software quality lab
improve your quality

Quality Knowledge Letter

www.software-quality-lab.at

Ausgabe 2009 / 3
Erscheinungsart: ca. 4 x jährlich in elektronischer Form

Black-Box-Testautomatisierung für Embedded-Systems

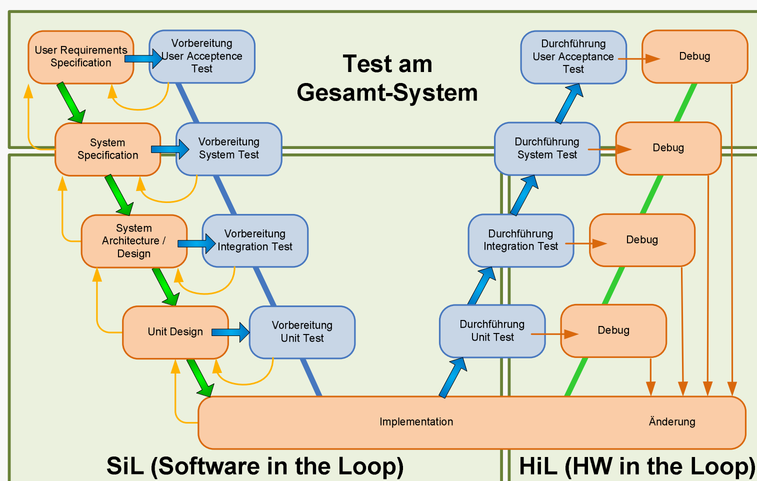
In dieser Ausgabe:

⇨ Begriffe: SiL (Software in the Loop) und HiL (Hardware in the Loop)
⇨ Black-Box-Testautomatisierung für eingebettete Systeme
⇨ Exkurs - NUnit
⇨ Literatur und Links

SiL (Software in the Loop) und HiL (Hardware in the Loop)

➲ SiL (Software in the Loop)

Bei der Methode Software in the Loop (SiL) wird ein erstelltes Modell der Software in einen für die zu testende Zielhardware verständlichen Code umgewandelt. Der Code wird auf dem Entwicklungsrechner zusammen mit dem simulierten Modell ausgeführt, anstatt wie bei HiL auf der Zielhardware zu laufen. Es handelt sich dabei also um eine Methode, die *vor* dem HiL anzuwenden ist. Vorteile von SiL sind unter anderem, dass die Zielhardware noch nicht feststehen muss, und dass die Kosten aufgrund der fehlenden Simulationsumgebung weitaus geringer ausfallen. Das Modell kann ev. auch beim HiL weiter verwendet werden, und somit sind die einzelnen Testläufe miteinander vergleichbar.



Schematische Anwendungsbereiche von HiL und SiL im W-Modell

➲ HiL (Hardware in the Loop)

Dabei wird das zu steuernde Gesamtsystem (z. B. Auto) über Modelle simuliert, um die korrekte Funktion des zu entwickelnden Teilsystems (z. B. Motorsteuergerät) zu testen. Die Eingänge des Steuergeräts werden mit Daten aus dem Modell stimuliert. Um die Test-/Reglerschleife (Loop) zu schließen, wird die Reaktion der Ausgänge des Steuergeräts, z. B. das Ansteuern eines Elektromotors, in das Modell zurückgeführt.

Die HIL-Simulation muss meist in Echtzeit ablaufen und wird in der Entwicklung und Test benutzt, um Entwicklungszeiten zu verkürzen und Kosten zu sparen. Insbesondere lassen sich wiederkehrende Abläufe simulieren und automatisieren (für Regressionstests).

Die Tests an realen (oft recht teuren) Systemen lassen sich dadurch stark verringern und zusätzlich lassen sich Systemgrenzen testen, ohne das Zielsystem (z. B. Auto und Fahrer) zu gefährden.

Embedded-System-Tests sind anders?

Tests im Rahmen von Embedded-Systemen sind in der Regel komplexer in der Durchführung wie Tests von reinen Software-Systemen.

Da die Test- und auch die Ziel-Systeme sehr oft teurer sind als bei reinen Software-Lösungen, muss dies bei der Architektur der Testautomatisierung auch entsprechend berücksichtigt werden (hier sind auch Themen wie HiL und SiL relevant).

Außerdem sind die Produktlebenszyklen von Embedded-Systeme meist auch langlebiger und in der Entwicklung stabiler als z.B. Web-SW-Lösungen. Damit rechnen sich in diesem Bereich auch aufwändige Testautomatisierungslösungen deutlich schneller.

Im Bereich der Testspezifikation und des Testprozesses sind die anzuwendenden Methoden und Vorgehensweisen in den meisten Fällen jedoch sehr ähnlich, wobei auf diese Themen im Rahmen des Newsletters nicht eingegangen wird.

Dieser Newsletter gibt einen grundlegenden Einblick in das Testen von Embedded-Systemen und stellt konkret anhand eines Projekt-Beispiels für Embedded-Systeme sowie eines kurzen Exkurses über NUnit dar, wie ein Testframework umgesetzt werden kann.

