für jeden Protokolljob festlegt, ob das Modul ausgeführt werden soll. Module mit dem `ExecutionType` *UserChoice* sind auch die einzigen Module, die außerhalb der Protokollengine, beispielsweise beim Aufrufen der Methode `GetAvailableModules()`, sichtbar sind. *Always* bedeutet, dass das Modul immer ausgeführt wird. Module mit dem `ExecutionType` *OnDemand* werden nur ausgeführt, wenn ein anderes Modul einen Wert anfordert, der von dem Modul bereitgestellt wird.

Dadurch, dass die Erzeugung der Plugin-Instanzen über ein Factory-Interface erfolgt, ist es möglich, auf einfache Weise eine neue Logik zum Laden der Plugins zu implementieren. Der Prototyp wird hierzu eine XamlModuleFactory realisieren, welche die Plugins anhand von XAML-Dateien erzeugt. XAML ist die Abkürzung für eXtended Application Markup Language. Es handelt sich um ein XML-basiertes Dateiformat zur Beschreibung von Objektbäumen. Durch die Verwendung dieses Formats ist es möglich, die Plugins auf einfache Weise zu konfigurieren, da das XAML-Format es erlaubt, beliebige Instanzen von Objekten inklusive deren Eigenschaftswerten zu definieren.

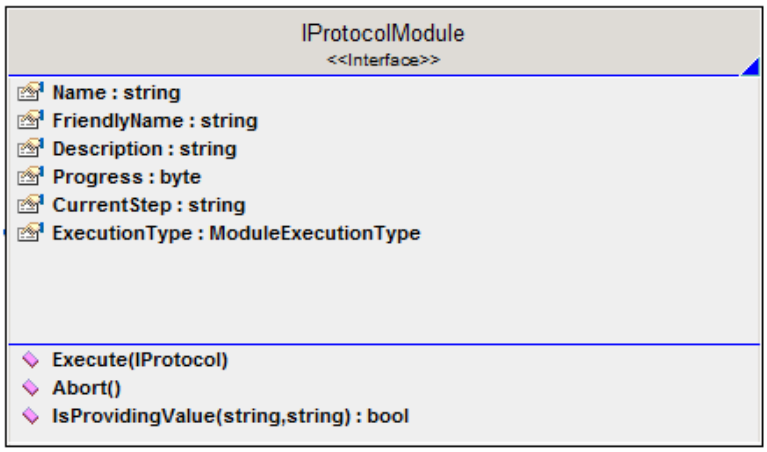


Abbildung 6: Plugin

5.1.4 Das Protokoll

Die `Protocol`-Klasse ist die zentrale Datenstruktur des Frameworks. Module erhalten zur Ausführung eine Referenz auf eine Instanz vom Interface-Typ `IProtocol` und können dann Werte in das Protokoll einfügen und Werte aus dem Protokoll abrufen. Durch das Verwenden eines Interfaces bleiben die Interna der Protokollimplementation dem Modul verborgen und Module können nur auf die Methoden zugreifen, die für den Zugriff der Module auf das Protokoll definiert wurden. Ein Modul kann beispielsweise nicht auf die Methode `ToXml()` der `Protocol`-Klasse zugreifen.

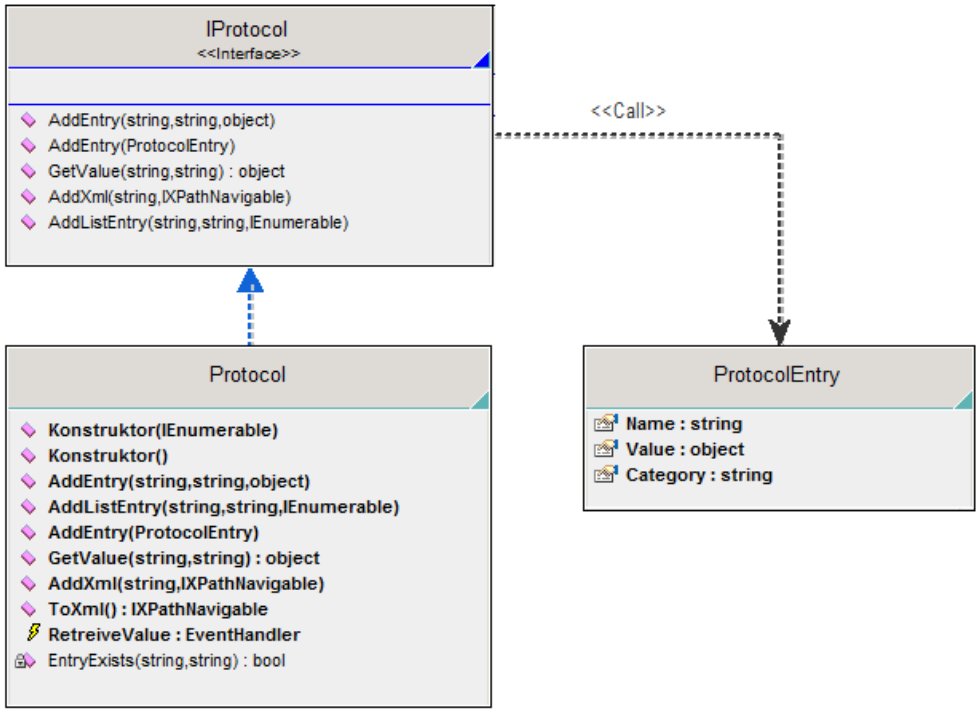


Abbildung 7: Klassendiagramm Protokoll

Ein Verbergen der internen Schnittstelle des Protokolls ist an dieser Stelle besonders wichtig, da die Module als Plugins geladen werden und beliebige neue Module von verschiedenen Entwicklern in das System integriert werden können. Weniger Methoden verbessern die Selbstdokumentation der Klasse, es wird verhindert, dass Methoden unsachgemäß verwendet werden und es kann bei Anpassungen an der Protokollengine davon ausgegangen werden, dass keine Module existieren, die auf die internen Methoden zugreifen.

5.1.5 Modul zu Modul Kommunikation

Die einzelnen Module fügen die ermittelten Informationen, zum Beispiel Testergebnisse, in das Protokoll ein. Dabei kann es vorkommen, dass ein Modul Informationen benötigt, die von einem anderen Modul beschafft werden müssen. Beispielsweise wird für den Prototyp ein Modul realisiert, welches den aktuellen Sourcecodestand eines Projektes aus der Versionskontrolle abruft und in einem bestimmten Pfad auf dem Testserver speichert. Ein weiteres Modul, welches eine Kompilierung des Sourcecodes durchführt, darf erst gestartet werden, wenn dieser Vorgang beendet ist und muss dann den Pfad, in dem der aktuelle Sourcecode liegt, aus dem Protokoll abrufen. Diese Modulabhängigkeiten werden von der Protokollengine berücksichtigt und bestimmen die Ausführungsreihenfolge der Module. Die Protokoll-Klasse spielt hierbei eine zentrale Rolle. Das Ereignis `RetrieveValue` wird ausgelöst, wenn ein Wert mit der Methode `GetValue()` aus dem Protokoll angefordert wurde, der nicht im Protokoll enthalten ist. In der Ereignisbehandlung wird nach einem Modul gesucht, welches diesen Wert bereitstellt und noch nicht vollständig ausgeführt wurde. Falls dies der Fall ist, wird das gefundene Modul gestartet bzw. auf die Beendigung des Moduls gewartet.

