UNIVERSITÄT DORTMUND
Fachbereich Informatik

Seminar zur Softwaretechnologie
"Testen und Sicherheit bei der komponentenbasierten Softwareentwicklung"
Prof. Dr. Volker Gruhn, Sami Beydeda
WS 2000/2001

"Anforderungen an das Testen von komponentenbasierten Softwaresystemen"

Verfasser:
Tim Schürmann
Dortmund, Januar 2001
E-Mail: tischuer@yahoo.de
Homepage: http://www.timshome.purespace.de

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

1.1. Hintergrund

Heutzutage gewinnt der Einsatz von Komponenten in der Softwareentwicklung immer mehr an Zuspruch. Gerade die Entwicklung großer Softwaresysteme, die sehr hohe Anforderungen an die Zuverlässigkeit und die Verfügbarkeit stellen, zogen bislang enorme Kosten nach sich (vgl. [Weyuker98]). Viele Organisationen haben deshalb begonnen, solche Systeme unter Verwendung von wiederverwendbaren Komponenten-Repositorien zu implementieren. Oft wird dabei auf sog. COTS ("Commercial of the Shelf", dt: "kommerzielle Komponenten aus dem Regal", auch "Components of the shelf") Komponenten zurückgegriffen (vgl. [GoMa99]). Der Einsatz dieser COTS-Komponenten oder Komponenten, die zur Wiederverwendung entworfen wurden, soll signifikant niedrigere Entwicklungskosten und kürzere Entwicklungszyklen zur Folge haben. Eine Verwendung solcher Komponenten soll unmittelbar zu Softwaresystemen führen, die weniger Zeit bei der Spezifizierung, beim Entwurf, beim Test und beim Support benötigen und dabei gleichzeitig die an sie gestellten, hohen Zuverlässigkeitsanforderungen erfüllen (vgl. [Weyuker98]). Nach [Weyuker98] wünschen sich viele Organisationen, dass sich "Repositorien-Komponenten" (d.h. Komponenten aus einem Repositorium) im Idealfall nahtlos in eine neue Umgebung integrieren lassen und COTS ("commercial of the shelf")-Komponenten ein "plug-and-play" Verhalten aufweisen. Hierdurch sollen die Komponenten einfach in ein Softwaresystem integrierbar sein, um dort ihre speziellen Fähigkeiten anderen Komponenten bereitszustellen.
Um die potentiellen Vorteile einer solchen Architektur realisieren zu können, müssen beim Bau von Systemen zuverlässige Komponenten verwendet werden, die untereinander sicher agieren und operieren. Damit dieses Ziel erreicht werden kann, sind bei der Entwicklung von einzelnen Komponenten umfassende Tests unabdingbar. Dies gilt sowohl für jede einzelne Komponente, als auch für die daraus entstehenden Systeme.
Nach Gosh et al.[GoMa99] stellt das Testen von Softwaresystemen ein komplexes Problem dar, das im Fall von verteilten Systemen sogar noch größer wird, da komponentenbasierte Systeme mit ihrer Größe und der Anzahl ihrer Komponenten wachsen. Erschwerend kommt oft hinzu, dass die zur Wiederverwendung bestimmten Komponenten plötzlich in einem System mit vollkommen anderen Spezifikationen eingesetzt werden sollen oder dass unterschiedliche Middleware, wie CORBA oder Java RMI beim Bau eines neuen Systems verwendet werden.

Nach Harrold et al. [HaLiSi] bilden Softwaretests und der Support zwei Drittel aller Kosten der Softwareproduktion. Softwaretools, die Programmanalysetechniken verwenden, könnten die Test- und Supportaufgaben automatisieren, somit die Kosten für diese Aufgaben reduzieren und gleichzeitig die Softwarequalität erhöhen. Durch den hohen Grad der Wiederverwendbarkeit einer Komponente, empfehlen Harrold et al.[HaLiSi] darüber hinaus, Techniken zur Qualitätssicherung heranzuziehen. Man benötigt also effiziente und effektive Wege, um diese komponentenbasierten Systeme zu testen, zu unterstützen und die Qualität der Komponenten sicher zu stellen.
Nach Gao [Gao00] existieren zum heutigen Zeitpunkt zwar viele Arbeiten, die sich mit dem Problem der Erstellung und der Entwicklung von komponentenbasierter Software auseinandersetzten, allerdings befassen sich nur wenige mit dem Thema des Testens von Komponenten und komponentenbasierter Software. Durch die Vorteile, die die Softwarekomponententechnologie mitbringt, wurde in der letzten Zeit immer deutlicher, dass die Qualität von komponentenbasierter Software von der Qualität der einzelnen Komponenten und der Effizienz des Softwaretestprozesses abhängt (vgl. [Gao00]).