Dipl.-Ing. Johannes Bergsmann
Staatl. befugter und beeideter Ingenieurkonsulent für Informatik

Der Quality-Knowledgeletter ist ein periodisches Informationsmedium von Software Quality Lab und dessen Partnern mit den Schwerpunkten IT-Qualitätsmanagement, Projekt- und Prozess-Management.

Inhalt: fachliche Beiträge und Schwerpunktthemen, Vorstellung neuer Produkte und Leistungen, neue wissenschaftliche Erkenntnisse und andere Fachbeiträge aus unseren Themenbereichen.

Aktuelle Fach- und Forschungsbeiträge sind willkommen. Einsendungen an info@software-quality-lab.at .

Weitere Infos zu diesem und anderen Themen finden Sie auf http://www.software-quality-lab.at .

Black-Box-Testautomatisierung für eingebettete Systeme

von Bernhard Groiß, MSc, Software Test Consultant bei Software Quality Lab

Eingebettete Systeme wie Mobiltelefone, Geräte der Medizintechnik oder Unterhaltungselektronik spielen eine entscheidende Rollen in unserem Leben und prägen den Alltag. Es handelt sich hierbei meist um sehr spezifische Produkte, die sowohl von der Hardwarearchitektur als auch vom Softwaredesign speziell an ihre Aufgaben angepasst werden.

Aufgrund finanzieller Vorgaben werden oft kostengünstige Hardware-Software Implementierungen gewählt, welche die Leistungsfähigkeit von Hardware mit der großen Flexibilität aber auch Fehleranfälligkeit von Software vereinen.

Dieser Newsletter gibt einen generellen Einblick in das Testen von eingebetteter Software und zeigt anhand eines Beispiels, wie Tests für solche Systeme automatisiert werden können.

Embedded Systems Testing — Eine Einführung

Beim Testen von Software für eingebettete Systeme müssen im Vorfeld unterschiedliche Aspekte betrachtet werden.

Eine Schwierigkeit liegt darin, dass sich schon bei der Softwareentwicklung für diese Systeme gravierende Unterschiede zur klassischen PC-Softwareentwicklung (als synonym für die nicht embedded SW-Entwicklung) zeigen.

Bei der Applikationssoftwareentwicklung am PC wird meist auf Standardhardware mit standardisierten Schnittstellen und gängigen Betriebssystemen entwickelt.

Bei der Entwicklung von Software für eingebettete Systeme kommen oft spezielle Hardwarearchitekturen zum Einsatz, die für konkrete Aufgaben (z.B. Fahrzeugindustrie, Automatisierungstechnik, Funkübertragung) konzipiert wurden. Die Entwicklung der Software passiert hierbei oft auf der sogenannten Hostplattform, da auf dem Zielsystem (Target) meist nicht direkt entwickelt werden kann.

Nach Fertigstellung der Software kann diese dann auf die entsprechende Zielplattform portiert werden. Zusätzlich erfolgt die Entwicklung meist in Assembler, C oder anderen von der jeweiligen Hardware unterstützen Programmiersprachen und beinhaltet meist viele hardwarenahen Programmcode.

Weiters übernehmen eingebettete Systeme oft hochkritische Aufgaben, wie zum Beispiel in der Aeronautik oder der Medizintechnik, wodurch sich einer immer größer werdenden Anzahl an Standards entsprechen müssen.

Diese Eigenschaften und Anforderungen ändern und erweitern daher die Voraussetzungen an die immer mehr an Bedeutung gewinnende Qualitätssicherung solcher Systeme.

Je komplexer die Aufgaben der eingebetteten Software sind, desto schwieriger und zeitaufwendiger wird die Implementierung des Codes. Aufgrund der Komplexität und steigenden Anzahl der *Lines of Code* steigt auch die Fehleranfälligkeit der Software, was für sicherheitskritische Systeme oft auch unmittelbare Auswirkungen auf den Menschen und dessen Gesundheit haben kann.

Aus den genannten Gründen erhöht sich für solche Systeme demnach auch der Testaufwand.

Abbildung 1. zeigt schematisch die Kosten, um einen Fehler in der Software über die Projektlaufzeitzu beheben.

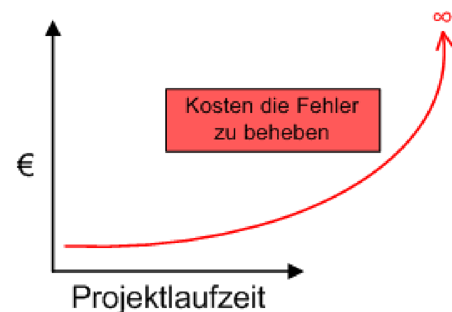


Abb. 1 — Kosten einen Fehler zu beheben als Funktion der Zeit im Produktlebenszyklus , Quelle: Ganssle

Um einen hohen Testabdeckungsgrad über die einzelnen Softwarekomponenten zu bekommen, ist eine systematische Qualitätssicherung während des gesamten Entwicklungsprozesses wichtig.

