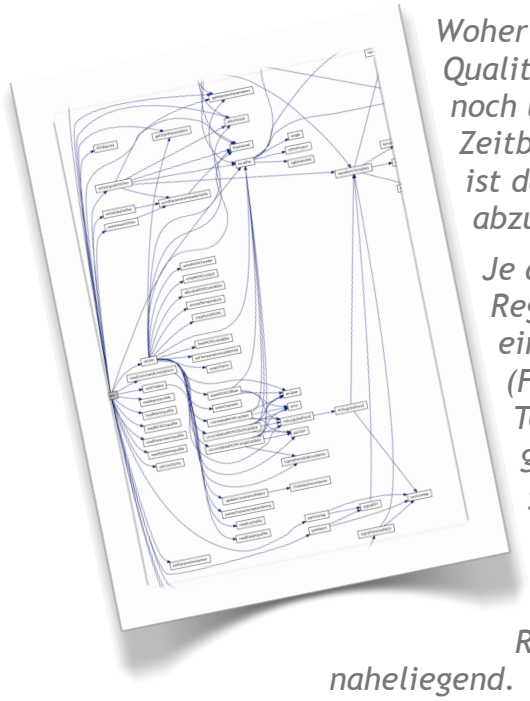
Der steinige Weg zu fehlerfreier Software

Embedded Software Engineering Report Nr. 23

WILLERT.

Über die Effizienz von automatisch ausgeführten Tests *(Regression Test)* zur Qualitätssteigerung von Embedded Software und Alternativen Testverfahren

Die Zeiten fehlerfreier Software gehören der Vergangenheit an. Heutige Software-Applikationen sind in der Regel so komplex, dass es im Bereich des Unmöglichen liegt, sie ohne Fehler zu produzieren. Gute Tests helfen, die Fehler in den kritischen Pfaden und meist genutzten Bereichen der Software (hoffentlich vor der Auslieferung), zu finden und lässt das Auftreten von Fehlern in einem für die Anwender akzeptablen Bereich halten. Erstaunlich viele Fehler bleiben unentdeckt, da sie im Betrieb der Software niemals auftreten. Aber es gibt auch den anderen Fall: Tritt ein Fehler zum falschen Zeitpunkt auf, kann er immensen Schaden anrichten.



Woher aber weiß ein Entwickler, wo er sich bezogen auf die Qualität seiner Software befindet? Enthält seine Software nur noch unkritische Fehler oder gibt es noch eine tickende Zeitbombe? Die Ausführung der Software (der dynamische Test) ist das meist eingesetzte Vorgehen Fehler zu finden und Qualität abzusichern.

Je älter Software ist, desto unverständlicher sind jedoch in der Regel die mit der Zeit gewachsenen Strukturen. Damit einhergehend wächst das Risiko von unerwarteten Effekten (Fehlverhalten) bei Änderungen und in Folge erhöht sich der Test-Aufwand bei Änderungen, wenn die Qualität konstant gehalten werden soll. Eine kleine Änderung ist in kurzer Zeit gemacht, aber die Tests, die notwendig sind um sicher zu stellen, dass diese Änderung kein Fehlverhalten produziert, können ein Vielfaches der Zeit benötigen. Da ist der Wunsch vieler Qualitäts-Verantwortlicher nach so genannten Regression-Tests (Automatisch ausführbaren Tests) naheliegend.

Ob automatisierte Tests immer der effizienteste Weg zu fehlerfreierer Software sind und wenn ja, wie die so genannten Regression-Tests mit vertretbarem Aufwand realisiert werden können, erfahren Sie in diesem Artikel.

Der Weg zu weniger Fehlern

Es gibt verschiedene Methoden Fehler zu suchen bzw. zu finden. Grundsätzlich wird zwischen statischer und dynamischer Analyse unterschieden. Da die dynamische Analyse die sehr viel häufiger angewandte Vorgehensweise ist werden wir uns im Rahmen dieses Artikels auf diese beschränken. An dieser Stelle nur der Hinweis, dass in einigen Situationen mit Hilfe von statischer Analyse sehr viel effizienter Fehler gefunden werden können. Die Auseinandersetzung mit den Möglichkeiten der statischen Analyse ist also ebenfalls lohnenswert.

Wie erfolgreich dynamisches Testen ist, um eine Software mit möglichst wenig Fehlern zu produzieren, ist zu erkennen, wenn diese Vorgehensweise als Funktion von Aufwand zu gefundenen Fehlern dargestellt wird.