Ein extremes Beispiel, warum Testen von Komponenten bei der Entwicklung von komponentenbasierenden Softwaresystemen äußerst wichtig ist, liefert Weyuker [Weyuker98]:
Beim Jungfernflug der Ariane 5 Rakete wurden Softwarekomponenten aus der Ariane 4 verwendet. Der mangelhafte Test dieser wiederverwendeten Software führte laut ESA zur Explosion der Rakete nach ihrem Start. Hieraus wird deutlich, dass bei einem alltäglichen Gebrauch von wiederverwendbaren Softwarekomponenten, insbesondere COTS-Komponenten, Testmethoden benötigt werden, die solche "Unglücke" verhindern können. Darüber hinaus muss auf eine kosteneffektive Weise sichergestellt werden, dass das resultierende System zuverlässig die angesteuerten Ziele erreicht.

Diese Seminararbeit soll sich mit den auftauchenden Problemen und den daraus ergebenden Anforderungen an das Testen von komponentenbasierenden Systemen auseinander setzen. Dazu werden in Kapitel 2 zunächst die auftretenden Probleme und Anforderungen beim Testen einer einzelnen Komponente und in Kapitel 3 die Probleme und Anforderungen beim Testen des gesamten, integrierten Systems erläutert. Danach soll in Kapitel 4 auf die Probleme und Anforderungen an die Testsysteme und Testsuiten eingegangen werden. Abschließend wird in Kapitel 5 ein Fazit gezogen.

1.2. Begriffe

Ein komponentenbasiertes System setzt sich aus Komponenten zusammen. Nach Harrold et al. [HaLiSi] sind dies Module, die sowohl Daten, als auch Funktionalitäten einkapseln und zur Laufzeit durch Parameter konfigurierbar sind.
Nach Gosh et al. [GoMa99] kann eine Komponente in einem verteilten System ein Client, ein Server oder beides sein. Jeder Server bietet demnach ein Interface ("Schnittstelle"), das normalerweise aus ein oder mehreren Methodensignaturen ("method signatures") besteht. Diese Methoden können auf dem spezifizierten System von Clients aktiviert werden. Eine Methodensignatur ("method signature") spezifiziert dabei den Namen und die Parameter, die zwischen Client und Server ausgetauscht werden, sobald die Methode aufgerufen wird. Darüber hinaus wird dort ein Rückgabewert und keine oder mehrere Ausnahmen, die während der Ausführung zurückgegeben werden können, festgelegt. Jede Komponente ist in einer oder mehrerer Programmiersprachen geschrieben. Da die Komponenten auf Maschinen in einem Netzwerk verteilt sein können, wird von ihnen weiter angenommen, dass sie nicht alle die gleiche Laufzeitumgebung aufweisen.
Ein System heißt dynamisch, wenn sich nach dem Start und während der Laufzeit, die Anzahl der Komponenten verändert. Ein System, bei dem sich diese Zahl nicht ändert, heißt statisch (vgl. [GoMa99]).

Die Kriterien, die im Zusammenhang mit der Analyse und dem Testen von komponentenbasierten Systemen auftauchen, können nach Harrold et al. [HaLiSi] aus zwei Perspektiven betrachtet werden: die Komponentenanbietersicht und die Komponentenbenutzersicht. Der Komponentenanbieter betrachtet Analyse- und Testkriterien, die im Interesse des Anbieters von Softwarekomponenten liegen, wo hingegen die Komponentenbenutzersicht, Analyse- und Testkriterien betrachtet, die den Benutzer betreffen. Der Anbieter sieht die Komponenten unabhängig vom Kontext, in dem die Komponenten verwendet werden. Daher muss der Anbieter alle Konfigurationen einer Komponente in einer kontextunabhängigen Weise testen. Im Gegensatz dazu betrachtet der Benutzer die Komponenten als kontextbezogene Einheiten, da die Komponentenapplikation des Benutzers den Kontext bietet, in dem die Komponenten verwendet werden. Der Benutzer