Die beginnt mit Unit Tests, die meist durch die Entwickler selbst erstellt und durchgeführt werden, über Systemtests bis hin zu Abnahmetests mit den Kunden. Gerade im embedded Systems-Umfeld ist oft die Qualität eines Produkts der ausschlaggebende Parameter für den Kauf, da Fehler in den embedded-Systemen oft auch mit teuren Rückholaktionen verbunden sein können.

Wie auch beim Test von PC-basierter Applikationssoftware wird im Bereich Embedded Systems Testing zwischen den folgenden Verfahren unterschieden:

(Fortsetzung auf Seite 3)

(Fortsetzung von Seite 2)

- Whitebox Testing
- Blackbox Testing
- Greybox Testing

Whitebox Tests werden in der Regel als Unit-Tests von den Entwicklern der Software implementiert und möglichst parallel zur Entwicklung ausgeführt und analysiert.

Greybox Tests vereinen die Vorteile von Black– und Whitebox Tests und werden meist im Vorfeld der eigentlichen Softwareentwicklung realisiert.

Bei Blackbox Tests besitzt der Tester keine Kenntnis über die Interna (Programmcode, innere Architektur) der zu testenden Software, verfügt jedoch im Idealfall sehr wohl über ausgeprägte Kenntnis über die Anforderungen an die Software.

In diesem Artikel wird primär auf die Möglichkeiten eingegangen, die Blackbox Tests bieten und unter welchen Voraussetzungen diese eingesetzt werden können.

Die wichtigsten Aspekte, die im Zusammenhang mit Tests für eingebettete Software beachtet werden müssen, sind in Tabelle 1 dargestellt:

1.	Embedded Software muss oft hoch verfügbar sein — Ausfälle dürfen lediglich sehr kurze Zeitspannen betreffen
2.	Embedded Software wird oft in Anwendungen eingesetzt, wo Menschenleben auf dem Spiel stehen
3.	Embedded Software ist oft sehr kostspielig in der Entwicklung
4.	Embedded Software muss oft Probleme der Hardware kompensieren (Zeitverhalten)
5.	Simulation von Embedded Systemen ist oft sehr schwierig bzw. kostspielig
6.	Anforderungen an das Zeitverhalten und Abhängigkeiten von physikalischen Umgebungsbedingungen (Sensordaten, HW-Daten)
7.	Embedded Software wird oft in Form von vernetzten und verteilten Systemen (z.B. CAN-Bus, LIN-Bus, etc.) implementiert
8.	Zeitgesteuerte vs. ergebnisgesteuerte Realisierungen von Embedded Software

Tabelle 1 — Eigenschaften von eingebetteter Software

Aufgrund dieser Unterschiede zwischen dem Testen von eingebetteter Software und der Applikationssoftware am PC unterscheiden sich die Tests unter anderem in folgenden Bereichen:

- Viel Testaufwand fließt normalerweise in den Test des Echtzeitverhaltens und der Nebenläufigkeit
- Testen von hoher Verfügbarkeit spielt eine wesentlichere Rolle als bei Applikationssoftware
- Mehr Testaufwand in Richtung Performanz– und Kapazitätstesten nötig
- Metriken zur Codeabdeckung spielen eine wesentlichere Rolle (vor allem für die Erfüllung von spezifischen Standards)
- Oftmals großer Aufwand im Bereich der (proprietären) Schnittstellen

Der folgende Abschnitt beschreibt wie automatisiertes Blackbox Testing von Embedded Systems konzeptionell durchgeführt werden kann.

Black-Box-Testing von eingebetteten Systemen

Aufgrund der bereits erwähnten Tatsache, dass eingebettete Systeme oft sicherheitskritische Aufgaben übernehmen, müssen entscheidende Teile im Code einer Verifikation unterzogen werden, ob diese korrekt funktionieren.

Das manuelle Durchführen von Tests ist zwar in der Fehlerfindung sehr effektiv, erweist sich jedoch langfristig gesehen als sehr kosten– und zeitintensiv. Zusätzlich stellt sich die Auswertung von manuellen Testfällen als schwieriger dar.

Automatisierte Modultests sollten jedoch nicht die einzigen Tests bleiben — es macht keinen Sinn, wenn eine spezielle Funktion zur Berechnung einer Prüfsumme korrekt funktioniert, jedoch beispielsweise die gesamte Software unter speziellen Rahmenbedingungen falsch arbeitet.

Es lohnt sich daher bei eingebetteter Software, im Laufe der Softwareentwicklung bzw. nach der Fertigstellung, Systemtests durchzuführen.

Meist wird für die System-Tests lediglich eine kompilierte und auf der Zielhardware ausführbare Version der Software benötigt.