5.1.6 Die Testverwaltung

Die Testverwaltung setzt auf der Protokollengine auf und bietet zum einen eine testspezifische Sicht auf die Protokollerzeugung und zum anderen eine einfach zu bedienende Schnittstelle zum Zugriff auf das System und zur Auswertung des erzeugten Protokolls.

5.1.7 Der Test-Manager

Die `TestManager`-Klasse stellt den Zugriffspunkt auf die Testverwaltung dar. Eine Instanz der Klasse wird unter Angabe einer `IProtokollEngine` erzeugt, auf der die Testverwaltung arbeitet. Einzelne Protokolljobs werden von der Testverwaltung durch die Klasse `TestSeries` gekapselt. Die `TestManager`-Klasse bietet Methoden, um die aktuell laufenden Testreihen abzurufen und neue Testreihen zu erzeugen.

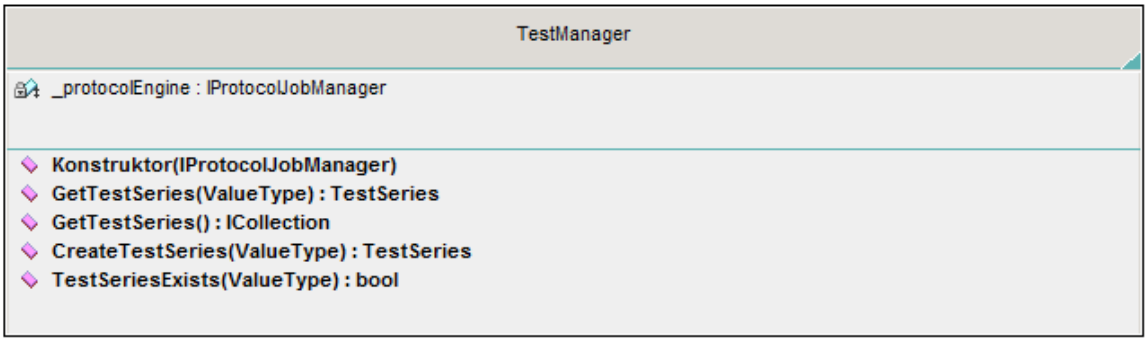


Abbildung 8: Klassendiagramm TestManager

5.1.8 Die Testreihen

Die einzelnen Testreihen werden durch die Klasse `TestSeries` repräsentiert. Eine Instanz dieser Klasse kann von Benutzern der Schnittstelle nur über die Methode `CreateTestSeries()` der `TestManager`-Klasse erzeugt werden. Dies wird durch einen Konstruktor sichergestellt, der als `internal` deklariert ist. Beim Erzeugen einer neuen Testreihe erzeugt der `TestManager` einen neuen Protokolljob auf der Protokollengine und erstellt eine `TestSeries`-Instanz mit der übergebenen Id und einer Referenz auf die Protokollengine. Die `TestSeries`-Instanz kann dann unter Angabe der eigenen Id die verschiedenen Methoden der Protokollengine aufrufen und so die angebotenen Methoden zum Starten und Abbrechen der Testreihe und zum Zugriff auf die Protokollierungsergebnisse realisieren.

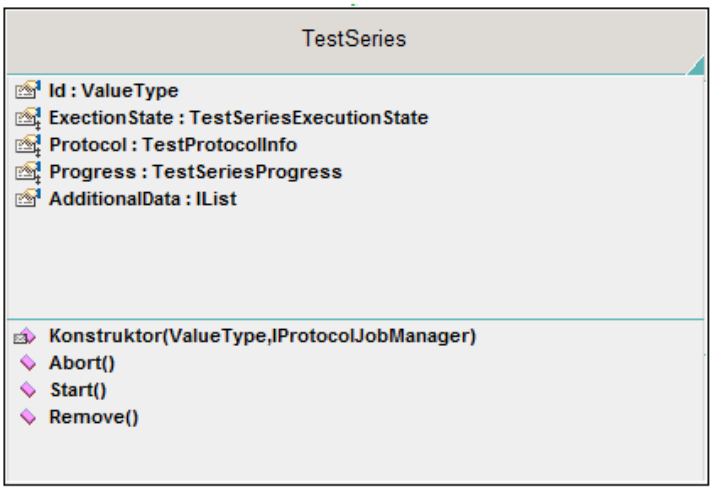


Abbildung 9: Klassendiagramm TestSeries

5.1.9 Protokollengine Client/Server

Wie weiter oben beschrieben wird die geplante Verteilung des Systems unter Verwendung der Windows Communication Foundation (WCF) realisiert. Prinzipiell wird zur Realisierung der Client/Server-Kommunikation ein Kommunikationsvertrag (Contract) erstellt. Dieser Kommunikationsvertrag wird durch ein .NET-Interface definiert, welches die Methoden spezifiziert, die von dem Server angeboten, beziehungsweise vom Client aufgerufen werden können. Das Interface muss auf beiden Seiten bekannt sein. Für das hier beschriebene System wird das `IProtocolJobManager`-Interface als Contract verwendet. Die Details der

Kommunikation, wie zum Beispiel das verwendete Transportprotokoll, werden von der WCF übernommen und sind vollständig konfigurierbar.

5.2 Anforderungen an zu integrierende Werkzeuge und Subsysteme

Prinzipiell kann jedes System und jedes automatisierbare Werkzeug als Plugin integriert werden. Die Plugin-Schnittstelle besteht in der Hauptsache aus einer `Execute`-Methode, welcher eine Protokoll-Instanz übergeben wird. Innerhalb der `Execute`-Methode können beliebige Applikationen oder Dienste angesprochen werden. Auch für die Ausgabe, welche in das Protokoll geschrieben wird, besteht im Prinzip keine Einschränkung, da sowohl Werte mit einfachen, primitiven Datentypen, als auch Listen, oder sogar, für komplexere Anwendungsfälle, komplette XML-Fragmente, in das Protokoll geschrieben werden können.

Davon abgesehen bestehen Unterschiede bezüglich des Integrationsaufwandes für verschiedene Testwerkzeuge:

Integrationsaufwand	Eigenschaften des Testwerkzeuges
Gering	• Aufruf über die Kommandozeile möglich • Beim Aufruf über die Kommandozeile wird keine grafische Benutzerschnittstelle angezeigt • Die Ausgabe des Werkzeuges erfolgt direkt auf der Standardausgabe oder in eine Datei
Mittel	• Aufruf über die Kommandozeile möglich • Es wird eine grafische Benutzeroberfläche angezeigt, aber es ist keine zusätzliche Benutzerinteraktion nötig • Die Ausgabe des Werkzeuges erfolgt direkt auf der Standardausgabe oder in eine Datei
	• • •
Hoch	• Kein Aufruf über Kommandozeile möglich • Es wird immer eine grafische Benutzeroberfläche angezeigt, die Eingaben vom Benutzer erwartet • Die Ausgabe des Werkzeuges erfolgt auf der Benutzeroberfläche

5.3 Testbarkeit / Teststrategie

5.3.1 Test-First

Die Architektur des Systems ist so spezifiziert, dass an vielen Stellen Interfaces verwendet werden. So erhält die Protokollengine beispielsweise einen Verweis auf Plugins vom Interface-Typ `IProtocolModule` und die Testverwaltung arbeitet nicht direkt mit einer Protokollengine-Instanz, sondern nur auf einem Interface vom Typ `IProtocolEngine`. Die einzelnen Module können durch *Stubs* [Beck02] substituiert werden, was den *Test-First Ansatz* [Beck02] begünstigt.

