

Testbarkeit im Entwicklungsprozess

Dr. Stefan Jungmayr, Teradyne Diagnostic Solutions, info@testbarkeit.de

Abstract

Um Software-Entwicklungs- und Wartungskosten unter Kontrolle halten zu können, darf Testbarkeit nicht dem Zufall überlassen werden. Bekannte Entwurfsrichtlinien (wie z.B. betreffend Kopplung und Kohäsion) alleine reichen nicht aus, um einen effizienten Testprozess zu gewährleisten. In diesem Beitrag wird dargestellt, wie von der Anforderungsermittlung bis zum Test der erforderliche Grad an Testbarkeit systematisch erreicht werden kann.

1 Testbarkeit nicht dem Zufall überlassen

Testbarkeit fasst jene Eigenschaften von Software zusammen, die für den Testaufwand bzw. den Testerfolg entscheidend sind. Zuletzt ist diese Software-Eigenschaft durch agile Software-Entwicklungsprozesse, insbesondere durch die "testgetriebene Entwicklung" [8], in den Blickpunkt gerückt. Bei testgetriebener Entwicklung wird ein Mangel an Testbarkeit (bzgl. des Entwickler- bzw. Modultests) während der Programmierung entdeckt und nach Möglichkeit durch Refactoring des Source-Codes behoben.

Testbarkeit sollte aber nicht dem Zufall bzw. den aktuellen Anforderungen des Entwicklers überlassen werden: Neben den Entwicklern gibt es noch weitere Stakeholder, die ein vitales Interesse an der Testbarkeit haben wie Tester, Benutzer und Projektmanager. Ihre funktionalen und nicht-funktionalen Anforderungen an die Software aus Sicht des Tests können und sollen *frühzeitig* bestimmt werden. Somit lässt sich Testbarkeit auch *systematisch* erzielen.

Die folgenden Abschnitte beschreiben, wie Testbarkeitsanforderungen während der Anforderungsermittlung und Analyse definiert und vom Architekturentwurf bis zum Test hin im Produkt realisiert werden können. Auch prozessbezogene Aspekte zur Erzielung der geforderten Testbarkeit werden kurz dargestellt.

2 Anforderungsermittlung und Analyse

Der Grundstein für testbare Software wird durch das Definieren von Testbarkeitsanforderungen während der Anforderungsermittlung gelegt - alle nachfolgenden Entwicklungsaktivitäten dienen dann (u.A.) der Umsetzung der Testbarkeitsanforderungen.

2.1 Testbare Anforderungen formulieren

Anforderungen müssen nicht nur korrekt, vollständig und konsistent sein, sie müssen aus Testsicht vor allem auch quantitativ formuliert werden, damit ein eindeutiger, objektiver Vergleich des tatsächlichen und des erwarteten Testergebnisses möglich ist (Testergebnis OK bzw. nicht OK). Dies gilt insbesondere für nicht-funktionale Anforderungen: Performanz kann z.B. als geforderte Reaktionszeiten im Kontext einer bestimmten Hardware-Ausstattung und allgemeinen Systemauslastung definiert werden, Benutzbarkeit kann als durchschnittliche Bewertung in einer Benutzerumfrage messbar gemacht werden.

Spätestens beim Versuch, Testfälle aus den Anforderungen abzuleiten, zeigt sich, ob die Anforderungen testbar formuliert sind oder nicht. Daher sollten Testfälle möglichst zeitnah zur Anforderungsermittlung erstellt werden.

2.2 Testbarkeitsanforderungen definieren

Definieren Sie im Rahmen der Anforderungsermittlung Anforderungen an die Testbarkeit der Software. Explizite Testbarkeitsanforderungen bieten Ihnen bezüglich der Entwicklung testbarer Software folgende Vorteile:

- *Unterstützung durch verbreitete Entwicklungsprozesse*: Volle Einbindung in Entwicklungsprozesse wie V-Modell bzw. V-Modell XT [12] und RUP [3] möglich.
- *Planbarkeit*: Eine Priorisierung ist möglich. Aufwände, die zur Sicherstellung der Testbarkeit anfallen, können geschätzt und verfolgt werden.
- *Verfolgbarkeit*: Alle Standard-Projektaktivitäten zur Kontrolle der Umsetzung von Anforderungen können auch auf die Testbarkeitsanforderungen angewendet werden.
- *Wiederholbarkeit*: in Folgeprojekten können Testbarkeitsanforderungen und darauf bezugnehmende Entwürfe und Implementierungen wiederverwendet werden, so dass die Ergebnisse besser vorhersehbar sind.
- *Effektivität*: Aus den Erfahrungen kann gelernt werden. Größere Refactoring-Schritte können vermieden werden.