Ab diesem Zeitpunkt stellt sich für Projektmitarbeiter und Testverantwortliche oft die Frage:

„Macht es für unser Unternehmen bzw. unsere Abteilung Sinn ein teures Werkzeug für das automatisierte Testen anzuschaffen oder gibt es eine Möglichkeit mit geringem Aufwand und Open-Source Werkzeugen eine solche Testumgebung aufzusetzen?“

(Fortsetzung auf Seite 4)

(Fortsetzung von Seite 3)

Die Antwort auf diese Frage kann nicht pauschal gegeben werden, da die Praxis gerade im Embedded Systems Engineering zeigt, wie unterschiedlich Projekte sein können. Diese Frage wird in diesem Newsletter jedoch nicht behandelt.

In der folgenden Abbildung (Abb. 2) werden die wichtigsten Aspekte dargestellt, die im Vorfeld einer Entwicklung eines sehr allgemein gehaltenen Black-Box-Testsystems diskutiert werden müssen.
Grundsätzlich kann es sich bei einem solchen Testsystems um eine Desktop-Anwendung unter allen gängigen Betriebssystemen handeln. Es muss zusätzlich vorausgesetzt werden, dass die Hardware des eingebetteten Systems korrekt funktioniert.
Die folgende Aufzählung beschreibt, welche Aufgaben die einzelnen Module im gesamten Testsystem übernehmen können.

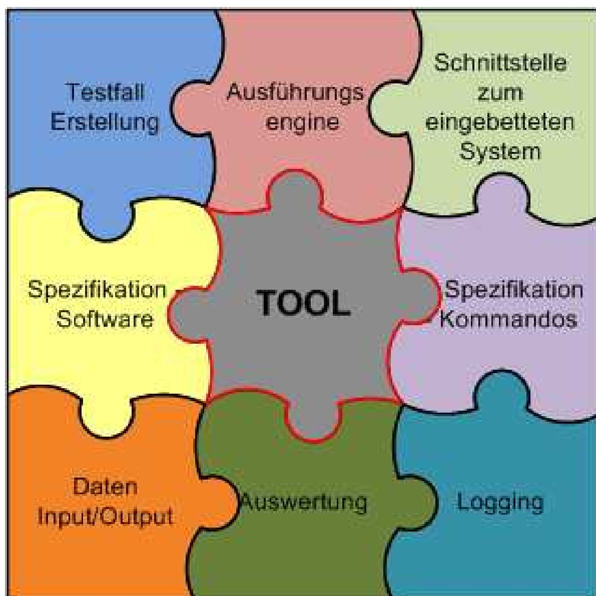


Abb. 2 — Wichtige Aspekte bei der Implementierung eines Black-Box-Testautomatisierungstools Quelle: Groiß

Testfall-Erstellung

Ein Testsystem zur Implementierung sowie Durchführung von Black-Box-Tests für Software in eingebetteten Systemen muss eine Möglichkeit bieten, Testfälle zu beschreiben bzw. zu erstellen.
Hierfür gibt es unterschiedliche Möglichkeiten, wie ein solcher Testfall aussehen kann. Denkbar sind etwa Testfälle, die in Form von XML-Dateien beschrieben werden, als Skripte (Python, VBA) oder in kompilierter Form als Funktionen bzw. Methoden einer .exe oder .dll Datei implementiert werden.

Ausführungsengine (Testtreiber)

Nach der Erstellung und Implementierung der Testfälle bedarf es einer Engine welche für die Ausführung bzw. den Aufruf des Testfalls verantwortlich ist.
Für den Fall, dass der Testfall in XML codiert ist, kann diese Ausführungseinheit auch in Form eines XML-Parsers implementiert sein, der die Umsetzung in eine Skript-Datei oder in eine .exe-Datei übernimmt.
Als Ausführungsengines gibt es Lösungen wie zum Beispiel MSTest oder NUnit, welche sich in eine eigens implementierte Umgebung integrieren lassen.
Mehr zu diesem Thema findet sich im Exkurs NUnit.

Schnittstelle zum eingebetteten System

Eines der wichtigsten Module in einem automatisierten Testsystem zur Durchführung von Black-Box-Tests ist die Schnittstelle zur Hardware bzw. zum eingebetteten Gerät (vgl. Abb.3).