kommt nur mit solchen Konfigurationen oder Verhaltensaspekten in Berührung, die für seine Anwendung relevant sind.
Gao [Gao00] unterteilt ein komponentenbasiertes Softwaresystem weiter in vier verschiedenartige Gruppen von Komponenten. Die erste Gruppe enthält kommerzielle Komponenten von Drittherstellern. Die zweite Gruppe enthält sog. "in-house" Komponenten (Komponenten die im eigenen Haus entwickelt wurden) für andere Projekte, die in verschiedenen anderen, eigenen Projekten wiederverwendet werden. Die dritte Gruppe besteht aus "in-house" Komponenten, die für eine Wiederverwendung in einem neuen Programm oder Projekt aktualisiert wurden. Die letzte Gruppe besteht aus der Menge der Komponenten, die für das Projekt neu konstruiert wurden.

Weyuker [Weyuker98] schlägt vor, dass komponentenbasierte Softwaresysteme mindestens drei Testphasen durchlaufen sollten: “unit-tests” ("Einheiten-Tests"), in denen die einzelnen Komponenten getestet werden, Integrationstests, in denen Subsysteme gebildet werden und dabei die individuell getesteten Komponenten als Einheiten ansehen und schließlich der Systemtest, in dem das System aus den getesteten Subsystemen als eine Einheit geformt und getestet wird. Darüber hinaus gibt es noch weitere Testmöglichkeiten, die von der Granularität der Softwareeinheiten abhängig sind. Ein “Einheiten-Test” (“unit-testing”) verwendet die Testfälle, die durch die Betrachtung des aktuellen Codes erzeugt wurden (“white-box test”). Im Gegensatz dazu verwenden Systemtests Testfälle ohne Referenz auf den Sourcecode (sog. “Blackbox” oder “spezifikationsbasierte Tests”). Selbst wenn der Sourcecode vorliegen würde, würde in einem solchen Fall zu viel Code existieren, um ihn durch eine der bekannten Methoden zu testen. Daher wird in einem solchen Fall nur das komplette System getestet. Integrationstests können sowohl Entwickler, als auch unabhängige Organisationen durchführen.

2. Probleme und Anforderungen beim Testen von Komponenten

Die Qualität eines komponentenbasierten Programms hängt nach Gao [Gao00] von der Qualität der einzelnen Komponenten, sowie der Effizienz der "Unit-", "Integrations-" und "Systemtests" ab. Daher ist es notwendig, sich zunächst mit den Problemen zu beschäftigen, die beim Testen einer einzelnen Komponente auftreten. Aus diesen Problemen resultieren dann zwangsläufig die Anforderungen an das Testen einer Komponente. Die in diesem Kapitel aufgeführten Probleme und Anforderungen betreffen zu einem großen Teil den Hersteller einer Komponente.
Die im Folgenden verwendete Einteilung ist dabei nicht immer eindeutig bzw. vollständig, da insbesondere die Probleme, die sich im Zusammenhang mit dem Testen von komponentenbasierender Software ergeben, erst nach und nach bekannt werden. Die hier verwendete Gliederung ist angelehnt an [GoMa99] und [Gao00].

