Konsistente Metriken zur Ermittlung der Testaktivitäten

Jederzeit wissen, wie weit die Entwicklung wirklich ist

Ingo Nickles, Vector Software

Abstract

In der modernen Softwareentwicklung gilt es, neue Funktionalitäten in immer kürzeren Zeitabständen zu implementieren. Der Quellcode wird immer umfangreicher und komplexer, dennoch wird erwartet, dass die Software Qualität gleich bleibt oder sich sogar noch verbessert. Die größte Herausforderung ist sicherlich, Release-Zyklen zuverlässig vorherzusagen und einzuhalten. Oft werden Aufwände falsch eingeschätzt und Ressourcen ungünstig verteilt, denn die Projektleiter wissen häufig nicht genau, welche Codeänderungen welche Ressourcen benötigen.

Natürlich tragen im Software Entwicklungsprozess viele Dinge zur Software Qualität bei. Neben einem funktionierenden Anforderungsmanagement sind dies z.B. eine entsprechende Software Architektur und ein gutes Design. Nichts desto trotz wird die Sofware Qualität aber am Ende im Testprozess gemessen bzw. nachgewiesen und so ist es durchaus sinnvoll, sowohl den Testprozess als solches als auch die sich daraus ergebenden Metriken näher zu beleuchten.

Durch kontinuierliches Testen sowie die kontinuierliche Integration von Code Änderungen und angepassten Testfällen ins Configurationn Management können häufig auftrende Termin-Verzüge vermieden werden. Änderungs-basiertes Ausführen der Testfälle erlaubt es jedem Entwickler, automatisiert alle Arten von Testfällen auszuführen die von einer Code-Änderung betroffen sind.

Konsistente und nachvollziehbare Metriken können wichtige Hinweise auf den momentanen Entwicklungsstand, den Entwicklungsverlauf und den aktuellen Testfortschritt liefern. Durch Kenntnis dieser Daten ist es möglich, einen sehr genauen Überblick über den momentanen Projektstatus und die Software-Qualität zu erhalten. Dadurch lässt sich ein möglicher Release-Zeitpunkt genauer vorhersagen. Ebenso ist es zu jeder Zeit möglich, eventuelle Engpässe zu erkennen und Ressourcen besser zu planen.

Einführung

Betrachtet man die Entwicklung von Software (SW)- zu Hardware (HW) Anteil in Produkten so ist ein deutlicher Trend zu mehr SW erkennbar. SW wird mehr und mehr zum „Differentiator“ – zu dem, was den Unterschied zwischen den Produkten zweier Hertsteller und damit den Wettbewerbs-Vorteil ausmacht. Dadurch, dass mehr Funktionalität in die SW wandert wird diese natürlich immer komplexer – aber auch wichtiger für den Produkt-Hersteller. Eine hohe Produkt-Qualität hängt immer mehr auch von fehlerfreier Software ab. Durch Einführung entsprechender Entwicklungs-Richtlinien kann die Implementierung von Fehlern ins Produkt sicherlich vermindert werden – dennoch kann und darf man auf einen ausführlichen Testprozess nicht verzichten. Betrachten wir also den Testprozess innerhalb eines Entwicklungsprozesses etwas genauer.

Software Test

Der wohl verbreitetste Entwicklungsprozess ist das V-Modell, das aufgeteilt ist in 3 Phasen:

- Design Phase
- Codierungs Phase
- Test Phase

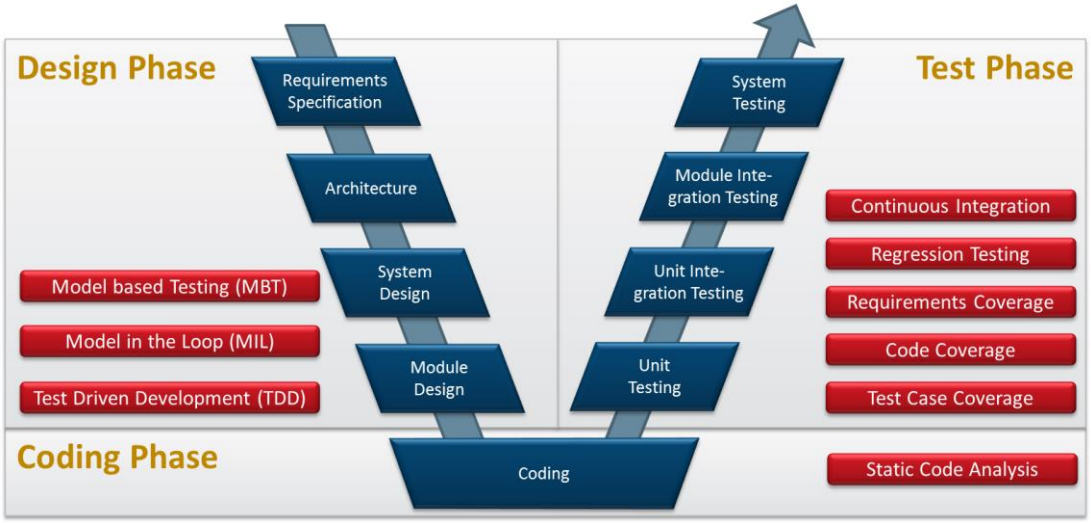


Abbildung 1 – Tests im V-Modell

Obwohl im V-Modell die Testphase an das Ende des Entwicklungszyklus gestellt ist, hat man bald erkannt, dass es durchaus sinnvoll ist, sehr viel früher mit dem Testen zu beginnen, denn je früher ein Fehler gefunden wird desto günstiger ist dessen Beseitigung.