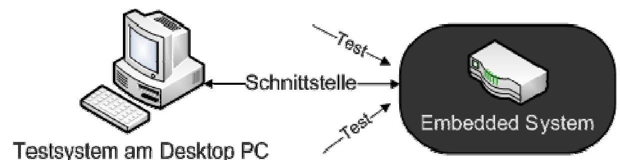


Abb. 3 — Schnittstellen , Quelle: Groiß

Diese Schnittstelle ist dafür verantwortlich einerseits die Kommandos bzw. Daten des Testfalls entgegenzunehmen und andererseits über eine Kommunikationsschnittstelle (seriell, USB, drahtlos) die entsprechenden Kommandos an die Software des eingebetteten Geräts zu übermitteln.
In der Gegenrichtung interpretiert diese Schnittstelle die Kommandos und Daten welche die Software am eingebetteten Gerät generiert und leitet diese an die Ausführungsengine bzw. das Auswertungsmodul weiter.
Diese Schnittstelle benötigt klarerweise eine *Gegenstelle* auf der Seite der Software am eingebetteten Gerät welche die Kommandos interpretiert und ausführt, Ergebnisse generiert und diese im Anschluss daran zurück an die Schnittstelle auf der PC-Seite sendet.
Für die Implementierung einer solchen Schnittstelle gibt es unterschiedliche Möglichkeiten wie zum Beispiel die Implementierung in Form eines COM-Servers in Form einer .exe oder .dll Datei, welcher die Kommandos an die eingebettete Software weiterleitet und auf der anderen Seite die Kommandos und Resultate entgegennimmt.
Es ist auch vorstellbar, diese Schnittstellensoftware in Form eines Kommandoparsers zu implementieren, dessen Funktionen zum Senden / Empfangen von der Ausführungsengine aufgerufen werden können.

(Fortsetzung auf Seite 5)

(Fortsetzung von Seite 4)

Wichtig ist jedoch, dass die Verbindung zum eingebetteten Gerät erfolgreich initialisiert wurde.

- **Spezifikation (Software)**
  Da die Systemtests für Firmware bzw. eingebettete Software meist von entwicklungsfremden Personen implementiert und durchgeführt werden, ist eine detaillierte Beschreibung der Anforderungen bzw. der funktionalen Eigenschaften der Software unabdingbar.
  Mit Hilfe einer Spezifikation der Software (Verhaltensbeschreibung, Zustandsautomaten, etc.) ist der Tester in der Lage, sinnvolle Testfälle zu erstellen, alle Anforderungen mit Hilfe von Testfällen abzudecken und für zeitliche Engpässe eine Priorisierung der Testfälle vorzunehmen.

- **Spezifikation (Kommandos)**
  Für die Implementierung eines Protokoll– bzw. Kommandoparsers ist eine Spezifikation jener Kommandos nötig, welche die eingebettete Software interpretieren bzw. senden kann.

- **Daten (Input/Output)**
  Bei der Implementierung von datengetriebenen Tests, welche aus Gründen der effizienteren Fehlerfindung zu präferieren sind, benötigt der Tester eine Übersicht über gültige/ungültige Eingabe– bzw. Ausgabedaten, welche die eingebettete Software verarbeiten bzw. generieren kann. Diese Daten können auch in geordneter Form (XML, CSV, Excel) als Input für entsprechend implementierte Testfälle dienen.

- **Auswertung**
  Einer der wichtigsten Punkte bei der Testautomatisierung ist eine übersichtliche Darstellung der Testergebnisse bzw. eine anpassbare Auswertung der Tests.
  Die meisten kommerziellen Werkzeuge bieten die Möglichkeit die Testergebnisse in XML, Excel oder HTML ausgeben zu lassen.
  Einige Open-Source Engines wie zum Beispiel NUnit schreiben die Ergebnisse der Testdurchführung in eine XML Datei, welche mit Hilfe eines Parsers schnell in eine übersichtliche Excel Tabelle überführt werden kann.

- **Logging**
  Um alle Interaktionen und Ereignisse die zwischen dem Testsystem und der eingebetteten Software ausgetauscht werden aufzuzeichnen, lohnt es sich diese Daten in eine Logdatei zu schreiben.
  Für diesen Zweck gibt es eine Fülle von Frameworks zum Loggen von Anwendungsmeldungen wie zum Beispiel log4j, NLog, Log4Net, u.v.m.

Mit Hilfe der genannten Komponenten ist es möglich Black-Box-Tests für Software von eingebetteten Systemen durchzuführen.

Darüber hinaus gibt es weitere Module, die bei Bedarf in ein solches Rahmenwerk integriert werden können bzw. sollen:

- Mit Hilfe eines speziellen Moduls **Testfallgenerierung** bestünde die Möglichkeit, aufgrund der Spezifikation der Software entsprechend Testfälle automatisch generieren zu können.
  Eine Voraussetzung für diese Funktionalität ist jedoch, dass die Spezifikation der Software mit Hilfe einer formalen durch den Generator lesbaren Sprache formuliert ist.

- Durch Implementierung eines **Vorlagengenerators** ist es möglich, vorgefertigte Muster für Testfälle zu generieren, die schnell und einfach angepasst werden können.

- Eine weitere, interessante Aufgabe könnte ein **Vergleichsmodul** übernehmen.
  Damit wäre es möglich, die Abweichungen in den Ergebnissen zweier gleicher Testfälle zu unterschiedlichen Ausführungszeitpunkten zu analysieren und auszuwerten.
  Darüber hinaus gibt es noch eine Vielzahl an weiteren Funktionen, welche ein solches Werkzeug übernehmen kann, auf die im Rahmen dieses Newsletters jedoch nicht mehr eingegangen wird.

Praktische Herangehensweise — Software Architektur

Dieser Abschnitt geht auf einen möglichen Architekturvorschlag zur Realisierung eines Testsystems für Black-Box-Tests von eingebetteten Systemen ein.