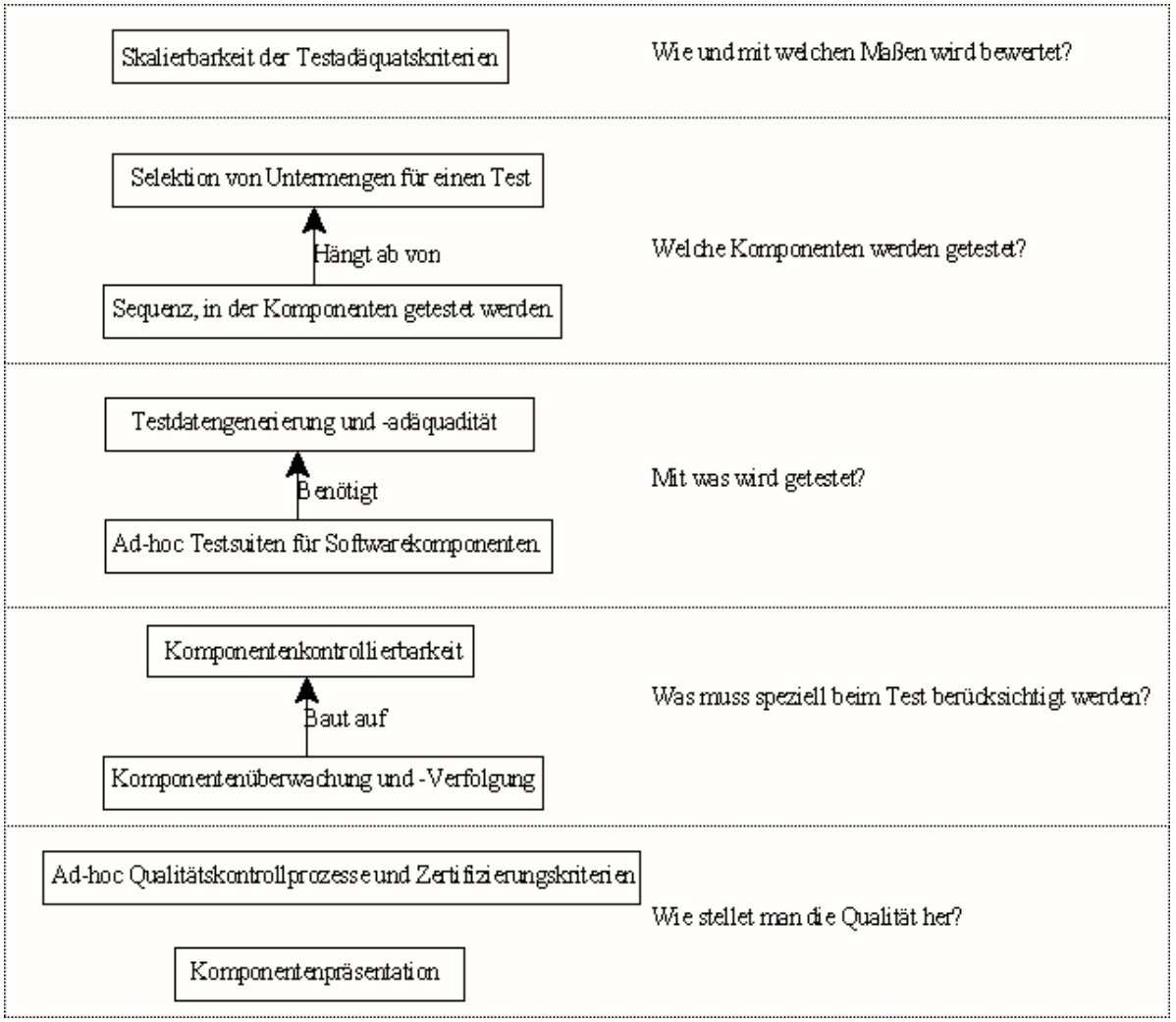

Abbildung 1 Übersicht über die in Kapitel 2 genannten Kriterien

2.1. Skalierbarkeit der Testadäquaditätskriterien

Ist der Sourcecode einer Komponente verfügbar, können nach Gosh et al. [GoMa99] herkömmliche Testkriterien verwendet werden, wie z.B. Kontrollfluss-, Datenfluss- und Mutationsverfahren. Datenfluss basierende Testkriterien wurden bereits auf die Klassen in der objekt-orientierten Programmierung erweitert. Die dort auftauchenden Probleme betreffen komponentenbasierte Systeme in zweierlei Hinsicht. Zum einen können die Komponenten selbst in einer objekt-orientieren Sprache geschrieben worden sein und zum anderen lassen sich einige objekt-orientierte Konzepte auch auf komponentenbasierende Systeme übertragen.
So können beim Testen von Komponenten verschiedene Ausgaben entstehen. In einem solchen Fall reicht die Behandlung von einzelnen Methoden nicht mehr aus, da die Reihenfolge, in der sie aufgerufen werden, für die Ergebnisse ebenfalls von entscheidender Bedeutung sind. Darüber hinaus können sowohl innerhalb der Komponenten - z.B. durch die Verwendung von Vererbungen - als auch zwischen den einzelnen Komponenten komplexe Beziehungen auftreten. Man könnte in diesem Fall eine Testmethode entwerfen, die jede Klasse bzw. Komponente separat testet. Dies würde aber ein re-testen vieler Attribute bedeuten, die z.B. bereits in ihrer Elternklasse getestet wurden. Gosh et al. [GoMa99] bemängeln in diesem Zusammenhang die kaum vorhandene Skalierbarkeit bisheriger Testverfahren und Kriterien. Nach ihrer Auffassung sind diese zwar sehr hilfreich beim Testen von kleinen Einheiten, wie Funktionen in einem C-Programm, sie sind aber nicht mehr kosteneffektiv, wenn es um das Testen von großen Systemen geht. Die Schwierigkeit liegt dabei in der Explosion der Anzahl der möglichen Pfade, die genommen werden können, um das kleinste Element, wie z.B. eine Entscheidung oder ein "definition-use-Paar" zu erreichen. Gosh et al. [GoMa99] sind sogar der Auffassung, dass sich die Größe der zu testenden Bereiche mit der Anzahl der Komponenten exponentiell erhöht. Eine entsprechende Testmethode und die dazu gehörenden Kriterien für Komponenten müssen daher eine Skalierbarkeit aufweisen bzw. berücksichtigen.

