Test-First = Erst Testen, dann Denken?

Test-Driven Development von Embedded Systemen

Remo Markgraf, MicroConsult GmbH

Prolog

„Sie dürfen nicht alles glauben, was Sie denken!“ - *Heinz Erhardt*
„Glauben heißt nicht wissen.“ - *Wilhelm Weitling*

Ungeniert alle Gedanken herauszuplaudern ist nicht immer die beste Idee, zumal die Umwelt mit ungefilterten, teils abstrusen Gedanken belastet würde. Um das zu vermeiden, müssen wir mehr über unsere Gedanken wissen. Darum testen wir implizit erst unsere Gedanken gegen das jeweilige Umfeld und lassen sie nur in korrigierter Form nach außen dringen. Merken Sie was? Wenn das kein klassischer Test-First-Ansatz ist!

Abbildung 1: Test-Driven Development - mit kleinen Schritten ans Ziel

Zusammenfassung

Viele agile Entwicklungsframeworks verweisen auf den Test-First-Ansatz, der unabhängig von der Teststufe darauf beruht, als ersten Schritt zur eigentlichen Realisierung von Funktionalität mit dem Testen zu beginnen: Testen zu einem Zeitpunkt, an dem man noch mit dem „was“ beschäftigt ist und das „wie“ noch vor einem liegt. Test-Driven Development (TDD) ist die Umsetzung des Test-First-Ansatzes im Komponententest und bedeutet das Schreiben der Unit-Testfälle vor der eigentlichen Implementierung. Die Einhaltung von nur drei Regeln und ein paar Tricks im Umgang mit dem Target-Hardware-Bottleneck ermöglichen TDD auch für Embedded-Systeme.

Test-First-Ansatz

Die Kosten zum Auffinden von Fehlerzuständen und deren Beseitigung nehmen mit dem Projektfortschritt deutlich zu und erreichen teils enorme Höhen zur Beseitigung im Feld. Der Test-First-Ansatz hat das Ziel, Fehler so früh wie möglich aufzudecken, indem Tests erstellt und geprüft werden, noch bevor der jeweilige Testgegenstand existiert. Test-First lässt sich in allen Teststufen anwenden, vom Acceptance-Test bis zum Komponenten-Test und der Implementierung von Software mittels Test-Driven Development (TDD).

Test-First in den verschiedenen Teststufen

- **Test-First im Acceptance-Test**
  In agilen Entwicklungsmodellen, z.B. Scrum, wird üblicherweise mit User- und Systems-Stories gearbeitet; daraus werden zu entwickelnde Backlog-Items

abgeleitet. Der Test-First-Ansatz auf dieser Ebene bedeutet, dass, noch bevor darüber nachgedacht wird, wie ein Backlog-Item realisiert werden kann, Akzeptanzkriterien ermittelt werden. Diese Akzeptanzkriterien müssen erfüllt sein, damit ein Backlog-Item in der Sprintabnahme als „done“ gewertet werden kann. Durch die Definition der Abnahmetests vor der eigentlichen Realisierung entstehen vollständigere und besser testbare User/System-Stories, da alle in den Testfällen genannten Bedingungen, Eingaben und erwartete Ausgaben in die User/System-Stories einfließen. Die auf den Backlog-Items basierende Aufwandsschätzung ist daher leichter und genauer zu bewerkstelligen.

- **Test-First im Systemtest**
  Im Systemtest liegt der Fokus mehr auf den Last- und Stresstests in einer möglichst betriebsnahen Umgebung, um die nicht-funktionalen Qualitätsmerkmale nach ISO/IEC 9126 der Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit nachzuweisen. Auch hier bedeutet der Test-First-Ansatz die Spezifikation und Erstellung der Systemtests, bevor das Produkt selbst entwickelt ist. Aus der Erstellung der Systemtestfälle ergeben sich oft technische Anforderungen, die zu einem möglichst frühen Zeitpunkt unter dem Begriff „Design for Testability“ noch in die Planung und das Design des Produktes einfließen können.