Im klassischen V-Modell versucht man dieser Tatsache dadurch Rechnug zu tragen, dass man die kleinste isolierbare Einheit der Software, sobald verfügbar, auf Herz und Nieren testet: Eine Unit, bzw. ein einzelnes Source File. Nachdem das einzelne File, bzw. die darin implementierten Funktionen, alle Tests überstanden haben fasst man mehrere Files zusammen und überprüft deren Zusammenspiel im Integrationstest. Im Systemtest wird schließlich die gesamte Software getestet. Betrachten wir also die „rechte Seite“ des V-Modells genauer:

System Test

Testen aller Files. Testen der gesamten Applikation

Black Box:

100% System Requirements

Ca. 50% Code Coverage

White Box: Ca 70% Code Coverage sind möglich

Integrations Test

Testen des Zusammenspiels mehrerer Files

Black Box: 100% High Level Requirements (System Design)

White Box: ca 85% Code Coverage

Unit Test

Testen eines einzelnen Files

Black Box: 100% Low Level Requirements (Modul Design)

White Box: Ziel ist 100% Code Coverage

Statische Code Analyse

Überprüfung des Codes auf formale Korrektheit, Codierungsregeln

Analyse von Speicher-Verwendung und übliche Fehler-Ursachen

Eine zentrale Rolle im Test kommt der Automation zu. Das manuelle Wiederholen immer gleicher Test-Abläufe ist einem Menschen nicht zuzumuten weshalb Test-Automation nicht nur der Mitarbeiter Zufriedenheit dient sondern auch dazu beiträgt, dass Tests auch wirklich ausgeführt werden. Daher sollte bei der Bildung der Test-Landschaft darauf geachtet werden, dass alles, was automatisierbar ist, auch automatisiert wird. Denn nur so ist ein vernünftige Regressionstest realisierbar.

Regressionstest

Am Ende eines V-Entwicklungsprozesses steht eine gut getestete und qualitativ hochwertige Software. Wenn damit die Möglichkeiten einer SW Änderung an Ihrem Produkt enden, weil Sie es z.B. zum Mars schießen, dann spielt für Sie der Regressionstest keine große Rolle. Aber auch dann wenn Sie keine Möglichkeiten von Software Änderungen am fertigen Produkt haben kann sich schon aus dem Einsatz eines anderen Entwicklungsmodells die Notwendigkeit eines Regressionstests ergeben. Z.B. erfordern agile Entwicklungsmethoden ein kontinuierliches erneutes Ausführen der bereits vorhanden Testfälle. Weitere Gründe für SW Änderungen und damit ein Wiederholen aller Tests können sein:

- Fehlerbeseitigungen
- Neue Feature (CI)
- Geänderte Hardware
- Geänderte Anforderungen
- Redesign

Tatsächlich haben Untersuchungen z.B. der FDA (Food and Drug Administration, USA) ergeben, dass ein Großteil (79%) von SW Fehlern, die zu einem Produkt-Rückruf geführt haben, nachträglich durch Code Änderungen ins fertige Produkt eingeführt wurden [1]. Code Änderungen, deren Auswirkungen auf die SW Qualität durch einen entsprechenden Regressionstest hätten entdeckt werden können.

Testumgebungen
Betrachtet man die Anzahl der Testumgebungen, die in einem empfohlenen Entwicklungsprozess in einem Projekt entstehen, so kommt einiges zusammen:

- Eine Unit-Test-Umgebung je Source File
- Mehrere Integrationstestumgebungen mit jeweils mehreren Source Files
- Eine bis mehrere Testumgebungen mit allen Source Files für den Systemtest

Hinzu kommt, dass es in der Regel mehr als eine Konfiguration gibt, in der alle Testumgebungen ausgeführt werden sollten. So ist es z.B. offensichtlich sinnvoll und von verschiedenen Standards auch gefordert, die Tests auf dem späteren Ziel-System bzw. einem entsprechenden Board auszuführen. Nun findet ein Unit Test in der Regel aber sehr früh im Entwicklungs Prozess statt. Ein Board zur Ausführung der Unit Tests ist aber möglicherweise selbst Teil des zu entwicklenden Produkts und steht dementsprechend nocht nicht für eine Ausführung von SW Tests zur Verfügung.