Die Testbarkeitsanforderungen leiten sich direkt aus den Aufgaben der in die Testaktivitäten involvierten Personen (wie z.B. Testfalldefinition, Testausführung und Testauswertung) ab und können von Projekt zu Projekt unterschiedlich ausfallen bzw. gewichtet sein.

2.3 Beispiele für Testbarkeitsanforderungen pro Rolle

Ein **Tester** kann z.B. folgende Anforderungen an ein Software-System stellen:

- Die Komplexität des Entwurfs und der Implementierung sind dem Anwendungsproblem angemessen, nicht höher (*Adäquatheit*).
- Jede Testeinheit kann isoliert getestet werden (*Isolierbarkeit*).
- Tests und die Auswertung der Testergebnisse können ausreichend automatisiert werden (*Automatisierbarkeit*).

- Die Testeinheiten und Ressourcen können in den für den Test erforderlichen Initialzustand versetzt werden (*Kontrollierbarkeit*), unabhängig vom Entwicklungs- oder Produktivstand der Daten.

Beim Test von verteilten Systemen lässt sich der Testablauf zentral steuern.

- Die Testergebnisse und Zwischenergebnisse können beobachtet werden (*Beobachtbarkeit*), wobei die Detailliertheit der Programmausgaben (z.B. Debugging-Ausgaben) durch den Tester bestimmt werden kann.

Beim Test von verteilten Systemen können die Testergebnisse von einer zentralen Stelle aus beobachtet werden.

- Fehler lassen sich einfach dem verursachenden Software-Teil zuordnen (*Fehler-Lokalisierbarkeit*).
- Fehler überschreiten nach Möglichkeit nicht Modulgrenzen (*Fehler-Lokalität*).
- Nach dem Auftreten von Systemfehlern können die verbleibenden Tests noch ausgeführt werden (*Robustheit*).

Ebenso wichtig für den Tester sind folgende prozessbezogene Aspekte:

- Die Dokumentation der Anforderungen und der Realisierung entspricht dem aktuellen Stand (*Aktualität*).
- Die Anforderungen und der Entwurf zu einem Softwaremodul sind leicht zugänglich und nachvollziehbar (*Verfolgbarkeit*) z.B. durch gemeinsame Verwaltung in einem Versionsmanagement-Werkzeug oder Repository und Querverweise.

Werden Anforderungen geändert, so lassen sich die dadurch notwendigen Änderungen an den Testfällen rasch identifizieren.

- Die Änderungsrate der Anforderungen und der Realisierung sind gering (*Stabilität*).
- Die Anforderungen sind testbar formuliert (*Testbarkeit der Anforderungen*), d.h. der Erfolg der Tests lässt sich (mit vertretbarem Ressourcen-Einsatz) eindeutig bestimmen.
- Die Kritikalität bzw. Priorität der Anforderungen sind definiert (*Planbarkeit*).

Ein **Software-Entwickler in der Wartungsphase** kann z.B. folgende Anforderungen an das Software-System stellen:

- Ein Test hat keine oder vernachlässigbare Auswirkungen auf das Systemverhalten (*Minimal-Invasivität*), z.B. auf das Laufzeitverhalten.

Bei Systemen, die Dauerbetrieb erfordern, können Programmteile im laufenden Betrieb getestet und ausgetauscht werden.

- Der Systemzustand ist zugänglich (*Beobachtbarkeit*), d.h. der aktuelle Systemzustand kann (ohne Verfälschung der Testresultate) jederzeit abgefragt werden.