Die Erstellung des Prototyps soll nach dem Test-First Ansatz erfolgen. Entsprechend werden nach dem Definieren der einzelnen Klassen (und weiterer Typen) vor der Implementation die Tests für das System in Unit-Tests beschrieben. Erst wenn alle Anforderungen durch Unit-Tests abgedeckt sind, erfolgt die Implementation.

5.3.2 Integrationstest

Nachdem die einzelnen Systemmodule der durch die Unit-Tests festgelegten Spezifikation entsprechen, erfolgt ein Integrationstest, bei dem getestet wird, ob alle Systemmodule korrekt miteinander interagieren.

5.3.3 Regressionstests

Die vor der Implementation erstellten Unit-Tests können anschließend verwendet werden, um bei Änderungen Regressionstests durchzuführen. Ebenso werden für alle Fehler, die beim Integrationstest gefunden wurden, die Unit-Tests um entsprechende Testfälle ergänzt, um so das wiederholte Auftreten eines bereits behobenen Fehlers zu vermeiden.

5.4 Fallbeispiel: Integration eines Testwerkzeuges

Um die Funktionsweise des Systems zu verdeutlichen, soll an dieser Stelle exemplarisch die Integration eines neuen Werkzeuges in das System gezeigt werden.

In diesem Beispiel soll ein Unit-Test Framework integriert werden. Um das Beispiel einfach zu gestalten, soll es sich um ein Werkzeug mit geringem Integrationsaufwand handeln, welches komplett über die Kommandozeile gesteuert werden kann. Der Aufruf des Werkzeuges auf der Kommandozeile ist folgendermaßen aufgebaut:

```
UnitTest [Quellverzeichnis] [Ausgabedatei]
```

Das Werkzeug liest alle Unit-Test Binaries aus dem angegebenen Quellverzeichnis und schreibt die Testergebnisse in die angegebene Ausgabedatei.

Zur Integration dieses Werkzeuges sind folgende Schritte nötig:

1. Implementation des Plugins

Erster Schritt ist die Implementation des `IProtocolModule`-Interfaces. Hierbei spielt insbesondere die Methode `Execute` eine entscheidende Rolle, da dies die Methode zur Ausführung des Plugins ist.

```
public class UnitTestPlugin : IProtocolModule
{
    public void Execute(IProtocol protocol)
    {
        //Aufruf des Werkzeuges
        string commandLine =
            String.Format("UnitTest {0} {1}", _sourcePath, _outFile);
        CommandLine.Execute(commandLine);

        //Einlesen der Ausgabe
        TestResults results = TestResults.Read(_outFile);
        protocol.AddEntry("UnitTestResults", results);
    }
}
```

2. Hinzufügen des Plugins zur Anwendungskonfiguration

Die Plugin-Konfiguration erfolgt über XAML-Dateien, in welchen definiert ist, welche Plugins verwendet werden sollen und wie diese Plugins konfiguriert werden. Zum Hinzufügen eines Plugins muss eine neue Plugin-Beschreibungsdatei im Plugin-Verzeichnis der Anwendung erzeugt werden. Für das aktuelle Beispiel könnte diese Datei folgendermaßen aussehen:

```
<UnitTestPlugin
    xmlns="clr-namespace:Layer2.Testing;assembly=UnitTesting"
    Name="UnitTestModul"
    FriendlyName="Unit-Testing"
    Description="Dieses Plugin führt Unit-Tests durch"
/>
```

Der Hauptknoten der XAML-Datei gibt den Klassennamen des Plugins an. Über den XML-Namespace wird gleichzeitig der Ort des Plugins (Namespace der Klasse und Name der Assembly) angegeben. Zusätzlich können beliebige Eigenschaften gesetzt werden, in dem diese als weitere Attribute des Knotens angegeben werden.

Nach diesen zwei Schritten wurde das Werkzeug in das System integriert. Es wird anschließend bei jeder Testserie mit geladen.

5.5 Fallbeispiel: Konfigurationsszenario

Um das Zusammenspiel einzelner Plugins zu verdeutlichen und einen Überblick über die gesamte Funktionsweise zu geben, soll an dieser Stelle beispielhaft eine Plugin-Konfiguration dargestellt werden. Hierzu wird angenommen, dass folgende Aufgaben von dem System durchgeführt werden sollen:

- Abrufen des aktuellen Quellcodes aus der Versionskontrolle
- Ermittlung einiger Code-Metriken auf Basis des Quellcodes
- Kompilierung des Quellcodes
- Anwendung von Unit-Tests auf die Kompilate

Hierzu müssen folgende Plugins realisiert und in das System integriert werden:

- Plugin zum Zugriff auf die Versionskontrolle
- Plugin für die Ermittlung der Code-Metriken
- Plugin zur Kompilierung des Quellcodes
- Plugin zur Durchführung von Unit-Tests

5.5.1 Durchführung einer Testserie

1. Die Testserie wird gestartet. Dabei muss als Ausführungsparameter eine Angabe gemacht werden, mit der das Plugin für den Zugriff auf die Versionskontrolle das Projekt identifizieren kann. Zum Beispiel, je nach Versionskontrollsystem, eine Projekt-ID oder ein Dateisystem- bzw. Netzwerkpfad.
2. Das System erzeugt für die gestartete Testserie neue Instanzen der Plugins (eine Instanz des Versionskontroll-Plugins, eine Instanz des Code-Metrik-Plugin, usw.) und ruft sequenziell die `Execute`-Methode für jede Plugin-Instanz auf.
3. Alle Plugins, mit Ausnahme des Versionskontroll-Plugins, unterbrechen ihre Ausführung und warten auf die Beendigung des Quellcode-Updates. Anschließend werden sowohl das Plugin für die Code-Metriken, als auch das Plugin für die Quellcode-Kompilierung parallel ausgeführt. Das Unit-Test-Plugin setzt seine Ausführung erst fort, wenn die Kompilierung abgeschlossen ist.
4. Alle Plugins wurden ausgeführt und haben ihre Ergebnisse in das Protokoll geschrieben. Der Qualitätsreport wird zurückgegeben.

5.5.2 Implementation der Plugins

Typischerweise würden zur Realisierung der Funktionalität vorhandene Werkzeuge verwendet werden. Es müssen dann lediglich, wie in Kapitel 4.5 gezeigt, Wrapper-Plugins entwickelt werden, um diese Werkzeuge im Kontext des Testsystems zu verwenden.

5.5.3 Ablaufsteuerung

Die Ablaufsteuerung, das Sicherstellen der korrekten Ausführungsreihenfolge, wird nicht explizit konfiguriert. Das System ermittelt aus den Aufrufen, die ein Plugin auf das aktuelle Protokoll macht, welche Plugins beendet werden müssen, damit die Ausführung eines bestimmten Plugins fortgesetzt werden kann. Den Kern dieser Funktionalität stellt das Protokoll dar. Wenn Plugins Informationen aus dem Protokoll anfordern, wird zuerst festgestellt, ob die erforderliche Information bereits im Protokoll vorhanden ist. Falls dies nicht der Fall ist, wird für alle aktiven Plugins geprüft, ob sie den gesuchten Wert bereitstellen. Wenn ein solches Plugin gefunden wird, wird die Ausführung des abhängigen Plugins pausiert und erst wieder fortgesetzt, wenn das Plugin, welches die Information bereitstellt, seine Ausführung beendet hat.