Im Fall von händisch ausgeführten dynamischen Tests ergibt sich der in Abbildung Nr.1 dargestellte Verlauf. Zu Anfang werden relativ viele Fehler

gefunden, mit zunehmendem Aufwand werden es immer weniger, bis sich die Kurve asymptotisch der 0 Fehler-Grenze nähert, die jedoch nie erreicht wird.

Um die Aussagekraft dieser Funktion und der konkreten Auswirkung in der Praxis besser zu verstehen, betrachten wir einmal folgende zwei Situationen:

Stellen wir uns im ersten Fall vor, wir entwickeln eine Software-Applikation mit einer der Komplexität entsprechend unzureichenden Vorgehensweise unter schlechten Arbeitsbedingungen *(Zeitdruck, hohe Quote an change requests ...),* dann wird die Software vor der Durchführung der Tests evtl. 100 Fehler beinhalten.

Nun stecken wir einen definierten Aufwand in den Test und finden 70% der Fehler. Bleiben 30 Fehler die wir nicht gefunden haben.

Grad der Komplexität der Software abhängig. Bei minimaler Komplexität ist es eine Gerade, bei steigender Komplexität nimmt die Krümmung zu. Je komplexer unser System ist, desto stärker tritt dieser Effekt zutage.

Grundsätzlich ist es also nicht möglich Qualität in ein Produkt hinein zu testen. Das sollte jedem Projektverantwortlichen bewusst sein. Ich halte diese Aussage für eine der wichtigsten in diesem Artikel. ***Es ist nur möglich Qualität in ein Produkt hinein zu designen***.

Aber im Bereich des Software-Designs haben wir ein ähnliches Verhalten. Mit steigender Sorgfalt im Design können anfangs die Fehlerraten sehr schnell verbessert werden, aber um fehlerfreie Software zu produzieren steigt der Aufwand auch im Design ins Unendliche.

Optimale Effizienz hinsichtlich der Qualität wird erreicht, wenn Aufwand im Design und Test im