Log-Informationen können in der gewünschten Granularität erzeugt und gelesen werden.

- Der Tester kann selbstdefinierte Abfragen an das System stellen (*Diagnostizierbarkeit*).

Ein **Benutzer** kann aus Testsicht z.B. folgende Anforderungen an ein Software-System haben:

- Der Systemzustand ist zugänglich (*Beobachtbarkeit*), d.h. an den Software-Hersteller können im Fehlerfall aussagekräftige, vom System generierte Fehlermeldungen gesandt werden.
- Der aktuelle Systemzustand bzw. Fehlermeldungen sind nachvollziehbar (*Verständlichkeit*).
- Das System kann seine Integrität durch einen Selbst-Test selbst prüfen (*Diagnostizierbarkeit*).

2.4 Funktionale und nicht-funktionale Testbarkeitsanforderungen

Der Autor unterscheidet Testbarkeitsanforderungen funktionaler und nicht-funktionaler Art. Tabelle 1 zeigt eine mögliche projektspezifische Zuordnung¹ für unterschiedliche Kategorien von Testbarkeitsanforderungen.

Anforderungskategorie	funktional	nicht-funktional
Automatisierbarkeit		X
Beobachtbarkeit	X	X
Diagnostizierbarkeit	X	
Fehler-Lokalität		X
Isolierbarkeit		X
Kontrollierbarkeit	X	X
Lokalisierbarkeit		X
Minimal-Invasivität	X	X
Verständlichkeit		X

Tabelle 1: Funktionale und nicht-funktionale Testbarkeitsanforderungen

Funktionale Testbarkeitsanforderungen können Sie modellieren wie alle anderen funktionalen Anforderungen, z.B. mit Hilfe von Anwendungsfällen. Abbildung 1 enthält ein Anwendungsfalldiagramm für den Bereich Software-Wartung, wobei Testbarkeitsanforderungen bzgl. der Beobachtbarkeit des Systemzustandes als Anwendungsfall "Systemzustand abfragen" (Abbildung 2) beschrieben sind.

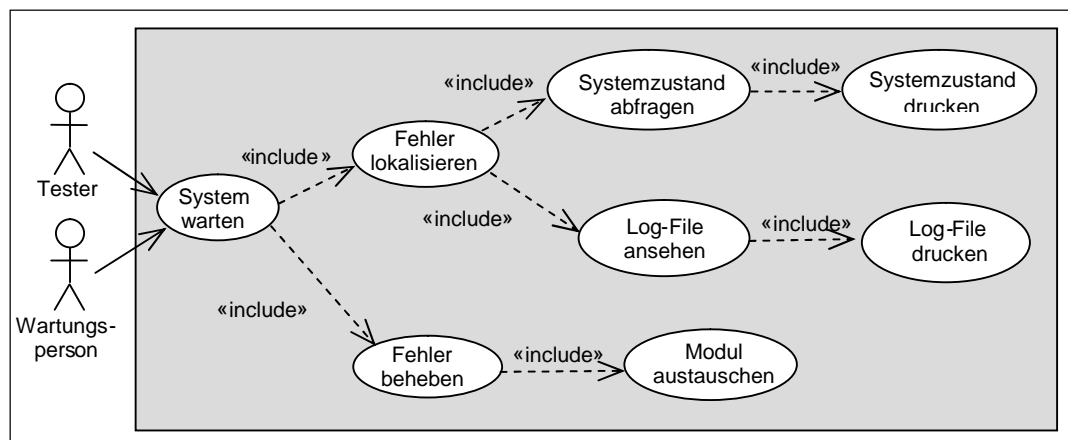


Abbildung 1: Anwendungsfalldiagramm zur Systemwartung

¹ Testbarkeitsanforderungen einer Kategorie können auf System-Ebene funktionaler Art sein (z.B. betreffend Funktionen zum Beobachten des Systemzustands) und auf Klassen-Ebene nicht-funktionaler Art (z.B. betreffend zusätzlicher Testmethoden zum Auslesen des Objektzustands).