Das Kompilierungs-Plugin könnte beispielsweise folgendermaßen implementiert sein:

```
public class BuildPlugin : IProtocolModule
{
    public void Execute(IProtocol protocol)
    {
        //Auslesen des Pfades mit den Quelldateien aus dem Protkoll
        string sourcePath =
            (string)protocol.GetValue("ProjectData", "SourceFilesPath");

        //Aufruf des Werkzeuges
        string commandLine =
            String.Format("Build {0} {1}", sourcePath, _outPath);
        CommandLine.Execute(commandLine);

        //Pfad zu den Kompilaten in das Protokoll schreiben
        protocol.AddEntry("ProjectData", "CompiledFilesPath", _outPath);
    }
}
```

In der ersten Codezeile wird aus dem Protokoll der Wert *SourceFilesPath* in der Kategorie *ProjectData* angefordert. Wenn das Versionskontroll-Plugin seine Ausführung noch nicht beendet hat, stellt das Protokoll fest, dass der gesuchte Wert nicht vorhanden ist und versucht, ein Plugin zu finden, welches diesen Wert bereitstellt. Es findet das Versionskontroll-Plugin und blockiert den aktuellen Thread (den Thread für das Kompilierungs-Plugin), bis das Versionskontroll-Plugin seine Ausführung beendet hat. Anschließend ist der Wert *SourceFilesPath* im Protokoll vorhanden und wird an das Kompilierungs-Plugin zurückgegeben. Dieses kompiliert die Quelldateien und schreibt den Pfad zu den Kompilaten in das Protokoll. Das Unit-Test-Plugin, welches auf diesen Wert wartet, würde erst dann ausgeführt werden, wenn das Kompilierungs-Plugin seine Ausführung beendet hat.

6 Der Prototyp

Um die Machbarkeit des vorgestellten Frameworks zu beweisen, wurde ein Prototyp erstellt, der eine Teilimplementation der Spezifikation darstellt und somit nur eine Teilmenge der festgelegten Anforderungen erfüllt.

6.1 Verwendete Werkzeuge und Technologien

Zur Erstellung des Prototyps wurden unterschiedliche Werkzeuge und Technologien verwendet:

- **Microsoft® VisualStudio™ -** Integrierte Entwicklungsumgebung zur Entwicklung von Programmen auf Basis des Microsoft® .NET-Frameworks.
- **Microsoft® Visual C# -** Der Prototyp wurde in der Programmiersprache Visual C# entwickelt.
- **microTOOL ObjectiF –** UML-Modellierungswerkzeug mit Integration in Microsoft® VisualStudio™. Unterstützt *reverse engeneering*.
- **JetBrains Resharper –** Visual Studio Add-In, welches die Visual Studio IDE um viele Features für visuelle Programmierunterstützung und Refactor-Funktionen erweitert.

6.2 Umfang des Prototyps

Der erstellte Prototyp realisiert folgende Module:

- **Protokollengine** – Da dieses Modul die Kernfunktionalität des Systems beinhaltet, wurde zur Realisierung des Prototyps die komplette Protokollengine gemäß der Spezifikation implementiert.
- **Testverwaltung** – Dieses Modul wurde ebenfalls gemäß der Spezifikation vollständig implementiert.
- **Verteilung des Systems (Client/Server)** – Da die Verteilung des Systems zum Aufzeigen der prinzipiellen Machbarkeit keine Rolle spielt, wurde auf die Implementation des Clients und des Servers für den Prototyp verzichtet.
- **Benutzeroberfläche** – Für den Prototyp wurde eine einfache, grafische Benutzeroberfläche realisiert.
- **Ergebnisverarbeitung** – Um die Ergebnisse aus der Protokollengine abzurufen und zu verarbeiten wurde eine Ergebnisverwaltung realisiert, die allerdings nur aktiv ist, wenn die Anwendung ausgeführt wird. Für den produktiven Einsatz sollte dieses Modul aber als unabhängiger Dienst permanent aktiv sein.

- **Plugins** – Für den Prototyp wurden folgende Plugins realisiert:
  - **Build-Plugin** – Ein Plugin zur Kompilierung von .NET-Projekten
  - **Subversion-Plugin** – Ein Plugin zum Abrufen der Quelldateien aus der Subversion-Versionskontrolle
  - **Unit-Test Plugin** – Ein Plugin zur Ausführung von Unit-Tests

Auf Basis der implementierten Funktionalität ist der Prototyp in der Lage, Unit-Tests auf dem aktuellen Quellcodestand auszuführen, indem

- die aktuellen Quelldateien über das Unit-Test Plugin abgerufen werden,
- die Quelldateien durch das Build-Plugin kompiliert werden,
- die Unit-Tests auf den aktuellen Kompilaten ausgeführt werden.

Damit kann der Prototyp bereits testen, ob

- der aktuelle Quellcode kompiliert werden kann,
- deim Kompilieren des Quellcodes Warnungen erzeugt werden,
- die aktuellen Kompilate die Unit-Tests bestehen.

6.3 Plattformunabhängigkeit

Um die Plattformunabhängigkeit des Prototyps sicherzustellen, muss ermittelt werden, ob alle verwendeten Klassen und Sprachfeatures auch von dem *Mono*-Framework (siehe Kapitel 4.3.4) unterstützt werden. Hierzu wird ein Analysewerkzeug mit dem Namen *Mono Migration Analyzer* (MoMA) angeboten. Dieses Werkzeug prüft, ob alle Assemblies (Kompilate) eines Systems von einer bestimmten Mono-Version unterstützt werden und erzeugt ein Analyseprotokoll, aus dem ersichtlich wird, welche Klassen oder Sprachfeatures nicht von dem *Mono*-Framework angeboten werden.

Eine Analyse des Prototyps mit dem MoMA-Werkzeug hat ergeben, dass, mit einer einzigen Ausnahme, der gesamte Prototyp auf Mono 2.2 ausgeführt werden kann. Die Ausnahme bildet die Klasse `XamlReader`, welche vom .NET-Framework angeboten wird, aber in Mono noch nicht implementiert wurde.

