# Test-Driven Development von Embedded-Systemen – Teil 2: Die drei TDD-Regeln der kleinen Schritte

**Test-Driven Development (TDD) ist die Umsetzung des Test-First-Ansatzes im Komponententest und steht für 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 ermöglicht TDD auch für Embedded-Systeme.**

**Test-Driven Development lässt sich nach Bob Martins 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, um den aktuellen Unit-Testfall zu bestehen.

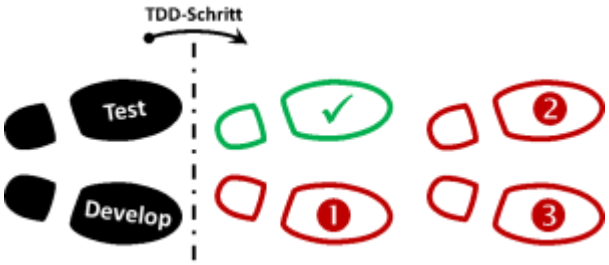


*Abbildung: Die drei TDD-Regeln der kleinen Schritte*

Nach Regel 1 beginnt die Entwicklung einer Funktionalität mit dem Erstellen eines Unit-Tests. 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.

## Kleine Schritte mit dem Tester-Bein

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 vorausgeeilten Tester-Bein vorbei, sondern eben nur bis auf gleiche Höhe. Wir stehen nun wieder mit beiden Beinen nebeneinander auf dem Boden. Das genau ist der einzige erlaubte Startpunkt zum optionalen Einschieben einer Refaktorierung, 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.

Eine Voraussetzung zum Umgehen der eben genannten Probleme ist das Dual-Targeting. 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 konstanter Geschwindigkeit, vermeidet Probleme mit der Anhäufung von nicht getestetem Code und erlaubt das Testen von Code vor der Verfügbarkeit der Target-Hardware.

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

### 1. 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.

2. **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 sollten die Tests dann mit der Test-on-Target-Strategie wiederholt werden. Die verwendeten Test-Tools müssen hier auf dem Target laufen. Dabei können nur kleine und dadurch funktional eingeschränkte Test-Tools, wie z.B. Embedded Unit oder CppUnit, verwendet werden.

3. **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.

**Wichtig: TDD ersetzt das Testen nicht!**

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

Gerade die erfolgversprechende letzte Aussage macht aber auch deutlich:

Test-Driven Development ersetzt nicht das Testen selbst; 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.

**Teil 1 des Beitrags behandelt den Test-First-Ansatz und TDD-Cycle.**

## Weiterführende Informationen

MicroConsult Training & Coaching zum Thema Test & Debug

MicroConsult Fachwissen zum Thema Test & Debug

MicroConsult Training & Coaching zum Thema Qualität, Safety & Security

MicroConsult Fachwissen zum Thema Qualität, Safety & Security

## Autor

**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 Engineering, Projekt-, Produkt-, Innovations- und Business Development Management sowie dem technischen Vertrieb.