Anwendungsfall Systemzustand abfragen**Akteur:** Tester, Wartungsperson**Ablauf:** Der Akteur fragt den Systemzustand ab. Der Systemzustand beinhaltet Ereignisse (Ein- und Ausgabe-Operationen, Fehlerereignisse) und Systemkennzahlen (CPU-Nutzung, Speichernutzung, Anzahl der Objektinstanzen, Anzahl der Threads). Die Abfrage des Systemzustands ist passwortgeschützt. Der Akteur kann die Ausgaben sortieren und filtern (nach Zeitpunkt und Kritikalität). Der Akteur kann Ereignisse und Systemkennzahlen auswählen und drucken (**include** „Systemzustand drucken“).**Vorbedingung:** keine (soweit möglich, soll der Anwendungsfall in allen möglichen Systemzuständen ablauffähig sein)**Nachbedingung:** Der aktuelle Systemzustand wurde ausgegeben.**Ausnahmefälle:** keine

Abbildung 2: Anwendungsfall „Systemzustand abfragen“

Für nicht-funktionale Testbarkeitsanforderungen sind geeignete Validierungskriterien anzugeben, wie z.B.:

- **Isolierbarkeit:** In 90% aller Klassen ist für das Testsetup die Instanziierung von Objekten aus *weniger* als 7 Klassen erforderlich.
- **Fehler-Lokalisierbarkeit:** Durchschnittlich sind weniger als 30 Minuten erforderlich, um einen Fehler zu isolieren.

Wie andere Anforderungen auch, so sollen Testbarkeitsanforderung priorisiert werden: dadurch können die richtigen Trade-Offs gegenüber anderen funktionalen und nicht-funktionalen Anforderungen getroffen werden.

2.5 Komplexität der Anwendungsfälle reduzieren

Abhängigkeiten zwischen Anwendungsfällen entstehen u.A. durch Vor- und Nachbedingungen der Anwendungsfälle, durch include- und extend-Beziehungen, durch Zugriff auf gemeinsame Objekte (d.h. Klassen) der Domäne oder durch zeitliche Abhängigkeiten. Diese Abhängigkeiten wirken sich direkt auf Abhängigkeiten zwischen Testfällen aus und sollten daher minimiert werden, z.B. durch Schwächung der Vorbedingung von Anwendungsfällen und durch Schwächen von Multiplizitäts-Beziehungen zwischen den Klassen der Anwendungsdomäne.

2.6 Schwer testbare Anforderungen identifizieren

Manche Anforderungen sind inhärent schwer testbar aufgrund ihrer Komplexität und fehlenden Möglichkeit zur automatisierten Testauswertung, wie z.B. komplexe Berechnungen (Simulationen, NP- harte Probleme, Berechnungen ohne inverse Funktion), komplexe graphische Operationen sowie Funktionen mit einem großen Eingabe- und Ausgaberaum. Hier sind gezielte Maßnahmen zur Verbesserung der Testbarkeit erforderlich wie z.B.:

- Re-Design der Anforderungen mit dem Kunden,
- gezielte Verbesserung der Beobachtbarkeit von Zwischenergebnissen,

- andere oder zusätzliche Repräsentationsformen der (Zwischen-)Ergebnisse vorsehen, welche die Auswertung der Testergebnisse vereinfachen.

2.7 Testkritische Domänenklassen identifizieren

Identifizieren Sie jene Klassen, die zuständig sind für

- schwer testbare Anforderungen oder
- Funktionen, die den Testverlauf stark verlangsamen oder beeinträchtigen wie z.B. den Zugriff auf eine Datenbank oder das Internet.

Diese Klassen müssen im Entwurf besonders für den Test in Isolation vorbereitet werden.

3 Architekturentwurf

Im Architekturentwurf wird die Grundlage für die Testbarkeit einer Anwendung – insbesondere bzgl. der nicht-funktionalen Testbarkeitsanforderungen – gelegt. Testbarkeitsmängel in der Architektur sind naturgemäß besonders aufwendig zu beheben. Daher sollte auch ein Prototyp oder Durchstich² genutzt werden, um grobe Testbarkeitsmängel aufzudecken. Dieser Abschnitt beschreibt Maßnahmen zur Umsetzung der Testbarkeit auf Architektur-Ebene.