Aber auch bei vorhandener HW kann es sinnvoll sein die echte HW zu meiden. Z.B. wird üblicherweise die Test-Applikation auf die HW „ge-flashed“ und die Anzahl der von der HW unterstützten Flash-Zyklen ist dabei begrenzt. Um einen übermäßigen Verschleiß an HW zu vermeiden kann man also auf Simulatoren oder Emulatoren umsteigen. Dies macht auch das Ausführen von Testfällen schneller, was der Ungeduld des Testers beim Erstellen der Testfälle entgegenkommt.

Bei nicht-vorhandensein von Simulatoren kann es auch sinnvoll sein, eine Testumgebung auf dem Host aufzusetzen. Natürlich ist dies mit größerem Aufwand bei der Testumgebungs-Erstellung verbunden: Cross-Compiler spezifische Keywords müssen „weg-definiert“ werden um den Code mit einem Host Compiler kompilieren zu können. Hardware Zugriffe müssen „weg-gestubbt“ werden bzw. durch das Einbinden anderer Header Files ausgetauscht werden. Nichts desto trotz macht sich dieser initiale Aufwand später bezahlt durch verschiedene Vorteile: Schnellere Testfall Erstellung, Schonen der HW Resourcen, und nicht zuletzt ein Überprüfen der Software in einem weiteren Testumfeld (anderer Compiler, anderes Betriebssystem). So wird Ihnen Windows z.B. einen „Segmentation Violation Error“ beim Zugriff auf invalide Pointer liefern, während der gleiche Testfalls auf Ihrem RTOS im schlimmsten Fall sauber durchläuft.

Last but not least wird SW haüfig in mehr als einer Konfiguration an spätere Kunden übergeben. Z.B. kann es verschiedene Ausbaustufen Ihres Produkts geben die auf unterschiedlichen Boards laufen. Oft werden auch Funktionalitäten in der SW per Compile-Defines an- bzw. ausgeschaltet. Oder zentrale Teile der Software laufen in komplett unterschiedlichen Produkten.

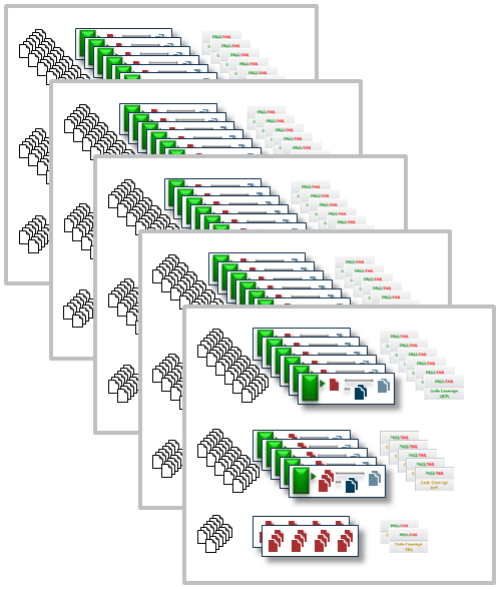


Abbildung 2 – Unit-, Integrations- und System-Testumgebungen in 4 Konfigurationen

Was bedeutet das jetzt für die Anzahl der Testumgebungen?

```
Anzahl der Testumgebungen  =
    1 Host Konfiguration
  + 1 Simulator Konfiguration
  + (Anzahl Boards * Anzahl Compile-Define-Kombinationen)
```

Die „Compile-Define-Kombinationen“ in denen Ihr Produkt schließlich ausgeliefert wird ist sinnvollerweise definiert sodass die Anzahl bekannt ist. Ist dies nicht der Fall, da sich z.B. Kunden Baukasten-artig ihr Produkt zusammenstellen können, kann es sinnvoll sein, nach Bekanntwerden der Kundenanforderung sämtliche Testfälle nochmals in der geforderten Konfiguration durchzuführen.

Obige Formel erhebt keinen Anspruch auf Vollständigkeit und in Ihrem Umfeld mag ein Faktor oder ein Summand hinzukommen oder wegfallen. Es soll lediglich verdeutlicht werden, dass es „eigentlich“ nicht ausreicht, Testfälle in nur einer Testumgebungs-Konfiguration durchzuführen. Prinzipiell sollten alle Test-Arten (also Unit-, Integration- und System-Tests) in allen möglichen Konfigurationen durchgeführt werden. Oder Würden Sie in einem Fluzeug sitzen wollen, dessen SW nur auf dem Host getestet wurde? Oder die Brems-Assistenz Ihres Fahrzeugs einem Stück SW überlassen, das in dieser Compile-Define-Kombination auf diesem Board noch nie getestet wurde?

Nicht selten wird behauptet, dass ein Testen in allen Kombinationen schlicht unmöglich sei. Durch Test-Automation und entsprechende Optimierungs-Algorithmen wird die Menge an entstehenden Testfällen aber beherrschbar und notfalls muss die Varianz beim Kunden reduziert werden. Flexibilität darf nicht zu Lasten der SW Qualität gehen.