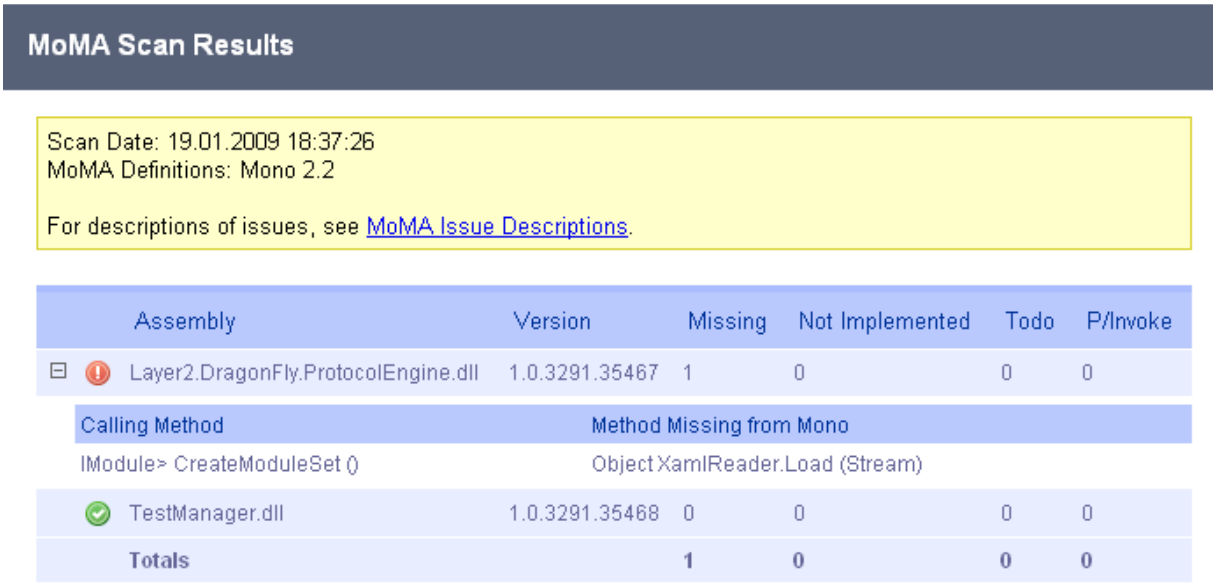


Abbildung 10: Ausgabe des MoMA-Werkzeuges

Die `XamlReader`-Klasse wird im Prototyp dazu verwendet, die Module-Factory auf Basis von XAML zu implementieren. Für eine Portierung des Systems auf Mono, und damit auf alternative Betriebssysteme, wie Linux oder Mac OS X, müsste also eine neue Module-Factory implementiert werden. Diese würde dann die Plugin-Erzeugung auf andere Weise, also ohne Verwendung des XAML-Formats, implementieren.

6.4 Erfüllung der Anforderungen

Der Prototyp erfüllt die folgenden Anforderungen (vergl. Kapitel 3.11 „Tabelle der Anforderungen“):

Nr	Titel	Beschreibung	Klassifizierung	Erfüllt
Grundsätzliche Anforderungen				
G001	Einfache Integration von Testwerkzeugen	Die Integration neuer Werkzeuge in das System soll mit minimalem Aufwand erfolgen können.	Must	Ja
G002	Geringe Anforderungen an Testwerkzeuge	Die Anforderungen an Testwerkzeuge, die verwendet werden können, sollen möglichst gering sein.	Must	Ja
G003	Erstellung eines zusammenfassenden Qualitätsreports	Das System soll einen Qualitätsreport erstellen, der die Ergebnisse aller Tests zusammenfasst.	Must	Ja
G004	Flexibles Format für den Qualitätsreport	Der Qualitätsreport soll in einem flexiblen, maschinenlesbaren Format erstellt werden.	Must	Ja
G005	Einfache Benutzerschnittstelle	Die Benutzerschnittstelle zum Starten und Verwalten von Testserien soll möglichst einfach gestaltet sein.	Must	Ja
Zusätzliche Anforderungen				
F001	Plugin-Schnittstelle	Das Hinzufügen neuer Testmodule soll erfolgen können, ohne dass das bestehende System in irgendeiner Weise verändert werden muss.	Should	Ja
F002	Ausführungsparameter	Es soll eine Möglichkeit geben, beim Starten eines Testlaufs beliebige Parameter an das System zu übergeben, welche von den Modulen als Ausführungsargumente verwendet werden können.	Should	Ja
F003	Nicht-Test Plugins	Es soll möglich sein, Plugins zu entwickeln, die keine Testwerkzeuge ausführen sondern beispielsweise Workflows über eine Workflow-Engine starten oder nur Informationen über die verwendete Testmaschine ermitteln.	Should	Ja
F004	Pluginabhängigkeiten	Es soll möglich sein, Abhängigkeiten zwischen Testmodulen zu berücksichtigen, so dass die Reihenfolge in der die Module ausgeführt werden, bestimmt werden kann.	Should	Ja
F005	Datenaustausch zwischen Plugins	Es soll möglich sein, dass Plugins Informationen an andere Plugins weitergeben, so dass ein Informationsaustausch zwischen den Plugins ermöglicht wird.	Should	Ja

F006	Nebenläufigkeit der Plugins	Die einzelnen Plugins eines Testlaufs sollen nicht sequenziell, sondern nebenläufig ausgeführt werden.	Should	Ja
F007	Nebenläufigkeit der Testläufe	Es soll möglich sein, mehrere Testläufe gleichzeitig durchzuführen.	Should	Ja
F008	Der Testserver	Es soll einen dedizierten Testserver geben, auf dem alle Testläufe durchgeführt werden.	Should	Nein
F009	Versionierung	Die Version der Plugins soll durch drei Versionsnummern beschrieben werden: • Die Versionsnummer des verwendeten Verkzeuges oder Subsystems. • Die Versionsnummer des Plugins • Die Versionsnummer der Testfälle / Testkonfiguration	Should	Ja
F010	Isolierung	Plugins, die voneinander unabhängig sind, dürfen sich nicht gegenseitig durch Fehler beeinträchtigen	Must	Ja
F011	Fehlerbehandlung	Fehler, die bei der Ausführung von Plugins aufgetreten sind, sollten dem Benutzer gemeldet werden.	Should	Ja

Wie aus der Tabelle ersichtlich wird, erfüllt der Prototyp bereits alle Anforderungen mit Ausnahme der Client/Server-Verteilung.

6.5 Tätigkeiten bis zur Produktreife

6.5.1 Verteilung

Wie bereits im Kapitel *Architektur* dargelegt, ist eine Verteilung des Systems aus mehreren Gründen sinnvoll. Für den produktiven Einsatz des Systems sollte dies noch realisiert werden.

6.5.2 Ergebnisverarbeitung

Die Verarbeitung der Ergebnisse erfolgt momentan nur, solange die Anwendung gestartet ist. Zur endgültigen Produktreife sollte ein Dienst realisiert werden, der permanent aktiv ist und fertige Protokolle aus dem System abruft und verarbeitet.

6.5.3 Weitere Plugins

Der Prototyp zeigt bereits, dass das System grundsätzlich machbar ist. Dennoch beschränkt sich dieser momentan im Prinzip auf Unit-Testing. Der große Mehrwert des Frameworks wird erst deutlich, wenn weitere Plugins für verschiedenste andere Tests in das System integriert werden.

7 Ausblick

Der Prototyp, welcher im Zusammenhang mit dieser Arbeit erstellt wurde, realisiert eine geringe Funktionalität, um die Machbarkeit und Funktionsweise des Systems zu demonstrieren. Darüber hinaus gibt es vielfältige Möglichkeiten das System anzupassen, zu erweitern oder auf andere Art und Weise zu verwenden.

7.1 Die Protokollengine

Die Protokollengine abstrahiert das gesamte System und ist nicht spezifisch für das Testen von Software implementiert. Es handelt sich prinzipiell um ein Plugin-System zum Sammeln von Informationen aus verschiedenen Quellen und bietet, für sich allein genommen, folgende Funktionalität:

- Ausführen beliebiger Subsysteme (in Plugins verpackt)
- Erstellung einer Zusammenfassung aller Ausgaben der Plugins
- Festlegen einer Reihenfolge für die Ausführung der Plugins
- Informationsaustausch zwischen Plugins in eine Richtung

Dieses Softwaremodul kann in vielen Szenarien eingesetzt werden. Immer wenn verschiedene Systeme / Werkzeuge mit einander integriert werden sollen und eine Zusammenfassung der Ausgaben aller Systeme / Werkzeuge gewünscht ist, kann dieses Modul eingesetzt werden. Somit wird durch die Abstraktion dieser Funktionalität ein grundsätzlicheres Problem gelöst.