3.1 Komplexität reduzieren

Vermeiden Sie Komplexität, die zur Bewältigung des Anwendungsproblems nicht erforderlich ist. Vermeidbare Komplexität kann insbesondere in den folgenden Entwurfselementen stecken:

- bidirektionale, zyklische und nicht-hierarchische Navigationsstrukturen in der GUI-Oberfläche, nicht-modale Fenster sowie redundante Fenster-Inhalte (d.h. Inhalte, die in mehreren Fenstern gleichzeitig angezeigt werden)
- Zustandsverhalten von Komponenten,
- hohe Anzahl von möglichen Systemkonfigurationen,
- dynamische Abhängigkeiten zwischen Komponenten, die sich statisch nicht vorhersagen lassen,
- dynamische Konfiguration von Komponenten,
- Parallelität,
- ereignisgesteuerte Interaktion: die Reihenfolge der Ereignisse (Events) ist nicht deterministisch; viele unterschiedliche Ereignisreihenfolgen sind möglich (Anm.: Abhilfen sind hier z.B. Queues und Clocking),
- nicht-deterministisches Verhalten,
- Verteiltheit der Anwendung,
- Einsatz von heterogenen Implementierungs-Technologien.

Während jedes Entwurfselement einzeln betrachtet durchaus seine Motivation haben kann, so sollte es doch aus dem Blickwinkel der kombinatorischen

² Ein Durchstich ist eine Implementierung einer Teilfunktionalität der Anwendung, die z.B. von der GUI bis zur Datenhaltungsschicht alle Architekturschichten betrifft.

Explosion der Testfallanzahl (insbesondere bei Kombination mit anderen Entwurfselementen) hinterfragt werden.

3.2 Austauschbarkeit sicherstellen

Während der Testvorbereitung, Testdurchführung sowie insbesondere bei der Fehlersuche ist es für den Testaufwand sehr entscheidend, dass Komponenten isoliert getestet werden können bzw. am Test direkt oder indirekt beteiligte Dienstleisterklassen (der getesteten Klasse) für Testzwecke austauschbar sind. Folgende Entwurfsrichtlinien tragen dazu bei:

- Schichtenarchitektur bzw. n-Tier-Architektur nutzen.
- Architekturformen bevorzugen, welche eine einfache Austauschbarkeit von Komponenten durch lose Kopplung ermöglichen (z.B. Filter-Architektur).
- Zyklische Abhängigkeiten zwischen Moduln unterschiedlicher Software-Schichten vermeiden.
- Abhängigkeiten einer Komponente von außerhalb setzen lassen ("Dependency Injection") [11]. So können die Dienstleisterklassen vom Tester bei Bedarf einfach durch Stubs oder Mock-Versionen ersetzt werden.
- Objekt-Fabriken einsetzen, welche u.A. Mock-Versionen von Komponenten erzeugen können. Dies zentralisiert und vereinfacht den Austausch von Dienstleisterklassen.
- Komposition bevorzugen und Vererbung kontrolliert einsetzen.
- Die feste Verdrahtung von Ressourcen z.B. durch das Singleton- Muster vermeiden.

3.3 Verantwortlichkeiten trennen

Wenn eine Komponente eine eingrenzte Verantwortlichkeit hat, so vereinfacht dies die Testfalldefinition und Testfallwartung. Die Trennung der Verantwortlichkeiten betrifft u.A. die Trennung von GUI und Anwendungslogik, die Trennung von Fachlichkeit und (Web-)Technologie, sowie die Trennung der Validierung und Verarbeitung von Eingabeparametern.

3.4 Beobachtbarkeit sicherstellen

Bestimmte Architekturformen vereinfachen das Beobachten von Zwischenergebnissen an den Komponentenschnittstellen (wie z.B. Filter-Architekturen oder ereignisgesteuerte Architekturen). Dabei muss aber darauf geachtet werden, dass durch die zusätzliche Entwurfskomplexität dieser positive Effekt nicht wieder zunichte gemacht wird.