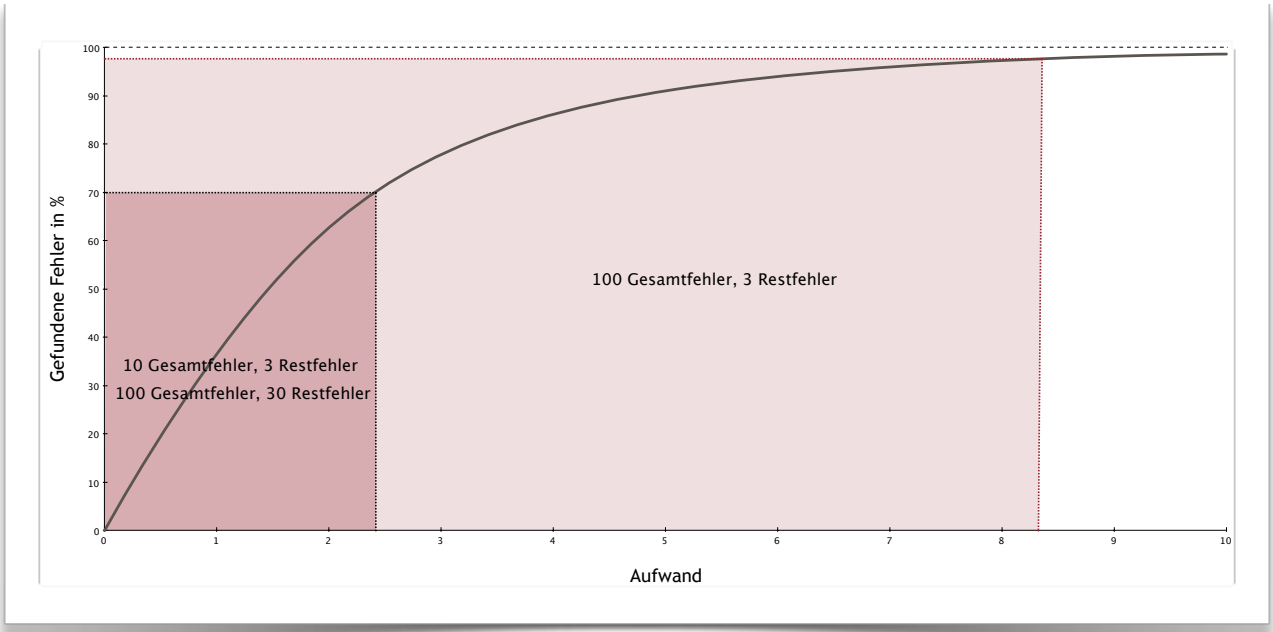


Abbildung Nr.1: Vorgehensweise des dynamischen Testens als Funktion von Aufwand zu gefundenen Fehlern. Krümmung der Kurve und Grad der Annäherung an 100% (Fehlerfrei) hängen von der Komplexität der Software ab.

Nun der zweite Fall. Wir entwickeln auf Basis einer Methode, die der Komplexität der Software gerecht wird und unter idealen Umständen im Arbeitsumfeld. Nun beinhaltet das Resultat evtl. 10 Fehler. Wir spendieren den gleichen Testaufwand und finden wieder 70% der Fehler. Es bleiben 3 Fehler, die wir nicht gefunden haben.

Verlängern wir nun einmal den Testaufwand in der Kurve bis die Fehler im ersten Beispiel auf 3 reduziert wird, dann erkennen wir, der Aufwand steigt überproportional.

Hier wird ein wesentlicher Charakter von dynamischen Tests ersichtlich. Führen wir einen bestimmten Pfad durch eine Software aus, finden wir im optimalen Fall mit dieser Durchführung alle in diesem Pfad auftretenden Fehler. In allen anderen Pfaden bleiben die restlichen Fehler unerkannt. Prozentual auf die Fehler bezogen ändert sich also nichts gegenüber den beiden Situationen hinsichtlich dem Aufwand. Der Verlauf der Kurve ist immer gleich. Er ist lediglich von dem idealen Verhältnis liegen.

In der Praxis ist das sehr schön zu erkennen, wenn dieses Verhältnis über die Lebenszeit der Software verfolgt wird.

Bei neuen Projekten liegt das Verhältnis üblicherweise im Bereich von 60% Implementation und 40% Test. Je älter der Code wird, umso destrukturierter ist er gewöhnlich. Je größer die Komplexität ist und je ungeeigneter die Vorgehensweise bei der Implementation, desto größer wird der Testaufwand, um die Qualität stabil zu halten.

Nicht selten hört man von Kunden, dass das Verhältnis bei 20% Implementations- und 80% Test-Aufwand liegt. Hier ist der Gedanke nach automatischer Durchführung von Tests verständlich, aber deren Einführung ist in dieser Situation häufig nicht nachhaltig erfolgreich. Die Ursache liegt hier in einer unzureichenden Vorgehensweise im Bereich Design und Implementation und das kann durch

dynamisches Testen nicht wirklich ausgeglichen werden.

Die Effizienz von dynamischen Tests kann also nur im Kontext des Vorgehens im gesamten Software-Entwicklungsprozess bewertet werden.

Effizienzsteigerung durch Regression-Tests

Betrachten wir den Prozess der Softwareentwicklung mit all seinen Phasen zum Beispiel im V-Modell, dann haben wir es in den meisten Bereichen mit kreativen Tätigkeiten zu tun, die nicht automatisiert werden können.

Bisher gibt es noch kein Werkzeug, das automatisch Requirements definieren oder eine Erweiterung in der Software durchführen kann. Betrachten wir jedoch den System- oder Integrations-Test, dann sieht es anders aus. Einmal definiert kann er in der Folge mit Werkzeugunterstützung automatisch ablaufen.

Dazu kommt der Umstand, dass Änderungen in der Software, durchgeführt an einer spezifischen Stelle, latentes Potential für Fehler in ganz anderen Bereichen der Software beinhalten. Das Verhältnis von Änderung zu Testaufwand steht in einem sehr ungünstigen Verhältnis. In diesem Fall können auf Basis von Regression-Tests erhebliche Anteile der Arbeiten im Software Engineering automatisiert von einem Werkzeug durchgeführt werden.

Dem gegenüber steht ein anfangs wesentlich erhöhter Aufwand automatische Tests zu erstellen. Dieser amortisiert sich mit dem Grad der Komplexität der Software *(Hohe Komplexität entspricht großem Umfang an Tests)* und der Häufigkeit von Änderungen *(Anzahl der Änderungen im Lifecycle der Software)*.