Wie in Abbildung 4. ersichtlich besteht der Implementierungsvorschlag für das Testsystem aus einer vierschichtigen Softwarearchitektur, wobei die Aufgaben zwischen den Layern — je nach zu testendem System — auch unterschiedlich verteilt werden können.

Die Grafik zeigt einerseits die Softwareschichten des Testsystems auf der PC-Seite und andererseits die

(Fortsetzung auf Seite 6)

(Fortsetzung von Seite 5)

Hardware-/Software-Kombination des eingebetteten Systems.

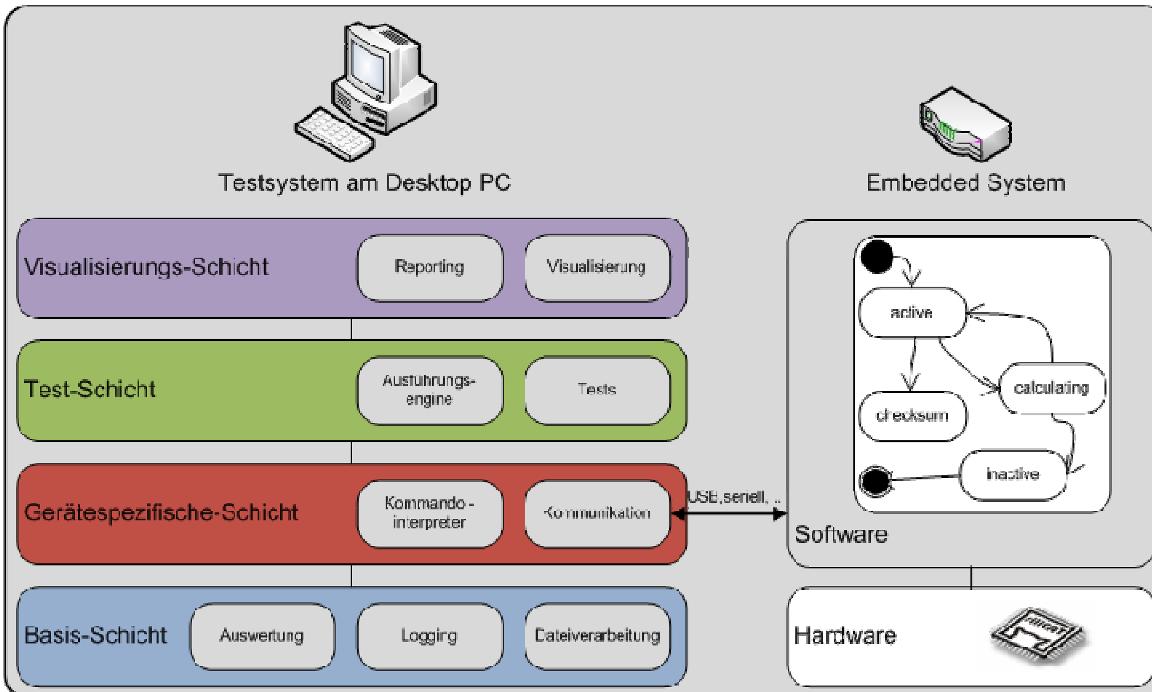


Abb. 4 — Architekturvorschlag (Schichtenarchitektur)
Quelle: Groiß

Der **Basislayer** der Software ist für allgemeine Aufgaben verantwortlich wie zum Beispiel das Auswerten unterschiedlicher Parameter, das Loggen von Daten oder hardwareunabhängige Aspekte wie das Generieren von Dateien oder Verschlüsselung.

Eine Schicht darüber ist der **gerätespezifische Layer** angesiedelt, dessen Aufgaben darin bestehen einerseits die Kommunikation mit der eingebetteten Software herzustellen und aufrecht zu erhalten und andererseits die Kommandos in beide Richtungen zu routen bzw. zu interpretieren. Diese Schicht muss für unterschiedliche eingebettete Systeme bzw. Hardware– oder Softwarekombination entsprechend adaptiert werden.

Der **Testlayer** bietet die Möglichkeit Testfälle zu erstellen bzw. zu warten und zu erweitern sowie diese Testfälle entsprechend auszuführen und die Daten an den Kommandointerpreter bzw. die Kommunikationsschnittstelle weiterzugeben. Nachdem die Antwort der eingebetteten Software gegeben wurde, werden die Antworten und Ergebnisse an die Ausführungsengine weitergeleitet und das Reporting bzw. die Auswertung der Testergebnisse durchgeführt.

Der **Visualisierungslayer** bietet dem Tester einen Überblick über die aktuell durchgeführten Tests und deren Stati. Zusätzlich werden die Ergebnisse mitprotokolliert und über diese Schnittstelle kann auch in Testfälle eingegriffen werden.

Diese Schichtenarchitektur stellt lediglich einen Vorschlag dar, mit dem ein Testsystem für automatisierte Black-Box-Tests realisiert werden kann.