3.5 Test-Funktionalität vorsehen

Klassen bzw. Komponenten können zur Vereinfachung des Tests Testschnittstellen (d.h. zusätzliche Methoden, die nur dem Test dienen) bereitstellen. Dabei ist technisch oder prozess-technisch sicherzustellen, dass die Testschnittstellen nicht zur eigentlichen Programmierung verwendet werden können.

Insbesondere bei Frameworks muss die Testbarkeit der in das Framework zu integrierenden Klassen oder Software-Module (Plug-Ins) sichergestellt werden.

3.6 Automatisierbarkeit sicherstellen

Wählen Sie eine Plattform bzw. ein Betriebssystem, das durch Testwerkzeuge gut unterstützt wird. Sollen Capture-and-Replay-Testwerkzeuge eingesetzt werden, so ist darauf zu achten, dass die verwendeten (ggf. Nicht-standard-) GUI-Elemente für die Nutzung dieser Testwerkzeuge kein Problem darstellen.

Command-Objects [1] erlauben das einfache Aufzeichnen und Abspielen von Benutzer-Interaktion unter Umgehung der GUI, was die Testausführung wesentlich beschleunigt und die Nutzung von Capture-and-Replay-Testwerkzeugen vermeiden hilft.

Eine strikte Trennung von GUI- und Anwendungsschicht (vergl. Kapitel 3.3) ermöglicht es, die Anwendungslogik direkt über eine XML- oder WSDL-Schnittstelle im Batchmodus zu testen.

3.7 Fehler-Isolierbarkeit verbessern

Wählen Sie eine Programmiersprache, welche bestimmte Fehlerarten (z.B. betreffend der Pointer-Arithmetik) von vornherein vermeiden hilft sowie das Isolieren von Fehlern durch gute Compiler-Meldungen oder Zusicherungen (Assertions) unterstützt.

Vermeiden Sie Re-Entrance-Situationen [6], die das Code-Verständnis sehr erschweren.

Vermeiden Sie zu tiefe Vererbungshierarchien mit Yo-Yo-Effekt (d.h. Methodenaufrufreihenfolgen, die in der Vererbungshierarchie wiederholt hin- und herspringen).

4 Komponententwurf

Achten Sie im Komponententwurf darauf, dass die konkreten Schnittstellen und Abhängigkeiten zwischen Komponenten keinen negativen Einfluss auf die Testbarkeit haben. Hohe Schnittstellen-Komplexität sowie starke Abhängigkeiten durch tiefe Vererbungshierarchien und Objektzyklen sollten vermieden werden. Das "Law of Demeter" [7] hilft, die Abhängigkeiten einer einzelnen Klasse zu reduzieren bzw. zu kontrollieren.

Identifizieren Sie testkritische Abhängigkeiten, d.h. Abhängigkeiten zu testkritischen Klassen (siehe Kapitel 2.7) bzw. jene Abhängigkeiten, welche die Gesamtabhängigkeit im System stark erhöhen. Diese testkritischen Abhängigkeiten sollten besonders sorgfältig entworfen werden.

5 Implementierung

Während der Implementierung sind u.A. folgende Faktoren zu beachten:

- *Komplexität*: Umfangreiche Schnittstellen mit vielen Parametern, tiefe Klassenhierarchien sowie lange Kaskaden von Methodenaufrufen erschweren die Testfalldefinition, Testvorbereitung und Fehlersuche.
- *Verständlichkeit*: Vermeiden Sie vermeintlich elegante, trickreiche Lösungen, Rekursion, komplexe Algorithmen und implizite Kontrolllogik. Erst ein

verständlicher und testbarer Code sollte bei Bedarf und existierenden Testfällen z.B. auf Performanz hin optimiert werden.