Je häufiger die Tests durchgeführt werden müssen, desto höher die Amortisation. Demzufolge sollten die Tests bereits zu Anfang des Lifecycles einer Software erstellt werden.

Häufig kommen Kunden zu uns in der Lifecycle-Phase, in der der Aufwand für Tests sehr hoch ist, um diesen dann mit Regression-Tests zu reduzieren. Bereits im vorherigen Kapitel wurde darauf hingewiesen, trotzdem hier noch einmal: Ist komplexe Software mit überalterter Struktur die Ursache für steigenden Testaufwand, dann würde die Einführung von Regression-Tests lediglich an den Symptomen und nicht an der Ursache ansetzen und es muss sehr genau abgewogen werden, ob nicht eine Renovierung der Software, verbunden mit der Einführung einer neuen Software-Architektur und entsprechend der Komplexität besser geeigneten Vorgehensweisen, im Software Design die bessere Alternative ist.

Wie können automatische Tests erstellt werden?

Das Vorgehen zur Erstellung von automatischen Testabläufen beinhaltet folgende Schritte:

- Das Testobjekt wird aus der gesamten Software frei geschnitten.
- Die dabei entstandenen Schnittstellen zum Testobjekt werden ermittelt.
- Es werden sinnvolle und hinreichende Testfälle festgelegt.
- Um das Testobjekt wird ein lauffähiger Rahmen gebaut, der die Schnittstellen parametrieren und Ergebnisse aufzeichnen kann.
- Der Rahmen wird für jeden Testfall parametriert, für Stimulanz und Protokoll. *(Pre- und Post-Conditions werden entsprechend berücksichtigt)*
- Die Testrahmen werden zusammen mit dem Testobjekt ausgeführt und ein Testprotokoll erstellt.
- Das Testprotokoll wird ausgewertet.

Um automatische Testabläufe einfacher zu erstellen, gibt es heute eine große Anzahl an Tools.

Das Vorgehen ist dabei immer wie zuvor beschrieben.

Grundsätzlich ein einfacher Vorgang, in der Praxis ist oftmals jedoch schon das Freischneiden eines Testobjektes ein Problem.

Freischneiden ist notwendig, um die Testparameter einschränken zu können. Die Erstellung eines Testfalls ist um so einfacher, je weniger Parameter für die Stimulanz der Software benötigt werden.

Für Software mit strukturierten Interfaces lassen sich mit wesentlich weniger Aufwand Regression-Tests erstellen, als für Software mit unstrukturierten Interfaces.

Was sind unstrukturierte Interfaces? In der Praxis häufig unkoordinierte Zugriffe auf globale Variablen. Basieren die Interfaces einer Funktion beispielsweise nur auf den Übergabeparametern, lässt sich sehr einfach ein Regression-Test definieren und implementieren.

Wird in der Funktion jedoch auch auf globale Variablen zugegriffen gestaltet sich der Aufwand entsprechend höher. Noch größer wird der Aufwand, wenn ganze Komponenten automatisch getestet werden sollen.

Nun wird in herkömmlicher Software häufig extensiv mit globalen Variablen gearbeitet. Was kann man in solchen Fällen tun? Für das Freischneiden gibt es inzwischen Toolunterstützung. Das Testwerkzeug Tessy zum Beispiel analysiert die zu testenden Software-Komponenten und deren Zugriffe auf Variablen und bietet für die Parametrisierung der Tests alle notwendigen Variablen als Interface an. Der Testrahmen für das Testobjekt kann ebenfalls automatisch erzeugt werden. Lediglich die Parametrisierung muss vom Tester vorgenommen werden.

Der häufiger anzutreffende Ausdruck ‚Design for Testability' hat also seine Begründung.

Es ist also noch einmal mehr zu berücksichtigen, in welchem Zustand sich die Software befindet. Schlecht strukturierte Software mit unspezifischen Interfaces der Komponenten verursacht wesentlich mehr Aufwand bei der Erstellung von automatischen Tests.

Um das komplette Vorgehen so weit wie möglich zu automatisieren gibt es heute sehr gute Toolunterstützung, IBM® Rational® Test RealTime, Tessy, Cantata, TestConductor, um nur einige zu nennen.