Beispielsweise könnten für einen Netzwerkadministrator verschiedene Messwerkzeuge in das System integriert werden, welche Kennzahlen aus dem Netzwerk ermitteln und in dem Protokoll zusammenfassen, so das dieser Administrator regelmäßig einen Statusbericht über den Zustand seines Netzwerkes erhalten könnte.

7.2 Ideen für Test-Plugins

Um die Unterstützung der Qualitätssicherung bei der Erstellung von Software durch das System weiter zu erhöhen, könnten unterschiedlichste weitere Plugins realisiert werden.

7.2.1 Unit-Tests

Unit-Tests sind eine typische und verbreitete Technik für automatisiertes Testen und sollten unbedingt in das System integriert werden.

7.2.2 Softwaremetriken

Eine Softwaremetrik bildet eine Eigenschaft einer Software in einem Zahlenwert ab und bietet so die Möglichkeit bestimmte Aspekte von Software zu bewerten und zu vergleichen.

Beispiele für Softwaremetriken sind:

- **Lines of Code (LoC)** – Die Anzahl der Codezeilen ohne Berücksichtigung von Leerzeilen und Kommentaren.
- **Zyklomatische Komplexität** – Ein Wert, der die Komplexität des Kontrollflusses eines Programms angibt. [Balzert98 / S. 481]
- **Number of Classes (NoC)** – Die Anzahl der Klassen in einem objektorientierten Softwaresystem.

Zur Erstellung von Softwaremetriken stehen eine Fülle von Werkzeugen zur Verfügung. Diese könnten durch entsprechende Plugins verpackt werden und den Qualitätsreport um einige informative Kennzahlen bereichern.

7.2.3 Lasttests / Stresstests

Auch für Lasttests bzw. Stresstests stehen viele Werkzeuge zur Verfügung. Diese könnten ebenso in das System integriert werden und den Qualitätsreport ergänzen.

7.2.4 Quellcode-Richtlinien Test

Es gibt auf dem Markt Werkzeuge, die den Quellcode auf Einhaltung der vom Unternehmen verwendeten Code-Richtlinien testet. Beispielsweise die Erstellung von Sourcecodeheadern, Groß- und Kleinschreibung für verschiedene Codeteile (Variablen, Typen, Konstanten, u. a.). Eine Integration eines solchen Werkzeuges würde ebenfalls die Qualität der Software erhöhen, indem die Entwickler auf die Einhaltung der Richtlinien hingewiesen werden.

7.2.5 Quellcode-Qualitätstest

Es existieren Werkzeuge, welche die Qualität des Codes im Hinblick auf bestimmte Schwächen und häufig auftretende Programmierfehler oder einfach nur fehleranfälligen Code prüfen („Best Practices“-Test). Ein solches Werkzeug zu integrieren würde ebenfalls die Qualität der Software stark erhöhen.

7.2.6 Kompilierungstest

Die Kompilierung des Sourcecodes kann ebenfalls als Test betrachtet werden. Selbstverständlich muss der Sourcecode kompilierbar sein. Darüber hinaus sollten auch Warnungen des Kompilers nur in den seltensten Fällen ignoriert werden.

7.2.7 Informationen der Testmaschine

Bei der Analyse eines Softwarefehlers kann es oftmals von großem Vorteil sein, wenn Informationen über das für den Test verwendete System vorliegen. Hierzu wäre es von Vorteil, ein Plugin zu erstellen, welches lediglich Informationen über den Testserver ermittelt. Beispielsweise Hardwarefakten (CPU, Arbeitsspeicher, u.a.) und Betriebssystem-Informationen (OS-Version, OS-Konfiguration, u.a.) und den Qualitätsreport mit diesen Informationen anreichert.

7.2.8 Informationen aus der Versionskontrolle

Ebenfalls wäre es eine große Bereicherung des Qualitätsreports, wenn dieser durch Informationen aus der Versionskontrolle angereichert wird. Beispielsweise könnten einige aktuelle Changelog-Einträge der Entwickler hinzugefügt werden oder Versionsinformationen einzelner Sourcecodedateien. Auch wäre es denkbar, die Aktivität (Änderungsrate) für einige Sourcecodedateien auszugeben, um problematische Stellen im Design der Anwendung zu identifizieren. (Oft geänderte Sourcecodedateien weisen auf Schwächen im Design der Anwendung hin. Evtl. sollte z.B. eine Klasse in mehrere Klassen aufgeteilt werden o.ä.).

7.2.9 Workflow-Integration

Da durch ein Plugin im Prinzip jedes beliebige Subsystem in das Testsystem integriert werden kann, liegt auch der Gedanke nahe, komplexere Prozesse, die mit der Qualitätssicherung von Software zusammenhängen, in das System zu integrieren. Hierzu könnte ein Plugin eine Workflow-Engine verwenden und damit Unterstützung für unterschiedlichste QS-Maßnahmen realisieren, die sonst vollständig manuell ausgeführt werden würden.

7.2.10 Manuelles Testen

Auch die Durchführung manueller Tests könnte über das System verwaltet werden. Hierzu müsste ein System zur Verfügung stehen, welches Testfälle für manuelles Testen verwaltet und eine Workflow-Engine, welche von dem Plugin angesprochen wird und den manuellen Testlauf auslöst. Das Plugin würde dann beispielsweise bei seiner Ausführung eine E-Mail an den verantwortlichen Tester senden. Diese würde dann Anweisungen zum Einrichten oder Verwenden einer vorhandenen Testumgebung enthalten und ein Testfallprotokoll liefern, welches die Testfälle enthält, die vom Tester abgearbeitet werden müssen. Der Tester arbeitet dann das Testprotokoll ab und bestätigt den Abschluss seines Tests. Anschließend werden die Ergebnisse aus dem Testprotokoll für den manuellen Test von dem Plugin als Ausgabe in den Qualitätsreport übernommen.

7.2.11 Reviews

Das System könnte ebenfalls über ein Plugin eine Unterstützung für Code-Reviews erhalten. Bestimmte Teammitglieder würden dann z.B. per E-Mail informiert werden, dass ein Code-Review für bestimmte Quellcodes durchzuführen ist und könnten dann ein entsprechendes Formular ausfüllen, in dem Verbesserungsvorschläge und vorhandene Fehler eingetragen werden oder sogar eine Bewertung der einzelnen Sourcecodedateien auf einer festgelegten Skala vornehmen. Auch diese Informationen würden den erstellten Qualitätsreport stark bereichern und wertvoller machen.

7.3 Weboberfläche

Die Erstellung einer Weboberfläche zum Zugriff auf das Testsystem hätte einige Vorteile. Zum einen müsste auf den Entwicklungsmaschinen kein Client installiert werden und zum anderen könnte jeder Tester und Entwickler über das Internet die aktuellen Testläufe verwalten und Testergebnisse abrufen.

7.4 Ausfallsicherheit

Die Idee Plugins zu entwickeln, welche Workflows in einer WF-Engine starten (siehe auch 7.2.10 *Manuelles Testen*) führt zu der Frage, wie sich das System verhält, wenn die Maschine, auf der es installiert ist, abstürzt oder einfach nur neu gebootet werden muss. Denn Workflow-Aktivitäten mit Benutzerinteraktion können durchaus sehr langlebig sein und mitunter kann ein Testlauf so Tage oder Wochen aktiv sein. Mit der aktuellen Implementation des Prototyps würden alle aktiven Testläufe abgebrochen werden, sobald das System, aus welchem Grund auch immer, beendet würde. Schon bei geringer Auslastung des Systems würde dies zu massiven Problemen führen.