- *Kontrollierbarkeit*: Komplexe Schleifenkonstrukte, unerreichbare Ausgabe-werte und unerreichbare Pfade erschweren die Bestimmung der Testdaten, die zur Erzielung eines bestimmten Kontrollflusses (bei White-Box-Tests) erforderlich sind.
- *Beobachtbarkeit*: Ein nur implizit repräsentierter Objektzustand, der Verlust von Zwischenergebnissen oder ein versteckter Kontrollfluss erschweren die Beobachtung von Testergebnissen und die Kontrolle der Testabdeckung.
- *Wiederholbarkeit*: Ein impliziter Input (z.B. undokumentierte zeitliche Abhängigkeit oder undokumentierte Abhängigkeit zu einer globalen Variable), der nicht in den Testfällen berücksichtigt ist, verringert die Wiederholbarkeit von Testergebnissen.
- *Abhängigkeiten*: Insbesondere zyklische Abhängigkeiten erschweren den Test in Isolation.
- *Automatisierbarkeit*: Die Verwendung von Nicht-Standard-Programmkonstrukten sowie nichtverfügbare Namen von GUI-Elementen erschweren den Einsatz von Capture-and-Replay-Testwerkzeugen.

Die laufende Bewertung der erzielten Testbarkeit mit Hilfe von Metriken, Sensitivitäts-Analysen und Reviews gewährleistet die frühzeitige Entdeckung und Behebung von Testbarkeitsmängeln während der Implementierung.

5.1 Metriken

Die derzeit existierenden Testbarkeitsmetriken fokussieren hauptsächlich auf die Komplexität sowie die Abhängigkeiten von Software-Moduln.

Die älteste Testbarkeitsmetrik ist die zyklomatische Komplexität von McCabe, welche die Anzahl der unabhängigen Pfade durch ein Modul misst und damit die Anzahl der Testfälle die notwendig sind, um das entsprechende Testkriterium zu erfüllen. Ein umfangreiches Set von Komplexitätsmetriken zur Testbarkeitsbewertung wird von Harry Sneed und dem Autor dieses Artikels in [10] beschrieben.

Metriken, welche die Abhängigkeiten zwischen Software-Moduln messen, tragen zur Bewertung des Testaufwands und der Möglichkeit zum Test in Isolation bei. Zu den Abhängigkeitsmetriken gehört die Metrik ACD von Lakos [6], die Metriken Abstractness und Stability von Martin, die im Standard ISO 9126 definierten Testbarkeitsmetriken [1], sowie Metriken des Autors [5].

5.2 Sensitivitätsanalyse

Sensitivitätsanalyse ermöglicht es, jene Codeteile zu identifizieren, welche besonders stark z.B. zur Gesamtanzahl der indirekten Abhängigkeiten in einem Software-Projekt beitragen [5]. Diese Codeteile sollten genauer untersucht und ggf. refaktoriert werden.

5.3 Reviews

Prüfen Sie die Umsetzung der Testbarkeitsanforderungen beim Review der Architektur- und Entwurfsdokumente. Erstellen Sie dazu Checklisten-Einträge

auf Basis der wichtigsten Testbarkeitskriterien bzw. gefundenen Testbarkeitsprobleme [4].

6 Test

Testprobleme, die während der Testaktivität zu Tage treten, sollten dokumentiert und behoben werden. Wichtige Vorbedingungen für die Behebung der Testprobleme sind allerdings:

- Die verfügbaren Ressourcen lassen ein Refactoring zu.
- Eine Refaktorisierung ist technisch bzw. praktisch möglich (ohne große Teile der Anwendung neu programmieren zu müssen).
- Der Code ist gut strukturiert und änderungsfreundlich gestaltet.
- Eine Test-Suite steht bereit, um zu prüfen, dass die Refaktorisierung die Funktionalität nicht beeinträchtigt hat.

Teilweise bedingen sich diese Faktoren gegenseitig.

7 Entwicklungsprozess

Neben produktbezogenen Maßnahmen sind auch prozessbezogene Maßnahmen für die Umsetzung von Testbarkeitsanforderungen wichtig.