Die Durchführung der Tests kann dann mehrere Stunden dauern, könnte z.B. dann automatisiert über Nacht geschehen. Das Protokoll kann in wenigen Minuten überprüft werden. Sind die Regerssion-Tests einmal erstellt, reduziert sich der Aufwand für die wiederholte Durchführung auf die Wartung und Pflege der Interfaces mit ihren Parametern und die Durchsicht der Protokolle. Daraus ergibt sich ein großer Zeitgewinn.

Dieses Vorgehen beinhaltet aber noch einen weiteren Nutzen. Der reduzierte Aufwand zur Wiederholung der Tests ermöglicht z.B. eine sehr viel kürzere Zykluszeit bei der Durchführung von aufwändigen Tests. Einige unserer Kunden fahren die kompletten Integrations- und Systemtests jede Nacht. Das führt dazu, dass Fehler früher gefunden werden. Ein Entwickler, der eine Änderung abgeschlossen hat und sein Modul freigibt, erhält sofort am nächsten Tag die Ergebnisse des Tests, die Vorteile kann sicher jeder nachvollziehen.

Regression Tests haben also nicht nur den Vorteil, dass Fehler mit weniger Aufwand gefunden werden, sondern parallel auch den Vorteil, dass Sie früher gefunden werden.

Höhere Amortisation durch Wiederverwendung

Noch vor einigen Jahren habe ich bei dem Wort *„Wiederverwendung“* immer daran gedacht Software aus einem Projekt in einem anderen wieder zu verwenden. Ohne Frage, Source Code in Folgeprojekten wieder zu verwenden macht die Softwareentwicklung effizienter.

Aber heute sehe ich die Wiederverwendung von Arbeitsergebnissen innerhalb des selben Projektes von einer Phase des V-Modells zu den folgenden als mindestens genau so effizienzsteigernd an.

Um das zu verdeutlichen möchte ich einmal zwei aktuelle Probleme im SW-Engineering beispielhaft herausgreifen.

1. Konsistenz zwischen Dokumentation und Source-Code. In den meisten mir bekannten Projekten wird die existierende Dokumentation als nicht zufriedenstellend bewertet. Das Problem liegt aber nicht so sehr in schlechter Dokumentation, sondern eher darin, dass der Stand der Dokumentation nicht dem Stand der Software entspricht. Hier wird in der Regel noch redundant gearbeitet, Änderungen in der Software müssen von Hand in der Dokumentation nachgepflegt werden und umgekehrt. Unter Zeitdruck im Arbeitsalltag schwierig durchzuhalten.
2. Konsistenz zwischen Pflichtenheft (Anforderungen) und Source-Code. Immer wieder bekomme ich von für System-Test verantwortlichen Mitarbeitern folgende Situation geschildert. Eine der zeitraubendsten Tätigkeiten ist der folgenden Frage nachzugehen, die bei Diskrepanzen zwischen Verhalten des Testobjektes und Beschreibung im Pflichtenheft entsteht. Handelt es sich in diesem Fall um ein Fehlverhalten des Source-Code oder um eine geänderte Anforderung, die im Pflichtenheft nicht nachdokumentiert wurde?

Derartige Probleme entstehen, da heute im Software Engineering in der Regel noch Code- bzw. Dokumenten-zentrisch gearbeitet wird. Arbeitsergebnisse aus einer Phase können nicht automatisch in anderen Phasen übernommen werden, sie müssen von Hand nachgepflegt werden.

Model-zentrisches Vorgehen (auf Basis eines Repository) hingegen ermöglicht Durchgängigkeit und Konsistenz der Arbeitsergebnisse phasenübergeordnet im gesamten Software-Zyklus. Die auf diese Weise erreichte Wiederverwendung von Arbeitsergebnissen beinhaltet enormes Potential für Effizienzsteigerung im Software Engineering.

In der Prozessbeschreibung „Harmony-SE“ von Peter Hoffmann zum Beispiel werden funktionale Anforderungen schon zu einer sehr frühen Phase auf Basis von UML Sequenz-Diagrammen beschrieben.

Die damit beschriebenen funktionalen Anforderungen werden dann im gesamten Software Lifecycle weiter verwendet. Im Architektur-Design dienen sie der Aufteilung des Systems und zur Definition der Schnittstellen, in der Testphase können die selben Szenarien zur Erstellung von Regression-Tests herangezogen werden.