Änderungs-basiertes Testen

Die zuletzt dargestellte Anzahl an Testfällen führt natürlich zu einem Dilemma. Wie sollen immer kürzere Produkt-Release-Zyklen realisiert werden und wie soll ein Software Entwickler „sauberen“ Code liefern wenn das Ausführen aller Testfälle 2 Monate dauert? Stand der Technik ist es zur Zeit, dass ein pragmatischer Ansatz verfolgt wird. Bei Code Änderungen werden alle Unit Tests des geänderten Files in einer Konfiguration ausgeführt bevor die Änderung in die Code-Basis eingecheckt wird. Integrations-Tests werden sporadisch, z.B. einmal pro Woche ausfgeführt und Systemtests z.B. nur alle 2 Monate. Das hat zur Folge, dass Fehler, die im Systemtest auffallen, ihre Ursache in einer Code Änderung haben die eventuell vor 2 Monaten eingecheckt wurde. neben dem zusätzlichen Aufwand der dadurch entsteht, dass zunächst der Verursacher des Fehlers eroiert werden muss, benötigt dieser dann mehr Zeit den Fehler zu korrigieren als wenn er direkt am Tage der Code Änderung auf das Problem aufmerksam gemacht worden wäre.

Ziel muss es also sein, dass jede Code Änderung möglichst zeitnah überprüft wird. Und zwar in allen Konfigurationen und in allen Test-Ebenen (Unit-, Integration-, System-Test). Um diesem Ziel näher zu kommen kann die Test-Landschaft zunächst einmal dahingehend optimiert werden, dass das Ausführen von Tests parallelisisert wird. Automatisierbare Tests können dann durch die Bereitstellung einer entsprechenden Anzahl von Test-Servern in sehr viel kürzeren Zeitabschnitten durchlaufen werden. Manuelle Tests gilt es zu reduzieren soweit wie möglich. Es ist tatsächlich nicht vertretbar, gleiche Tests in 300 verschiedenen Konfigurationen manuell durchzuführen wenn dies auch automatisiert möglich wäre.

Neben einer Parallelisierung der Testaktivitäten kann die Test-Ausführungs-Dauer zusätzlich durch eine intelligente Auswahl der Testfälle drastisch reduziert werden. Zunächst einmal ist es offensichtlich, dass von einer Source Code Änderung nur Testumgebungen betroffen sind, in denen das geänderte File vorhanden ist. Durch eine intelligente Auswahl kann die Menge an durchzuführenden Tests aber auch dadurch reduziert werden, dass nur die Testfälle innerhalb der betroffenen Testumgebungen erneut ausgeführt werden die von der Source Code Änderung betroffen sind. Insbesondere im Systemtest, der dadurch, dass üblicherweise alle Files zusammen getestet werden und der somit von jeder Code Änderung betroffen ist, kann durch eine intelligente Testfall Auswahl ein erheblicher Aufwand eingespart werden.

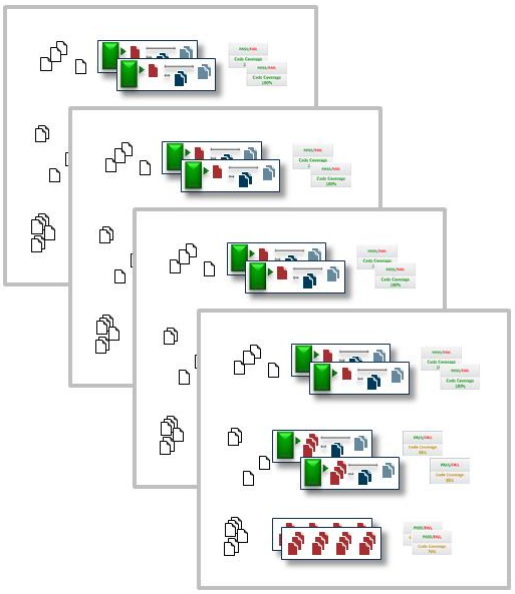


Abbildung 3 - Intelligente Auswahl betroffener Testfälle und Testumgebungen. Vgl. Abb 2.

Configuration Management und Continuous Integration

Professionelle SW Entwicklung arbeitet heute mit einem zentralen Configuration Management der Software. Zentral wird zum Beispiel auf einem Build Server eine Basis Version der Software gehalten. Entwickler können lokale Kopien der Basis erstellen, lokal Änderungen am Code durchführen und diese dann wieder in die Basis einchecken. Das zentrale Management der Software hat verschiedene Vorteile:

- Entwickler ändern in der lokalen Kopie des Codes, nicht in der Basis direkt
- Änderungs-Verfolgung: Wer hat wann was geändert
- Versions-Management der Basis SW: gib mir den SW Stand vom 1.4.2016