Die verschiedenen Ebenen können ineinandergreifen bzw. mit weiterer Funktionalität ausgestattet werden.

Tools

Im Zuge konkreter Projekte haben sich einige frei verfügbare Werkzeuge und Softwarepakete für Projekte im .NET-Umfeld als äußerst nützlich und leicht integrierbar erwiesen. Einige dieser Werkzeuge werden in der Folge kurz vorgestellt:

- **NUnit**
  Hierbei handelt es sich um ein Werkzeug für die Ausführung von Unit Tests für die .NET- Sprachenfamilie. Im Zuge des Exkurses NUnit wird auf dieses Werkzeug noch im Detail eingegangen.
- **NLog bzw. Log4Net**
  Hierbei handelt es sich um Engines bzw. Bibliotheken, die ein einfaches Loggen auf unterschiedlichen Stufen (Debug, Error, Info, …) in unterschiedliche Formate wie XML, Textdateien, in Datenbanken, auf die Konsole, etc.
  Die Konfiguration des Loggings erfolgt mit Hilfe einer einfach handhabbaren XML-Datei.
- **CarlosAg Excel XML Writer Library**
  Diese Bibliothek bietet die Möglichkeit XML- Dateien ganz einfach zu parsen und in eine formatierte, leicht zu lesenden Excel-Datei zu überführen.

Exkurs - NUnit

von Bernhard Groiß, MSc, Software Test Consultant bei Software Quality Lab

Das quelloffene Testing-Framework NUnit wurde entwickelt, um Tests für alle in .NET Sprachen (C#, VB, C++.net, J#) geschriebenen Programme zu erstellen und durchzuführen. Es handelt sich bei diesem Framework um das Pendant zu JUnit für Programme die in der Programmiersprache Java geschrieben wurden.

In diesem Exkurs wird dieses Testing-Framework im Überblick vorgestellt.

Konzeptionell basieren sowohl NUnit als auch JUnit auf dem vom amerikanischen Informatiker Kent Beck entworfenen xUnit Framework, das in seiner ursprünglichen Variante für die Programmiersprache Smalltalk geschrieben wurde. NUnit wurde in seiner aktuellen Version 2.5.2 (Stand: September 2009) vollständig in C# implementiert.

Tests in NUnit müssen als Methode realisiert, und mit speziellen, in eckigen Klammern geschriebenen Attributen versehen werden. Die Ausführung der Tests kann entweder über ein Kommandozeilenwerkzeug (Console runner) oder über eine interaktive und übersichtliche grafische Benutzeroberfläche (GUI runner) gestartet werden. Die freie Entwicklungsumgebung SharpDevelop bietet bereits eine integrierte NUnit Unterstützung, d.h. Tests könne ohne die Entwicklungsumgebung zu verlassen in der selben IDE entwickelt werden.

Um Tests implementieren zu können, ist es nötig, eine Testklasse zu schreiben, die mit dem Attribut [TestFixture] ausgezeichnet wird.

Einzelne Testfälle werden mit dem Attribut [Test] ausgezeichnet und beinhalten den Testcode und Assertionen. Nachdem die Klasse übersetzt wurde, können die Tests mit Hilfe eines der beiden Werkzeuge ausgeführt werden.

Vor- und Nachbedingungen, die für alle Testfälle gleich sind können mithilfe der Attribute [Setup] und [Teardown] ausgezeichnet werden.

Testfall-Bausteine und -Funktionalitäten

Die wichtigsten Bausteine und Funktionalitäten um effektive Testfälle gestalten zu können sind:

- **Assertions**
  Mit Assertions legt der Tester Bedingungen fest, die für Objekte oder Methoden zutreffen sollen. Dieses Paradigma erlaubt die Prüfung eines aktuellen Wertes gegen einen erwarteten Wert. NUnit bietet in der Klasse ‚Assert' eine Vielzahl an Methoden zur Modellierung von Assertionen. Schlägt eine Assertion fehl, wird der gesamte Testfall als fehlgeschlagen gekennzeichnet. Es hat sich hierbei als gute Technik erwiesen, lediglich eine Assertion pro Testfall zu formulieren.

- **Attributes**
  Attribute sind keine Besonderheit von NUnit, sondern werden von der .NET-Laufzeitumgebung unterstützt und dienen als Anmerkungen, die für Elemente des Quellcodes (Klassen, Methoden) gesetzt werden können. Sie erlauben dem Tester, Methoden als Testfälle bzw. als deren Vor– und Nachbedingungen zu definieren, in unterschiedliche Kategorien einzuteilen oder gewisse Eigenschaften (Plattform, zeitliche Ausführung, Beschreibungen, etc.) zu definieren. Attribute werden mit Hilfe von eckigen Klammern annotiert und vom NUnit Framework entsprechend interpretiert.