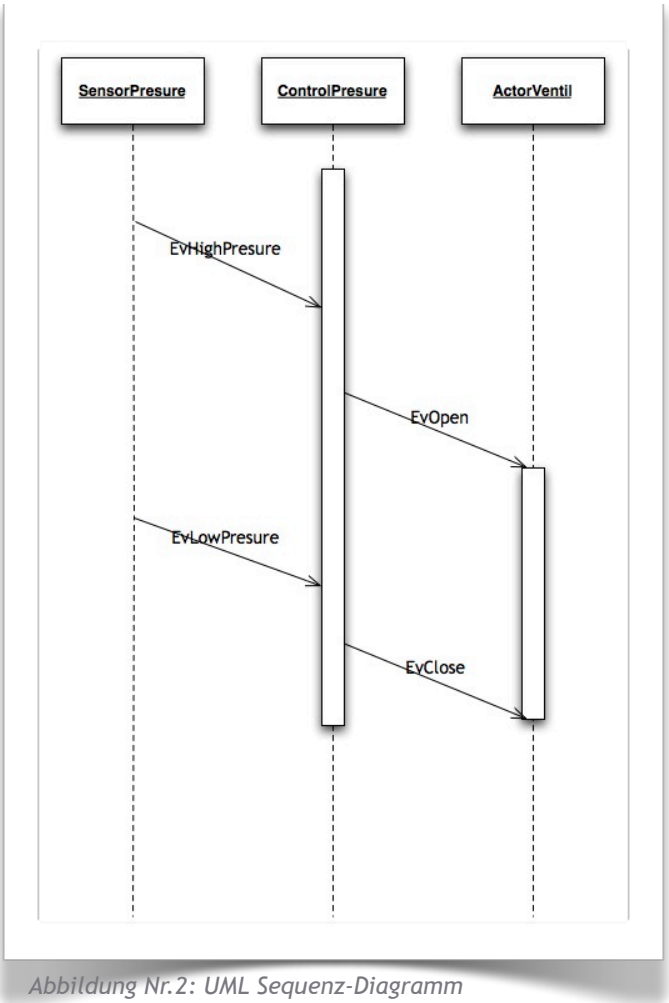


Abbildung Nr.2: UML Sequenz-Diagramm

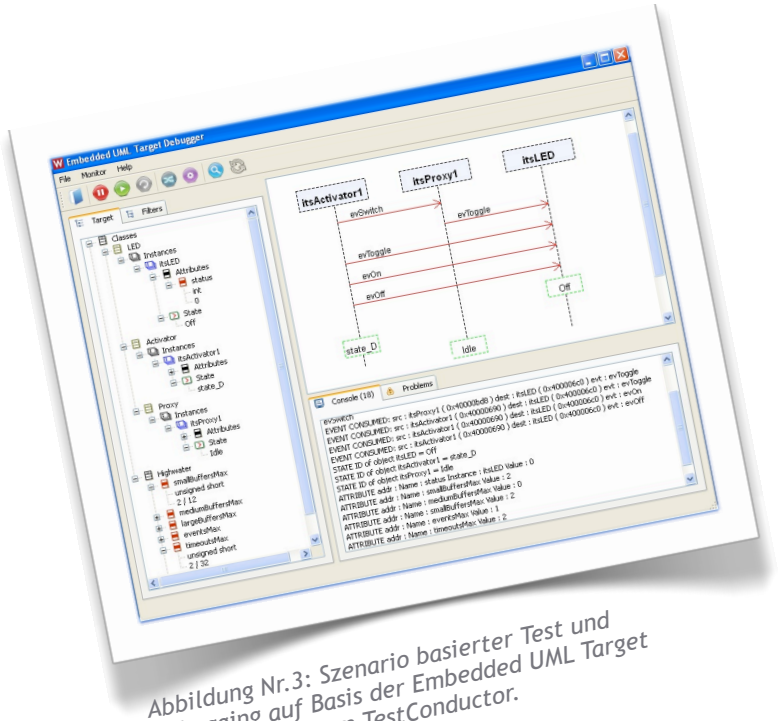


Abbildung Nr.3: Szenario basierter Test und Debugging auf Basis der Embedded UML Target Debugger und dem TestConductor.

Anforderung und Testfallbeschreibung verschmelzen zu einem Element. Notationen wie die UML machen es möglich. Die Notation der UML Sequenz-Diagramme ist festgelegt. Das ermöglicht einem Testwerkzeug, wie zum Beispiel dem TestConductor, auf Basis dieser Diagramme sehr einfach automatisch ablaufende Tests zu erzeugen.