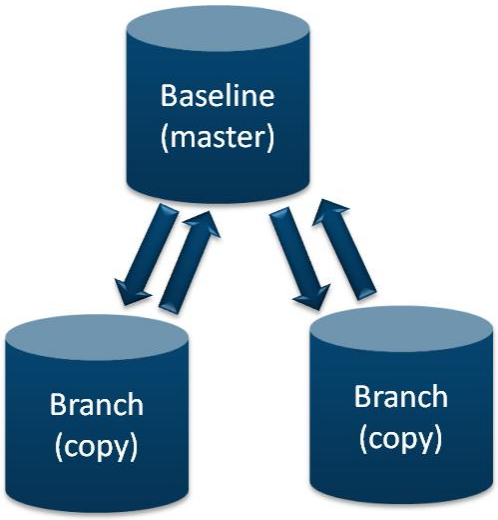


Abbildung 4 - Configuration Management mit zentraler Code Basis auf dem Build Server und lokalen Kopien

Probleme können dann entstehen, wenn das Einchecken der lokalen Kopien in zu großen Abständen erfolgt. Dies hat oft zur Folge, dass auch der Test der Code Änderungen erst sehr spät erfolgt, denn üblicherweise ist ein Software Entwickler nicht in die Lage versetzt, alle relevanten Testfälle auszuführen zu können. Darüber hinaus wird durch das verzögerte „Mergen“ mit der Basis die Wahrscheinlichkeit eines Merge-Konflikts oder sogar einer Inkompatibilität des einzucheckenden Codes größer. Eine optimierte Software Entwicklung sollte daher...

- ...jeden am Entwicklungs Prozess Beteiligten in die Lage versetzen, alle (relevanten) Tests auszuführen
- ...Merge-Zyklen der lokalen Kopien mit der Basis möglichst kurz halten

Das Ziel muss es sein, die Basis SW kontinuierlich in einem lauffähigen Zustand zu halten. Jede Code Änderung erfordert üblicherweise auch eine Anpassung der Testfall Daten sodass diese im Zuge der Code Änderung mit geändert und eingecheckt werden sollten.

Messen, Bewerten, Verbessern

Ein nicht zu unterschätzender Aspekt bei der Verbesserung eines Prozesses ist das Messen des Ist-Zustands. Leider wird dieses Messen allzuoft auch mit einem Kontrollieren der beteiligten Personen gleichgesetzt und daher häufig abgelehnt. Auch werden Metriken gerne als „Manager-Spielzeug ohne Mehrwert“ abgetan. Aber tatsächlich können Metriken auf allen Ebenen dabei helfen, Prozesse zu verbessern oder Abläufe zu optimieren. Zielsetzung der Bereitstellung von Metriken sollte also immer sein, jedem Mitarbeiter transparent die Zahlen zugänglich und deren Nutzen plausibel zu machen.

Auch die Automobil-Industrie misst der Bedeutung von Metriken eine immer größere Bedeutung zu. So hat der Arbeitskreis Softwaretest der Herstellerinitiaive Software (HIS, bestehend aus den Automobilherstellern Audi, BMW Group, DaimlerChrysler, Porsche und Volkswagen) eine Empfehlungen von relevanten Metriken (HIS-Metriken) inklusiver akzeptabler Obergrenzen der jeweiligen Werte herausgegeben[4].

Betrachten wir zunächst relevante Mess-Daten im SW Entwicklungs- und Test-Prozess:

- In Bezug auf Quell Code
    - Anzahl der Files
    - Anzahl der Funktionen
    - Anzahl Code Zeilen
    - Kommentar-Dichte
    - Compiler Warnings
    - Compiler Error
    - Zyklomatische Komplexität
    - Anzahl Funktionsaufrufe
        - Wie oft wird eine Funktion aufgerufen
        - Wieviele Funktionsaufrufe innerhalb einer Funktion

    - Anzahl der Funktionsparameter
    - Sprachumfang (Anzahl Operatoren und Operanden)
- Statische Code Analyse
    - Warnings
    - Errors
- In Bezug auf dynamische Testfälle
    - Auf allen Ebenen (Unit-, Integration-, System-Test)
        - Anzahl insgesamt
        - Anzahl durchgeführt
        - PASS/FAIL
        - Laufzeiten
- Code Coverage
    - Statement
    - Branch
    - Condition
    - MC/DC
    - function
    - function call
    - basis path
- In Bezug auf Requirements
    - Requirements/Testfall Coverage
    - Anzahl Testfall je Requirement
- In Bezug auf Code Änderungen
    - Anzahl der geänderten/hinzugefügten/gelöschten Code Zeilen
- Trends

Fragen die diese Metriken beantworten sollen sind zum Beispiel:

- Wie gut ist die Software Qualität?
- Ist meine SW "ready-to-release"?
- Wie groß ist das Risiko eines Fehlers im Code?
- Wieviel Tests wurden durchgeführt und wieviele Tests müssen noch durchgeführt werden? Wie lange wird das dauern?
- Wo muss mehr Testaufwand betrieben werden?
- Change Impact Analyse: Welche Testfälle sind von einer Code Änderung betroffen? Wie lange dauert deren Ausführung?

Beispiele für Darstellungen von Metriken

Die Kunst bei der Darstellung von Metriken ist es, die Zahlen in einer wahrnehmbaren Art zu präsentieren ohne dabei Problemstellen im statistischen Rauschen untergehen zu lassen. So ist z.B. eine durchschnittliche zyklomatische Komplexität von 3,5 je Funktion ein guter Wert. Interessant ist aber vielleicht der eine Ausreißer mit einem Wert von 158.