- Definieren Sie die Verantwortlichkeiten für die Planung und Umsetzung von Testbarkeitsmaßnahmen.
- Nutzen Sie möglichst eine inkrementelle Entwicklung und "daily builds". Diese helfen, Testprobleme frühzeitig zu entdecken (vergleiche testgetriebene Entwicklung).
- Erfassen und behandeln Sie Testprobleme wie alle anderen Produktfehler. Ergänzen Sie bei einer vorhandenen Fehlerdatenbank die Fehlerkategorie "Testproblem".
- Berücksichtigen Sie Testbarkeit explizit in Entwicklungsdokumenten wie z.B. Anforderungs- und Entwurfsdokumenten, Entwurfsrichtlinien, Qualitäts- und Testreports sowie Checklisten.
- Schulen Sie Entwickler (aber auch Tester) zum Thema testfreundliches Design und den konstruktiven Möglichkeiten der Testbarkeitsverbesserung.

8 Schnell-Start zu mehr Testbarkeit

Der in diesem Artikel vorgestellte Ansatz zur Entwicklung testbarer Software erfordert Know-how bei Projektleitern, Entwicklern und Testern:

- Tester müssen wissen, welche konstruktiven Möglichkeiten das Testen erleichtert und welche Testbarkeitsanforderungen daher definiert werden sollen.
- Entwickler müssen wissen, wie Testbarkeitsanforderungen realisiert werden können.

- Projektleiter müssen die richtigen Trade-Offs zwischen Testkosten und Entwicklungskosten auf Basis von Testbarkeitsanforderungen finden können.

Die explizite Verfolgung von Testbarkeitsmängeln als eine Fehlerkategorie einer Fehlerdatenbank ermöglicht, bei geringen Kosten das Problembewusstsein zu stärken. In bereits laufenden Projekten kann eine Task-Force Verbesserungsmaßnahmen für die dringendsten Testbarkeitsmängel erarbeiten. Bei erst anlaufenden Projekten stellt die Definition von Testbarkeitsanforderungen das Mittel der Wahl dar, die durch Vorkenntnis über potentielle Testmängel und konstruktive Möglichkeiten unterstützt wird.

9 Fazit

Software-Entwicklung wird auch weiterhin in vielen Unternehmen ohne explizite Berücksichtigung von Testbarkeitsanforderungen stattfinden. Qualität und zeitgerechte Lieferung gewinnen aber für den Markterfolg immer mehr an Bedeutung. Daher werden jene, die Testbarkeits-Technologien beherrschen und anwenden, einen wesentlichen Wettbewerbsvorteil für sich verbuchen können. Sie sollten dazu gehören.

Hinweis

Weitere Informationen und Artikel zum Thema Testbarkeit finden Sie auf der Website <http://www.testbarkeit.de>.

Danksagung

Vielen Dank an Michael Averstegge, Michael Brunner, Jens R. Calame, Harry Sneed und Prof. Dr. Mario Winter für Ihre Reviews von Draft-Versionen dieses Artikels.

10 Referenzen

- [1] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1994.
- [2] Standard *ISO/IEC 9126-2. Software product quality - external metrics*.
- [3] Ivar Jacobson, Grady Booch, James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999. ISBN 0201571692.
- [4] Stefan Jungmayr. *Reviewing Software Artifacts for Testability*. Presentation bei der EuroSTAR99, Barcelona, Spanien, 8.-12. November 1999.
- [5] Stefan Jungmayr. *Improving testability of object oriented systems*. dissertation.de, 2004. ISBN 3-89825-781-9. URL: <http://www.dissertation.de/FDP/sj929.pdf>
- [6] John Lakos. *Large-scale C++ software design*. Addison-Wesley, 1996. ISBN 0201633620.
- [7] Karl J. Lieberherr und I. Holland. Assuring good style for object-oriented programs. *IEEE Software*, September 1989, S. 38-48.
- [8] Johannes Link. *Softwaretests mit JUnit: Techniken der testgetriebenen Entwicklung*. 2. Auflage. dpunkt.verlag, 2005.
- [9] Robert C. Martin. *Stability. C++ Report*, Jan. 1997.

- [10] Harry Sneed und Stefan Jungmayr. Produkt- und Prozessmetriken für den Software Test. Beitrag eingereicht bei *Informatik Spektrum*.
- [11] URL: <http://www.martinfowler.com/articles/injection.html>
- [12] URL: <http://www.v-modell.iabg.de>