Dieses Vorgehen beinhaltet gleich mehrere Vorteile. Abgesehen davon, dass Abläufe auf Basis von Sequenz-Diagrammen sehr viel aussagekräftiger dargestellt werden können, als auf Basis von Text, können die Diagramme sowohl für Modul-, Integrations- und Systemtests wiederverwendet werden.

In der Software Design-Phase werden auf Basis der Diagramme die Interfaces der Komponenten entwickelt.

Benötigt ein Entwickler innerhalb der Implementation einer Komponente eine Änderung im Interface muss sie auf Basis der Szenarien neu definiert werden. Damit ist sowohl Requirement als auch Testfall kongruent mit der Implementation.

Hinsichtlich den Schritten zur Automatisierung von Tests entfallen mit dieser Vorgehensweise weitgehend das Freischneiden und das Ermitteln der Schnittstellen. Die vorhandenen Szenarien müssen evtl. aus der Testsicht um einige Szenarien ergänzt werden und lediglich die Parametrierung muss noch durchgeführt werden.

Es spricht für sich, dass der Aufwand für die Realisierung von Regression-Tests auf Basis dieser Vorgehensweise sehr effizient möglich ist.

Wann lohnt sich der Aufwand für Testautomatisierung?

Wie bereits mehrfach beschrieben kann die Einführung von automatischen Tests auch kontraproduktiv sein.

Hier noch einmal die Voraussetzungen für optimalen Gewinn an Effizienz durch die Einführung von Regression-Tests.

- Aufwand zur Einführung sollte möglichst früh im SW Lifecycle geschehen. Je früher und damit häufiger die Tests automatisch verwendet werden, desto größer ist die Amortisation.
- Häufigere Durchführung der Tests liefern den Entwicklern wesentlich früher Feedback zu ihren Änderungen bzw. Erweiterungen. Automatische nächtliche Durchführung der Integrationstests liefern dem Entwickler sofort am nächsten Morgen evtl. Fehler zu den Arbeiten des Vortages.
- Modellbasiertes Vorgehen auf Basis der UML verringert zum einen den Aufwand zur Erstellung der Regression-Tests, zum anderen sorgt es für Konsistenz zwischen Anforderungen, Test und Implementation.
- Die Software ist gut strukturiert und einzelne Testobjekte können freigeschnitten werden. Die entstandenen Schnittstellen sind mit vertretbarem Aufwand parametrisierbar.

Gerade der letzte Punkt wird häufig vernachlässigt. Ist die zu testende Software überaltert und hat kein sauberes Design mehr, dann kommt es in der Praxis häufig vor, dass der Aufwand für die Erstellung von Regression-Tests unterschätzt wird. Es wird dann viel Energie in die Einführung gesteckt, aber es kommt niemals zur nutzbringenden Automatisierung. Wäre diese Energie in ein neues Design der Software gesteckt worden, wäre das Ziel eventuell mit mehr Aufwand, dafür aber nachhaltig erreicht worden. Die erfolgreiche Einführung von Regression-Tests ist also abhängig von einer hinreichenden Qualität der Struktur der Software.

In überalterte Software mit unübersichtlicher Struktur kann die Einführung von Regression-Tests genau das Gegenteil bewirken und die Effizienz im Engineering zusätzlich ausbremsen.

Auf Basis einer saubereren Software-Architektur effizient und bereits in den frühen Phasen des Software Lifecycle eingeführte Regression-Tests bewirken, dass Tests kontinuierlicher und umfangreicher durchgeführt werden und führen damit zu weniger Fehlern. Ist der Aufwand für Tests gering, wird das Testen Bestandteil der täglichen Arbeit und nicht eine Phase am Ende des Projektes. Das gibt den Entwicklern frühes Feedback und steigert damit noch einmal die Effizienz und Qualität.

Werkzeugunterstützung

IBM® Rational® Test RealTime

Rational Test RealTime ist ein Testwerkzeug zur automatisierten plattformübergreifenden Komponenten- und Laufzeitanalyse.

Mit Test RealTime lassen sich Probleme in einem frühen Stadium des Entwicklungszyklus erkennen und lösen.

Durch die Nutzung von statischen und Laufzeitanalysefunktionen in Verbindung mit einem Framework für reproduzierbare Komponententests erhalten Tester in der Entwicklung eine Einzellösung für die proaktive Prüfung der Ausführung von Produkten und Software.