Um wirklich für jeden im Entwicklungsprozess interessante Informationen zu liefern ist es wichtig die Metriken interaktiv zu präsentieren. D.h. der Benutzer muss in die Lage versetzt werden, die Daten angezeigt zu bekommen, die für ihn relevant sind. Außerdem sollte es möglich sein in einzelne Bereiche der SW tiefer hineinzublicken bzw. die Sicht mehr zu globalisieren.

Abbildung 5 zeigt ein Beispiel für die Darstellung von Metriken mit sowohl Durchschnittswerten als auch Ausreißern.

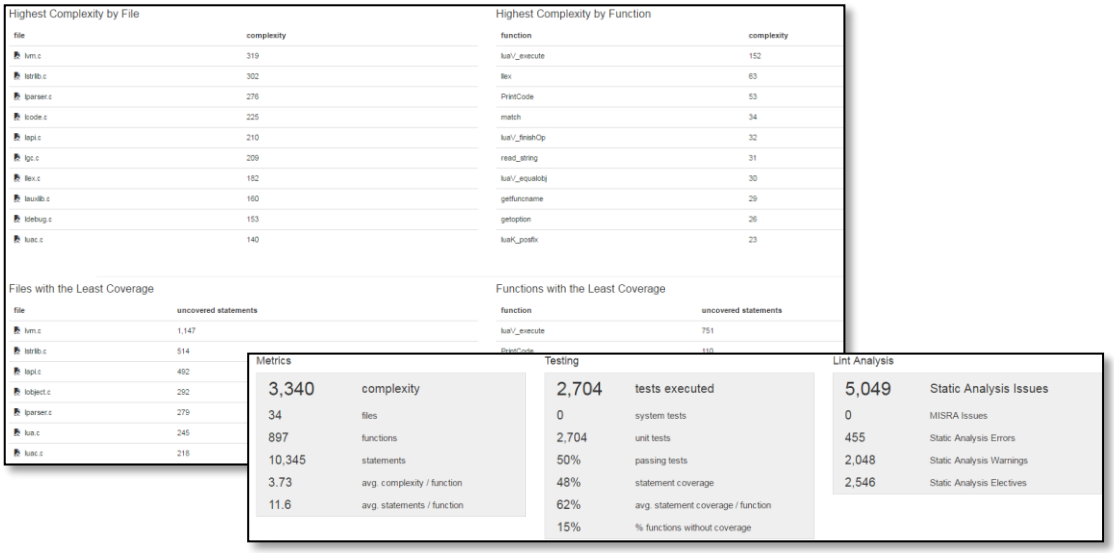


Abbildung 5 - Darstellung von Durchschnittswerten und Ausreißern

Eine zusätzliche Dimension lässt sich in der Darstellung von Metriken durch die Verwendung von Farben erzielen. Der Mehrwert wird deutlich bei der Betrachtung von Abbildung 6, in der Code Coverage, also der Anteil des Quellcodes der in dynamischen Software Tests ausgeführt wurde, farblich dargestellt ist. Die Größe des Blocks gibt eines Hinweis auf die Code Größe (auf der linken Seite) bzw. die Komplexität des Codes (rechte Seite). In diesem Beispiel befindet sich ein großer roter Block oben links. Der Block repräsentiert die Datei lvm.c, die nicht nur groß (im Sinne von „viele Statements“) sondern auch Komplex und noch dazu schlecht getestet ist. ein Hinweis für Tester, Test Manager oder Projekt Manager, dass noch ein gutes Stück Arbeit vor ihnen liegt.