- **Test-First im Integrationstest**
  Der Integrationstest erfordert ein stufenweises Vorgehen, in dem immer mehr Komponenten im Integrationstest zusammengefügt werden. Im Test-First-Ansatz werden schrittweise Integrationstests zusammen mit den erwarteten Ergebnissen erstellt, bevor die Komponenten überhaupt existieren. Schnittstellenprobleme können dadurch frühzeitig entdeckt und im Design und der Implementierung noch behoben werden. Die Spezifikation der Schnittstellen erfolgt unter dem Testaspekt, führt dadurch vom Einfachen zum immer Komplexeren hin und ergibt besonders schlanke und gut testbare Interfaces.

- **Test-First im Komponententest**
  Die Umsetzung des Test-First-Ansatzes im Komponententest bedeutet das Schreiben der Unit-Testfälle vor der eigentlichen Implementierung. Genau dieses Ziel verfolgt der Ansatz des Test-Driven Development in kleinen Schritten. Das aus dem Extreme Programming stammende TDD ist nicht Teil des Scrum-Entwicklungsframeworks, fügt sich aber perfekt ein. In Scrum werden Sprint-Backlog-Items zur Abarbeitung in Tasks heruntergebrochen. Die kleinste übliche Granularität des Aufwands einer Task liegt bei einem Entwicklungstag. Weiter herunter reicht das Scrum-Framework nicht. Genau hier setzt TDD an und bricht Tasks in kleine Schritte herunter, die man iterativ abarbeitet - natürlich wieder nach dem Test-First-Ansatz: erst Testen, dann Implementieren.

TDD-Cycle

Im Test-Driven Development entsteht der Code zusammen mit den Unit-Testfällen im TD- Cycle in kleinen, wiederholten Mikro-Schritten. Im Fokus eines TDD-Cycles liegt jeweils eine abgeschlossene Funktionalität, z.B. eine Funktion in C, ein C-Modul, eine C++ Klasse oder eine Methode eines Objektes. Eine Iteration im TDD-

Cycle besteht aus drei Hauptbestandteilen und dauert nur wenige Minuten - je kürzer, desto besser.

Die drei Hauptteile nennt man Red, Green und Refactor:

- **Red**
  Schreibe einen Test, der eine noch zu erstellende Funktionalität prüft. Da die Funktionalität noch nicht existiert, muss der Test bei Ausführung noch fehlschlagen (Red).
- **Green**
  Implementiere die fehlende Funktionalität, bis der Code den neuen Test und alle zuvor im Fokus dieses TDD-Cycles erstellten Tests besteht. (Green)
- **Refactor**
  Nur wenn alle zuvor erstellten Tests bestanden sind, darf optional ein Refactoring des Codes und/oder der Tests durchgeführt werden. Durch das Refactoring darf nur die innere Struktur verändert werden. Es darf keine von außen sichtbare Funktionalität hinzugefügt oder entfernt werden. Am Ende des Refactorings sind alle Testfälle erneut auszuführen und zu bestehen.