Features

- Unterstützt sicherheits- und geschäftskritische integrierte Anwendungen, wie zum Beispiel DO-178B.
- Ermöglicht mehr Proaktivität beim Debugging durch die Erkennung und Korrektur von Fehlern, bevor sie in den Produktionscode gelangen.
- Ermöglicht die automatisierte Quellcodeanalyse mit Berichten zur Einhaltung von Richtlinien für C-Quellcode, wie etwa MISRA-C.
- Kann mit IBM Rational-Lösungen für modellgetriebene Entwicklung, Testmanagement und Softwarekonfigurationsmanagement integriert werden.
- Kann mit führenden Entwicklerwerkzeugen wie IBM Rational Rhapsody, Microsoft Visual Studio und Wind River Workbench integriert werden.
- Unterstützt Eclipse-Plug-ins, was die nahtlose Integration mit den Eclipse C/C++ - Entwicklungswerkzeugen ermöglicht.
- Umfangreiche Zielplattform-Unterstützung ermöglicht Entwicklern die Erstellung und Wiederverwendung von Testassets in mehreren Umgebungen – an spezielle Situationen anpassbar.
- Unterstützte Betriebssysteme: AIX, HP Unix, Linux, Sun Solaris, Windows.

IBM® Rational® Rhapsody® TestConductor

Rhapsody TestConductor ist ein in das UML Softwareentwicklungswerkzeug Rhapsody integriertes Testwerkzeug und die erste UML konforme szenarienbasierte Test- und Validierungsumgebung für eingebettete Echtzeit-Anwendungen. TestConductor testet die in Rhapsody entwickelte Software automatisch auf der Basis von Sequenz-Diagrammen. Da Sequenz-Diagramme bereits im Rhapsody-basierten Softwareentwicklungsprozess modelliert werden, können diese von TestConductor als Grundlage für die Testfalldefinition wiederverwendet werden.

TestConductor überwacht die entwickelte Software in allen Phasen des Entwicklungsprozesses während der Ausführung und kann so in allen wichtigen Anwendungsfällen in Prozesse eingesetzt werden.

TestConductor protokolliert fehlerhafte Läufe mit, die dem Benutzer in Sequenz-Diagrammen grafisch angezeigt werden. Die Fehlerdiagnosezeiten werden dadurch ebenfalls entscheidend gesenkt.

Features

- Tests grafisch definieren durch Verwendung von UML Sequenz-Diagrammen
- Verwendung der Sequenz-Diagramme als Monitore und Testtreiber
- Verwendung parametrisierbarer Sequenz-Diagramme als Schablonen für Testfälle
- Analyse von Fehlern durch farbcodierte Sequenz-Diagramme
- Testausführungen im Batch-Mode
- Unterstützung von Reference Sequence Diagrams
- Funktionsaufruf- und Rückgabewertüberprüfung
- Black-Box-Testing

Embedded UML Target Debugger™

Die meisten UML-Lösungen bieten Simulations und Animations-Möglichkeiten auf Modell-Ebene. Die Techniken, die dabei angewandt werden, basieren in der Regel auf Code-Instrumentierung. Dies wirkt sich jedoch extrem auf Größe und Laufzeitverhalten des Sourcecodes aus und ist für kleine Embedded Systeme aus diesem Grund nicht geeignet, wenn der Code auf der Zielhardware ausgeführt werden soll.

Der Embedded UML Target Debugger arbeitet wie herkömmliche HLL-Debugger auf Basis eines kleinen effizienten Monitors. Der Monitor überträgt Laufzeitinformationen an den PC. Diese werden dort wieder auf die UML-Ebene transformiert und als UML Sequenz-Diagramm dargestellt. Als Schnittstelle von der Zielhardware zum Host PC können die üblichen Debugging-Schnittstellen wie J-Tag oder Emulatoren eingesetzt werden.

Auf diese Weise kann der von Rhapsody generierte Code auf dem realen Zielsystem ausgeführt und getestet werden. Über Backannotation wird das Ablaufverhalten auf der UML-Ebene animiert dargestellt.

Herausgeber:

WILLERT SOFTWARE TOOLS GMBH
Hannoversche Straße 21
31675 Bückeburg

www.willert.de info@willert.de

Tel.: +49 5722 9678 - 60

Autor: Andreas Willert