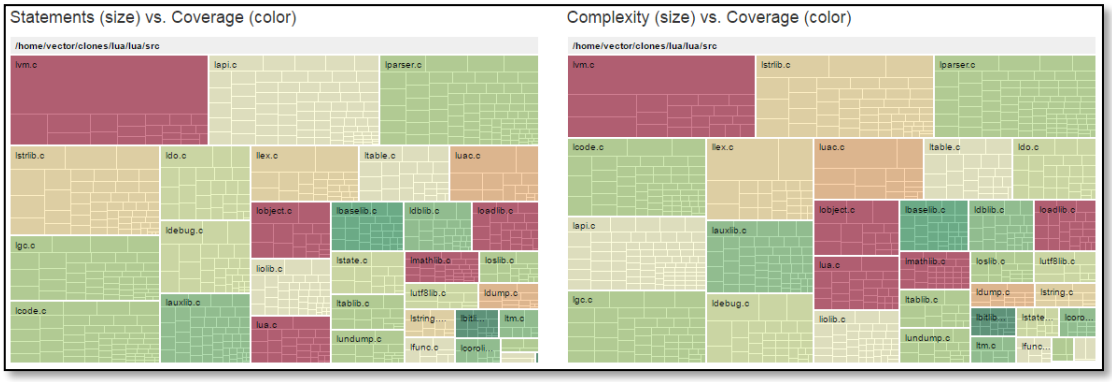


Abbildung 6 - Code Größe, Code Kompexität und Code Coverage

Die zyklomatische Komplexität der Software ist ein nicht zu unterschätzender Wert der eine nähere Betrachtung lohnt. In der Luft- und Raumfahrt wird von den Zertifizierungs-Behörden zum Beispiel eine Obergrenze von 10 festgelegt. Ebenso empfiehlt der HIS AK Softwaretest eine Obergrenze von 10 [4]. Tatsächlich zeigen Erfahrungsberichte [3], dass die Anzahl von „Abweichungen“ (unter einer Abweichung versteht man alles, was nach einem Review zu einer Änderung des Codes geführt hat, wie z.B. typos, Bugs, fehlende Kommentare, Verstöße gegen Coding Conventions...) im Code mit der zyklomatischen Komplexität steigt. Funktionen mit einer zyklomatischen Komplexität > 20 sollten generell vermieden werden, da diese sehr fehleranfällig und schlecht wartbar sind.

Abbildung 7 zeigt ein Beispiel einer Darstellung des Risikos eines Fehlers im Code, wie es sich aus der zyklomatischen Komplexität der implementierten Funktionen ergibt. Sie gibt Antworten auf die Fragen, wie groß der Anteil an sehr komplexen und damit fehlerträchtigen Funktionen ist und wie gut diese getestet sind.

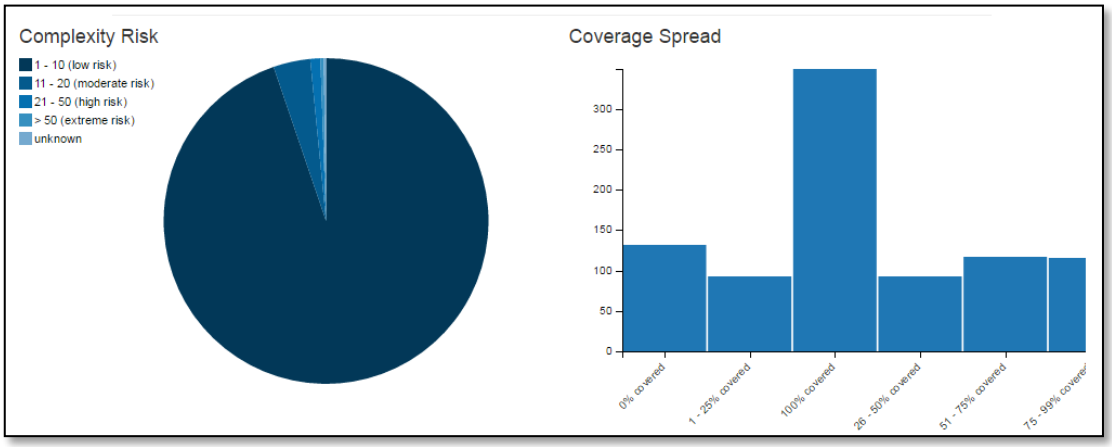
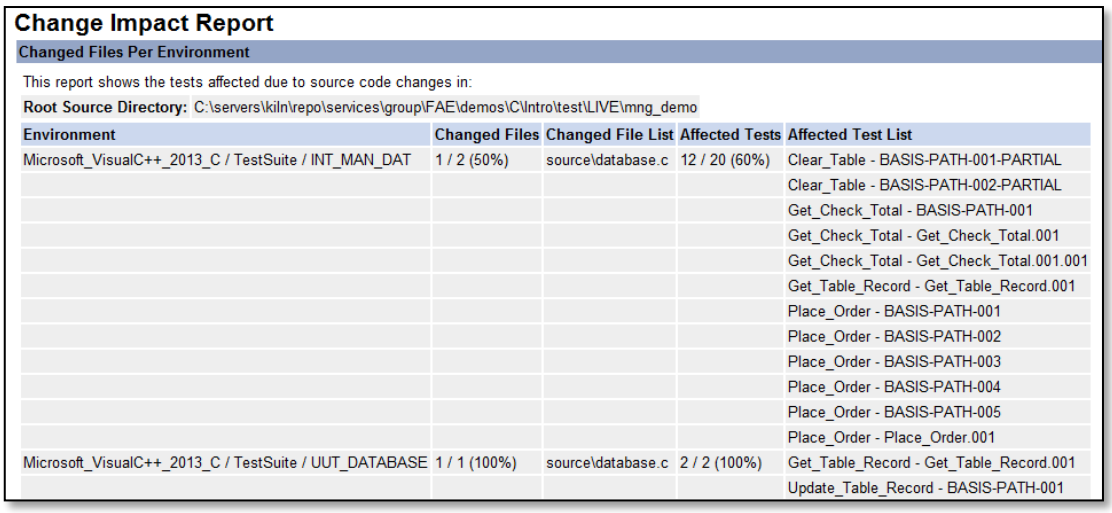


Abbildung 7 – Komplexitäts Risiko und Code Coverage

Als letztes Beispiel einer Information, die sich aus der Kombination bestehender Metriken ableiten lässt, soll noch der Change Impact Report, wie in Abbildung 8 dargestellt, erwähnt werden. Dieser ergibt sich aus dem Bewerten von Code Änderungen, Code Coverage, und Testfall Daten. Wenn man weiß, welche Testfälle von Code Änderungen betroffen sind, kann man den Aufwand abschätzen, den es bedeuten wird, eine Fehlerbeseitigung oder ein neues Release zu testen.