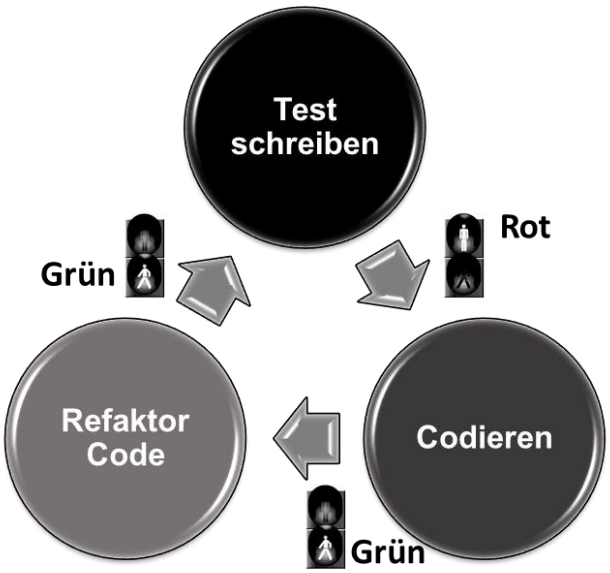

Abbildung 2: Test-Driven Development Cycle

Die drei Hauptschritte des TDD-Cycles werden so lange wiederholt, bis die geplante Funktionalität entwickelt und getestet wurde. Die konsequente Umsetzung dieses Vorgehensmodells fördert die schrittweise Verfeinerung vom einfachen Fall zu immer komplexeren Fällen hin. Um den sprichwörtlichen Wald vor lauter Bäumen nicht aus den Augen zu verlieren, ist es dringend erforderlich, eine Testliste mit den geplanten Iterationen des TDD-Cycles zu erstellen. Diese Liste ist Ihr persönlicher Abarbeitungsplan, wird nicht dauerhaft gespeichert und darf jederzeit verändert werden. Sie dürfen jederzeit einzelne Schritte hinzufügen, streichen, umbenennen oder in der Reihenfolge verändern.

Egal, ob eine neue Funktionalität oder nur eine Erweiterung entwickelt und getestet

werden soll, ist zu Beginn die Erstellung einer persönlichen Testliste dringend angeraten.

WICHTIG: Erstellen Sie unbedingt auch eine Testliste für vermeintlich kleine Change Requests!

Die drei TDD-Regeln der kleinen Schritte

Test-Driven Development lässt sich nach Bob Martins[1] durch drei einfache Regeln beschreiben:

1. Schreibe keinen Code, so lange dieser nicht zur erfolgreichen Durchführung eines Testfalls erforderlich ist.
2. Begrenze den Umfang eines Unit-Testfalls darauf fehlzuschlagen. Auch Compiler-/Linker-Fehler sind Fehler.
3. Schreibe nicht mehr Code als erforderlich ist, um den aktuellen Unit-Testfall zu bestehen.

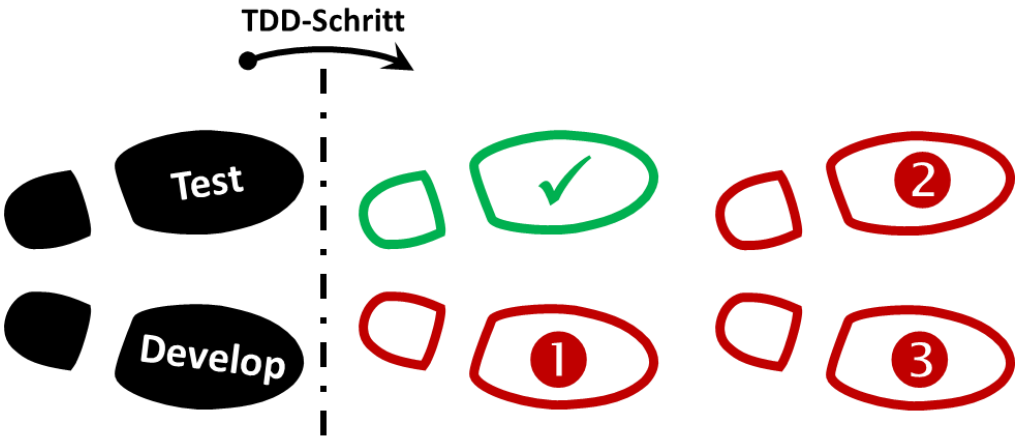

Abbildung 3: Die drei TDD Regeln der kleinen Schritte

Nach Regel 1 müssen Sie zur Entwicklung einer Funktionalität mit dem Erstellen eines Unit-Tests beginnen. Aber nach Regel 2 können Sie nicht sonderlich viel in diesem Unit-Test testen, denn sobald der Unit-Test scheitern kann, müssen Sie mit der Entwicklung des Unit-Tests stoppen und produktiven Code schreiben, um den Test zu bestehen, aber eben auch nicht mehr, da sonst Regel 3 verletzt würde.