Dieses Problem könnte in einer zukünftigen Version behoben werden, indem die Plugins so erweitert werden, dass sie die Fähigkeit erhalten, ein serialisierbares Token zu liefern, welches den aktuellen, internen Status des Plugins repräsentiert (Memento-Pattern [Gamma96 / S. 354]). Die Protokollengine würde dann beim Beenden des Systems eine vollständige Serialisierung aller Status der Plugins vornehmen und diese auf der Festplatte speichern. Um eine Sicherheit bei Systemabstürzen zu gewährleisten, könnte die Protokollengine regelmäßige Serialisierungen der aktuellen Testläufe vornehmen, so dass nach einem Absturz zumindest wieder ein möglichst aktueller Stand zur Verfügung steht.

7.5 Auswertung der Qualitätsreports

Wenn während eines Softwareentwicklungsprojektes regelmäßig Qualitätsreports erstellt werden, entstehen eine Fülle von Qualitätsreports aus verschiedenen Phasen der Entwicklung. Natürlich kann durch Einsicht des aktuellen Qualitätsreport der aktuelle Qualitätsstatus ermittelt werden und ein solcher Report eignet sich auch, um für den Kunden zu einem Release der Software auch die durchgeführten QS-Maßnahmen zu dokumentieren. Dennoch könnten die Reports auch weiter aufbereitet werden, um für Projektleiter, Qualitätsbeauftragte oder Führungskräfte Informationen zu liefern, die den Verlauf des Projektes dokumentieren. So könnte der Verlauf verschiedener Qualitätsmerkmale im Laufe der Zeit in einem Diagramm dargestellt werden. Es lässt sich evtl. ermitteln, in welchen Projektphasen viel Code produziert wurde und in welchen Phasen die beste Codequalität vorhanden war. Für verschiedenste Ereignisse (Einführung neuer Werkzeuge, Änderung der Teamzusammensetzung u.a.) könnte ermittelt werden, ob diese evtl. in einer Weise die Qualität der Software beeinflusst haben.

Es wäre auch möglich, einen Wert zu definieren, der aus den Ergebnissen aller durchgeführten Tests errechnet wird und einen einfachen Überblick über den Qualitätsverlauf eines Projektes bietet. Dieser Wert könnte sich beispielsweise zusammensetzen aus dem Verhältnis der bestandenen zu den nicht bestandenen Tests.

7.6 Qualitätssicherungsnachweis für den Kunden

Der Qualitätsreport, welcher von dem System erstellt wird, kann auch als Nachweis der durchgeführten Qualitätssicherungsmaßnahmen gegenüber dem Kunden eingesetzt werden. Hierzu müsste der Report in eine geeignete Form konvertiert werden. Technische Details, die

für einen Entwickler oder Tester von Bedeutung sind, aber für den Kunden keine Aussagekraft besitzen müssten ausgenommen werden.

7.7 Automatische Generierung von Entwickleraufgaben

Der Nutzen des Systems könnte evtl. erhöht werden, in dem bei der Auswertung des Qualitätsreports automatisch Aufgaben für Entwickler erzeugt werden. Dies setzt voraus, dass innerhalb des Unternehmens eine Aufgabenverwaltung existiert, was aber in den meisten Betrieben der Fall sein dürfte. Das System könnte dann für bestimmte Fehler direkt eine Aufgabe für den verantwortlichen Entwickler mit allen zugehörigen Informationen aus dem Qualitätsreport anlegen.

8 Fazit

Ein System, dessen Nutzung nur minimalen Arbeitsaufwand für Tester und Entwickler bedeutet, hat eine hohe Chance von den Benutzern angenommen und in den Arbeitsalltag integriert zu werden. Ein Tester muss sich nicht darum kümmern, wie die von Entwicklern definierten Unit-Tests durchgeführt werden. Der Entwickler braucht nicht zu wissen, wie der vom Qualitätsbeauftragten festgelegte Lasttest für das System konfiguriert und ausgeführt wird. Ein Projektmanager braucht sich nicht damit zu befassen, wie die Kennzahlen für ein Softwareprojekt kalkuliert werden. Alle diese Benutzer können durch Betätigung einer einzigen Schaltfläche Tests durchführen, die von den entsprechend qualifizierten Teammitgliedern definiert wurden. Hierdurch entsteht eine sehr niedrige Hürde bei der Einführung und Akzeptanz neuer Testwerkzeuge und -methoden.

Der Prototyp hat gezeigt, dass die Idee prinzipiell umsetzbar ist, allerdings wird der Wert des Frameworks erst deutlich, wenn weitere Plugins für das System bereitgestellt werden.

Anhang A: Quellenverzeichnis

[Wallmüller95] Ernest Wallmüller, Ganzheitliches Qualitätsmanagement in der Informationsverarbeitung, Hanser, 1995, ISBN: 3-446-17101-0

[Boehm81] Barry W. Boehm, Software Engeneering Economics, Prentince-Hall, 1981, ISBN: 0-13-822122-7

[Balzert98] Helmut Balzert, Lehrbuch der Software-Technik,Spektrum, 1998, ISBN: 3-8274-0065-1

[Petrasch98] Roland Petrasch, Einführung in das Software-Qualitätsmanagement, Logos Verlag Berlin, 1998, ISBN: 3-89722-056-3

[GeigerKotte08] Walter Geiger, Willi Kotte, Handbuch Qualität, Vieweg Verlag, 2008, ISBN: 978-3-8348-0273-6

[Voigt97] Hans-Dietrich Voigt, Qualitätssicherung – Qualitätsmanagement, Verlag Handwerk und Technik GmbH, 1997, HT: 3.582.02421.0

[ISO8402] International Standard ISO 8402

[DIN555350/11] Deutsche Industrie Norm 555350, Teil 11

[ISO9126] International Standard ISO 9126

[Li04] Kanglin Li, Effective Software Test Automation, SYBEX Inc., 2004, ISBN: 0-7821-4320-2

[Kruchten99] Philippe Kruchten, Der rational unified process, Pearson Education Deutschland, ISBN: 3827315433

[Beck02] Kent Beck, Test Driven Development, Addison-Wesley, 2002, ISBN: 0321146530