Change Impact Report

Changed Files Per Environment

This report shows the tests affected due to source code changes in:

Root Source Directory: C:\servers\kiln\repo\services\group\FAE\demos\C\Intro\test\LIVE\mng_demo

Environment	Changed Files	Changed File List	Affected Tests	Affected Test List
Microsoft_VisualC++_2013_C / TestSuite / INT_MAN_DAT	1 / 2 (50%)	source\database.c	12 / 20 (60%)	Clear_Table - BASIS-PATH-001-PARTIAL
				Clear_Table - BASIS-PATH-002-PARTIAL
				Get_Check_Total - BASIS-PATH-001
				Get_Check_Total - Get_Check_Total.001
				Get_Check_Total - Get_Check_Total.001.001
				Get_Table_Record - Get_Table_Record.001
				Place_Order - BASIS-PATH-001
				Place_Order - BASIS-PATH-002
				Place_Order - BASIS-PATH-003
				Place_Order - BASIS-PATH-004
				Place_Order - BASIS-PATH-005
				Place_Order - Place_Order.001
Microsoft_VisualC++_2013_C / TestSuite / UUT_DATABASE	1 / 1 (100%)	source\database.c	2 / 2 (100%)	Get_Table_Record - Get_Table_Record.001
				Update_Table_Record - BASIS-PATH-001

Abbildung 8 - Beispiel eines Change Impact Reports

Konklusion

Durch die zunehmemde Bedeutung von Software in allen Indutrie-Zweigen ergibt sich eine zunehmende Software Komplexität. Metriken können dabei helfen diese Komplexität zu beherrschen und Test-Aufwände durch Änderungs-basiertes Testen zu verringern. Viele der Kennzahlen auf die ich in diesem Beitrag eingegangen bin, liegen Ihnen vielleicht schon vor oder sie sind durch geringe Aufwände zu erhalten. Zum Dank liefern die Metriken Antworten auf einige Fragen, die allen am Prozess Beteiligten helfen effizienter zu arbeiten, die Anzahl der Software Fehler im Produkt zu minimieren und die Software qualitativ hochwertig und wart-bar zu halten.

Bei allen Arten von Messen und Bewerten muss aber auch immer darauf geachtet werden dass das Erstellen von bunten Bildern nicht zum Selbstzweck entartet. Außerdem sollten alle beteiligten Personen mit in den Bewertungs-Prozess einbezogen werden um den Beigeschmack der Kontrolle bzw. der Mitarbeiter-Beurteilung zu vermeiden.

Schließlich sollte bei aller Messerei und Bewerterei auch der gesunde Menschenverstand nicht ausgeschaltet werden. Feste Obergrenzen für einzelne Messwerte wie z.B. die zyklomatische Komplexität einzuhalten scheint zwar sinnvoll; der Prozess muss aber immer auch die Ausnahme von der Regel erlauben.

Richtig eingesetzt können Metriken dann am Ende helfen, Produkt Release Zyklen zu verringern und gleichzeitig die Produkt Qualität zu optimieren, was schließlich im Interesse aller ist: Manager, Mitarbeiter und Kunden.

Abkürzungen

Abb	Abbildung
AK	Arbeitskreis
CI	Continuous Integration
FDA	Food and Drug Administration
HIS	Herstellerinitiative Software
HW	Hardware
MC/DC	Modified Condition/Decision Coverage
RTOS	Real Time Operating System
SW	Software
vgl	vergleiche
z.B.	zum Beispiel

Referenzen

[1] General Principles of Software Validation; Final Guidance for Industry and FDA Staff, FDA, 2002

[2] James Martin, An Information Systems Manifesto, Prentice-Hall, Inc., Englewood Cliffs, New Jersey

[3] softwaretesting.vectorcast.com/acton/formfd/10305/0018:d-009d

[4] HIS Source Code Metriken des HIS AK Softwaretest

Autor

Ingo Nickles blickt auf 20 Jahre Berufserfahrung in der Softwarentwicklung für embedded Geräte für die Telekommunikationsbranche zurück. Seit 2012 ist er Field Application Engineer für die Firma Vector Software und verantwortlich für Schulungen, Präsentationen und den Support rund um die Produktfamilie VectorCAST.

Kontakt

Vector Software, Inc.
St. Töniser Str 2a, 47906 Kempen
Telefon: +49 (0)2152 8088 8082
E-Mail: ingo.nickles@vectorcast.com