Die folgende Auflistung beinhaltet interessante, praktisch relevante Funktionalitäten von NUnit, die kurz vorgestellt werden:

- **Datengetriebene Tests**
  NUnit 2.5 erlaubt es parametrierbare, bzw. datengetriebene Tests zu implementieren und auszuführen. Die Implementierung basiert auf der Verwendung von Attributen wie [Values] oder [ValueSource]. Die Testdaten können von einer anderen Klasse zur Verfügung gestellt, aber auch in Dateien gespeichert werden.

- **Theories**
  Das Auszeichnen von Tests mit dem Attribut [Theory] ermöglicht dem Tester, Angaben darüber zu machen, welche Bedingungen generell zutreffen sollen. Theories erlauben somit, sehr generalisierte Tests zu implementieren, wie zum Beispiel jene, die mathematische Berechnungen für alle positiven, natürlichen Zahlen durchführen.

- **Timeouts**
  Diese Eigenschaft ermöglicht es, Tests als fehlgeschlagen zu markieren, sollte die angegebene Zeitspanne überschritten worden sein. Dies ist oft bei Systemtests für eingebettete Systeme nötig, da aufgrund von Kommunikationsproblemen längere Wartezeiten auftreten können.

- **Combinatorial**
  Dieses Attribut ermöglicht es, alle möglichen Kombinationen von Daten eines Testfalls zu generieren.

(Fortsetzung von Seite 7)

NUnit bietet hiermit die Möglichkeit, Datenpermutationen für Testfälle zu erstellen.

Das folgende, sehr einfache praktische Beispiel, zeigt wie Tests mit Nunit implementiert werden können. Im nachfolgenden Beispiel ist eine Methode zu testen, welche eine Zufallszahl zwischen Eins und einer Million generiert.
Die Methode könnte z.B. wie folgt aussehen:

```
public int GenerateRandomNumber()
{
   random = new Random(seed);
   generatedNumber
   = random.Next(MIN_VALUE,MAX_VALUE);
   return generatedNumber;
}
```

Für den Unit-Test wird z.B. eine Testklasse implementiert, die mit dem Attribut [TestFixture] versehen ist.
Danach ist man bereits in der Lage, den ersten Testfall (versehen mit [Test]) zu entwickeln, der überprüfen soll, ob die Methode tatsächlich Werte zwischen Eins und einer Million generiert.

```
[TestFixture]
public class RandomTest
{
[Test]
public void TestBoundaryValues()
{
int min_value = 1;
// ...
   int number = randGenertor.
            GenerateRandomNumber();
Assert.That(number,Is.GreaterThan(min_value));
}
}
```

Die Assertion überprüft, ob eine Zufallszahl größer als Eins generiert wurde.
Dieses Beispiel gibt einen kurzen Einblick in das mittlerweile äußerst mächtig gewordenen NUnit-Framework.
Nähere Informationen zum Werkzeuge finden sich auf der Entwicklerseite, die unter den Links (siehe Kasten rechts) angeführt ist.
Abschließend kann gesagt werden, dass NUnit relativ einfach in die eigene Implementierungen von Testframeworks integriert werden kann.

Die Implementierung eines komplexen Frameworks auf dieser Basis wurde von OMICRON electronics GmbH in Zusammenarbeit mit Software Quality Lab durchgeführt.

Literatur und Links

- Ganssle, J. 2007 *Embedded Systems: World Class Designs*. Newnes.
- Bender, K. 2007 *Embedded Systems - qualitätsorientierte Entwicklung.* Springer-Verlag New York, Inc.
- Changhyun B. et al. 2007 *A Case Study of Black-Box Testing for Embedded Software using Test Automation Tool.* Journal of Computer Science 3 (3): 144-148, 2007
- *http://www.embedded.com/*
- *Nunit, http://www.nunit.org/*
- *NLog, http://www.nlog-project.org/*
- *Log4Net,http://logging.apache.org/log4net/index.html*
- *CarlosAgExcelXMLWriter, http://www.carlosag.net/Tools/ExcelXmlWriter/*
- *http://www.icsharpcode.com/OpenSource/SD/*
- *OMICRON electronics, http://www.omicron.at*

Leistungen

- Beratungen im Bereich Embedded Testing, sicherheitskritische Systeme und Standards für eingebettete Systeme
- Durchführung von Testspezifikation für Unit-Tests, Systemtests, Abnahmetests
- Optimierung und Anpassung des Vorgehensmodells für die Software– und Systementwicklung
- Risikoanalysen für Embedded Projekte
- Code-Analysen (statisch, dynamisch)
- Qualitätsbewertung von Programmcode
- Unit-Test Erstellung, Durchführung, Auswertung
- Entwicklung von Testframeworks inklusive Testfällen für Systemtests
- Beratung bei der System- und Software-Validierung
- Automotive-SPICE Beratung in der System– und SW-Entwicklung
- Testprozessanalysen, Testfall-Analysen
- Requirements-Erstellung und –Management
- Tool-Auswahl für das Testen von Embedded Systems
- Seminare zum Testen, Requirements-Eng., usw.

Zitat

“In der Informatik geht es genau so wenig um Computer, wie in der Astronomie um Teleskope.”

Edsger Wybe Dijkstra