[Gamma96] Erich Gamma, Entwurfsmuster, Addison Wesley, 1996, ISBN: 0-201-63361-2

Anhang B: Abkürzungsverzeichnis

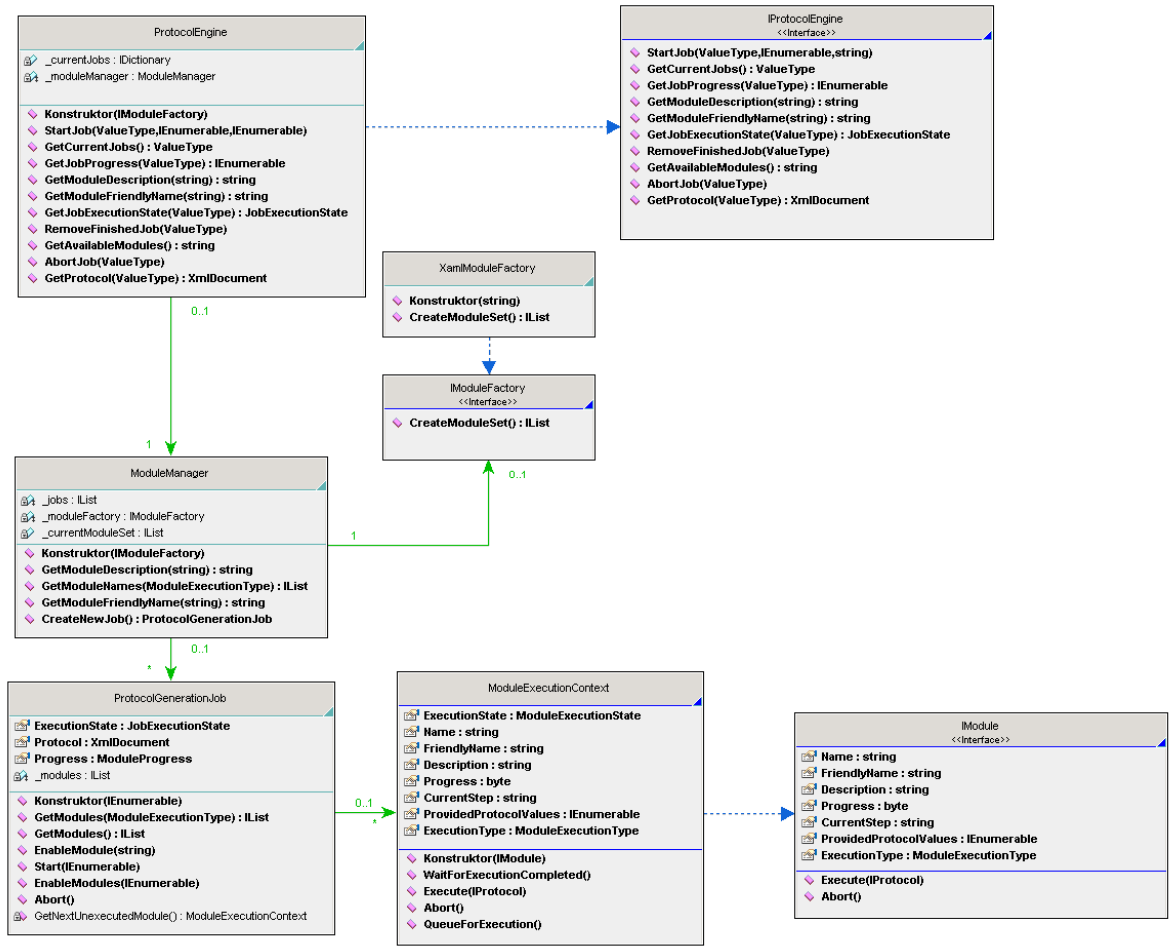
z.b.	Zum Beispiel
etc.	Et cetera
o.ä.	Oder ähnliche(s)
vs.	Versus
bzw.	Beziehungsweise
u.a.	Und andere(s)
evtl.	Eventuell
Inc.	Incorporated
QM	Qualitätsmanagement
QS	Qualitätssicherung
CPU	Central Processing Unit
XAML	Extended Application Markup Language
OS	Operating System
GUI	Graphical User Interface
IDE	Integrated Development Environment
HTML	Hypertext Markup Language
SOAP	Simple Object Access Protocol
HTTP	Hypertext Transfer Protocol
WCF	Windows Communication Foundation
TCP/IP	Transmission Control Protocol / Internet Protocol
XML	Extensible Markup Language

Anhang C: Abbildungsverzeichnis

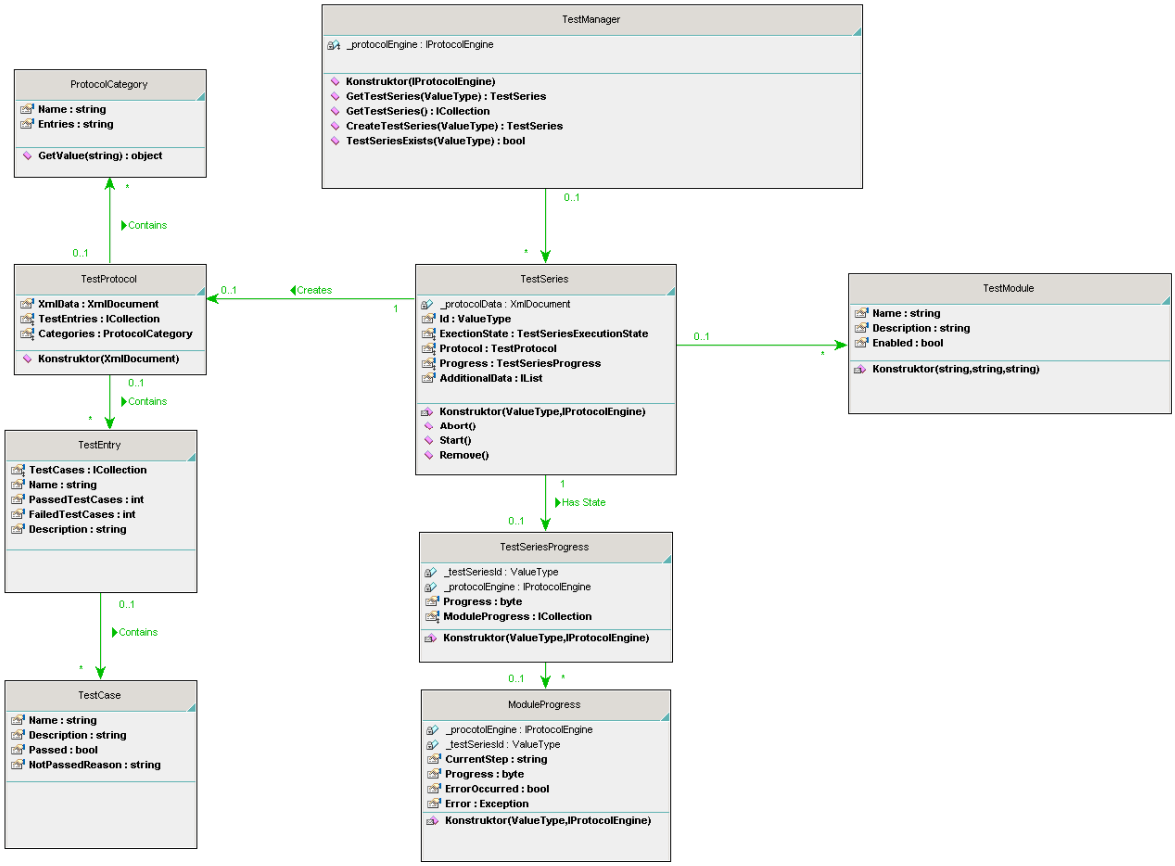
Anhang D: Stichwörterverzeichnis

Anhang E: UML-Diagramme

E.1 Protokollengine



E.2 Testverwaltung



Erklärung über selbstständige Erstellung der Arbeit

Hiermit erkläre ich, dass die vorliegende Arbeit im Sinne des §16, Absatz 5 der allgemeinen Prüfungs- und Studienordnung für Bachelor und Masterstudiengänge an der Fakultät Technik und Informatik, ohne fremde Hilfe selbständig Erstellt wurde und nur die angegebenen Quellen und Hilfsmittel verwendet wurden. Wörtlich oder dem Sinn nach aus anderen Werken übernommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____ _____

Ort, Datum Unterschrift