Die Einhaltung dieser 3 TDD-Regeln erinnert uns stets daran, dass wir uns in ganz kleinen iterativen Schritten auf das Ziel zubewegen müssen. Nehmen wir an, unser linkes Bein wäre unser Tester-Bein, während das rechte unser Entwickler-Bein wäre (siehe Abbildung 3). Wir starten jede Iteration des TDD-Cycles mit dem Tester-Bein, denn würden wir mit dem Entwickler-Bein starten, wäre Regel 1 verletzt. Machen wir unseren Schritt mit dem Tester-Bein zu groß, bestünde die große Gefahr, dass uns beim späteren Refaktorieren Fehler durchrutschen und unentdeckt bleiben, bis sie später schmerzlich zutage treten. Regel 2 zwingt uns also zu ganz kleinen Schritten mit dem Tester-Bein und erleichtert uns damit das Refaktorieren. Wenn Sie

im Zweifel sind, ob ein Schritt zu groß sein könnte, entscheiden Sie sich immer besser für mehrere kleinere.

Wir dürfen nun endlich mit unserem Entwickler-Bein den Schritt nach vorne wagen, aber gemäß Regel 3 nicht an dem vorgeeilten Tester-Bein vorbei, sondern eben nur bis auf gleiche Höhe. Wir stehen wieder mit beiden Beinen nebeneinander auf dem Boden. Das genau ist der einzige erlaubte Startpunkt zum optionalen Einschieben einer Refatorierung, um nach dem Bestehen aller bis dahin entstandenen Testschritte den nächsten kleinen Schritt in Richtung Ziel zu machen, natürlich mit dem Tester-Bein zuerst, oder wollen Sie mit dem falschen Bein aufstehen? :-)

Target-Hardware-Bottleneck

Das Target-Hardware-Bottleneck, ein Hindernis beim TDD von Embedded-Systemen, entsteht speziell bei der gleichzeitigen Entwicklung der Target-Hardware und der Software. Beim Entwickeln und Testen der Software ist noch keine Target-Hardware vorhanden, um TDD umzusetzen. Dazu kommen Probleme mit automatischen Test-Suiten auf dem Embedded-System, knappe Ressourcen und lange Flash-Load-Zyklen. TDD auf Embedded-Systemen braucht also spezielle Verfahren, um diese Probleme zu behandeln.

Dual-Targeting ist die Voraussetzung zum Umgehen der eben genannten Probleme. Von Tag eins an wird die Software weitestgehend plattformunabhängig entwickelt, zumindest aber für mindestens zwei Plattformen, die Entwicklungs- und die Target-Plattform. Das ermöglicht die Entwicklung mit konstanterer Geschwindigkeit, vermeidet Probleme mit der Anhäufung von ungetestetem Code und erlaubt das Testen von Code vor der Verfügbarkeit der Target-Hardware.

Basierend auf dem Dual-Targeting lassen sich verschiedene Teststrategien anwenden:

Test on Host
Für weitestgehend hardwareunabhängige Softwareteile kann als Testumgebung die Entwicklungsplattform (Host), zumeist der PC, eingesetzt werden. Das Testtool und die zu testende Software laufen auf dem Host. Damit lassen sich Vorteile kombinieren, wie die Vermeidung von Flashzeiten, sehr kurze Cyclezeiten und komfortable Test-Tools. Der Nachteil ist die Notwendigkeit, die Tests zu einem späteren Zeitpunkt auf dem Target nochmals zu wiederholen, um deren Funktion auf dem Zielsystem zu verifizieren. Als Test-Tools bieten sich hier neben professionellen Tools, wie Tessy oder Parasoft C++, auch die kostenfreien Tools Google Test und Mock an.

Test on Target
Hardwareabhängige Softwareteile, z.B. Device Driver, müssen auf der Target-Plattform getestet werden. Um Zeit zu gewinnen, ist es durchaus möglich, zunächst auch diese Softwareteile mit einer Test-on-Host-Strategie zu entwickeln und dabei die Hardwareabhängigkeit durch Mocks zu simulieren. Zeitnah mit Verfügbarkeit der Target-Hardware müssen die Tests dann mit der Test-on-Target-Strategie wiederholt werden. Die verwendeten Test-Tools müssen hier auf dem Target laufen. Daher können nur kleine und dadurch funktional eingeschränkte Test-Tools, wie z.B. Embedded Unit oder CppUnit, verwendet werden.

Remote Testing
Das Remote Testing verbindet die Vorteile der Test-on-Host- mit der Test-on-Target-Strategie. Die Grundidee liegt darin, das verwendete Test-Tool auf dem Host laufen zu lassen und hardwareabhängige Tests zunächst auf dem Host und später über eine Interprozesskommunikation auf dem Target auszuführen. Geeignete Test-Tools, z.B. Tessy, unterstützen dieses Verfahren, ohne die Tests verändern zu müssen. Die Interprozesskommunikation erfolgt dabei oft nicht direkt mit dem Target, sondern zwischen Test-Tool und der Entwicklungsumgebung. Das Test-Tool instrumentiert dazu den Code und lädt diesen über die Entwicklungsumgebung zur Testausführung auf das Target. Die Testergebnisse werden dabei über die Debugger-Schnittstelle zurückgelesen und im Test-Tool angezeigt. Aus Sicht des Benutzers sieht das Ergebnis eines Testlaufs auf dem Host und dem Target gleich aus. Er braucht also nur das Target umzustellen und muss durch das Up-/Download etwas länger auf die Testergebnisse warten.

TDD ersetzt nicht DAS Testen

TDD ist auch auf Embedded-Systemen erfolgversprechend einsetzbar. Die Qualität der Unit-getesteten Zulieferung an den Integrations-, System- und Akzeptanztest wird definitiv im Vergleich zu traditionell Unit-getesteter Software steigen. 40-50% weniger entdeckte Fehler im Systemtest dürfen Sie bestenfalls erwarten[2].

Gerade die erfolgversprechende letzte Aussage macht aber auch deutlich:
Test-Driven Development ersetzt nicht DAS Testen; vielmehr ist es eine strukturierte Methode des entwicklernahen Unit-Testens zusammen mit der Codeentwicklung. Integrations-, System- und Akzeptanztests können sehr vorteilhaft den Test-First-Ansatz verwenden, müssen aber für ein qualitativ hochwertiges Ergebnis dem TDD nachgeschaltet sein.

MicroConsult (https://www.microconsult.de/) ist auf Ausbildung, Weiterbildung und Beratung für Hersteller von Embedded-Systemen spezialisiert. Sehr gerne unterstützen wir Sie mit Rat und Tat auf Ihrem Weg zur Einführung neuer Testmethoden.

Quellen

[1] Bob Martins: *http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd*
[2] *Software Engineering, 2003. Proceedings. 25th International Conference on*
[3] Agile-TDD: *https://www.microconsult.de/1131-0*

Autor

Dipl.-Ing. (Univ.) Remo Markgraf ist Senior Management Consultant bei der MicroConsult GmbH. Neben Begeisterung für Innovation und Leidenschaft für Embedded-Systeme verfügt er über langjährige Projekt- und internationale Führungserfahrung in Softwareentwicklung, Systems und Test Engineering, technischen Vertrieb, Projekt-, Produkt-, Innovations- und Business Development Management. Er war als Leiter des Produktmanagements der Nokia Solutions Oy maßgeblich an der Einführung agiler Prozesse beteiligt. Er lehrt und berät umfassend zu den Themengebieten Cortex-M, Testen, Test-Driven Development und agile Entwicklung von Embedded-Systemen in deutscher und englischer Sprache. Seinen Erfahrungsschatz aus vielen internationalen Großprojekten gibt er nicht nur in den Trainings und Consulting-Projekten der MicroConsult GmbH, sondern auch in aktuellen Veröffentlichungen und Vorträgen an Kunden weiter.