2.2. Testdatengenerierung und -adäquadität

Üblicherweise bieten Testmethoden ein Maß, das angibt, wie viel getestet wurde. Ein überdeckter Bereich ("coverage domain") ist eine Menge von programmbezogenen Entitäten (z.B. eine Gruppe von Klassen), die bereits durch den Test "gemessen" wurden (vgl. [GoMa99]). Ziel ist es nun, die Testmengen so zu verbessern, dass möglichst viele Bereiche überdeckt werden, was nach Gosh et al. [GoMa99] wiederum zu einer Verbesserung der Zuverlässigkeit des Systems führen soll. Idealerweise wünscht man sich eine Testmenge, die während des Testens sämtliche, eventuell vorhandenen Fehler findet, so dass diese anschließend restlos beseitigt werden können. Das hierzu notwendige, extensive Testen ist allerdings in realen Softwareentwicklungsumgebungen nicht möglich. Eine weitere Schwierigkeit liegt darin, Testfälle zu generieren, die kleinere Elemente mit abdecken (vgl. Kapitel 2.1.). Daher müssen nach Gosh et al. [GoMa99] realistische Ziele gefunden werden, die die Sicherheit und Qualität einer Komponente in einem oder mehreren Kriterien sicherstellen.
Darüber hinaus besitzen nach Gao [Gao00] die meisten "in-house"-Komponenten keine angemessenen Testmengen, da sie für ein spezielles Projekt mit seinen jeweiligen Anforderungen und Funktionen entwickelt wurden. Im originalen System werden vielleicht Teile ausgeführt, die in einem anderen System niemals ausgeführt würden. Hieraus folgt, dass einige Tests für die neue Umgebung komplett irrelevant werden. Z.B. könnten im neuen Szenario Teile einer Komponenete gefordert werden, die im ersten nur als nie erreichbare Ausnahmefälle angesehen wurden und die

daher nur leicht oder gar nicht getestet wurden (vgl. [Weyuker98]). Die Wiederverwendung der Komponenten und ihrer Testmengen in anderen Projekten oder Produkten ohne jegliches Prüfen, kann darüber hinaus zu adäquaten Testproblemen führen.
Beim Generieren von Testfällen für eine Komponente, müssen diese deshalb so gewählt werden, dass sie möglichst viele Bereiche überdecken. Des weiteren sollten diese Testfälle wiederverwendbar sein.

2.3. Selektion von Untermengen von Komponenten für einen Test

Eine Komponente kann den Service einer anderen Komponente benötigen. Testet man diese Komponente, bleibt die Frage, wie man diese Services mit einbezieht. Man könnte sog. "Stubs" verwenden, die das Ergebnis so an die Komponente zurückgeben, wie es von ihr erwartet wird. Dies erzeugt aber nicht die Bedingungen, die in einem aktuellen System herrschen (vgl. [GoMa99]). Zudem können Deadlocks oder sog. "race conditions" ("kritische Timingprobleme") nur dann entdeckt werden, wenn eine Komponente zusammen mit anderen verwendet wird. Selbst wenn eine Komponente einwandfrei funktioniert, ist bei weitem nicht sichergestellt, dass sie auch mit anderen Komponenten korrekt zusammenarbeitet.
Das selbe Problem taucht auf, wenn eine Komponente "multithreaded" ist (vgl. Kapitel 3.8). Wird nur ein Thread eines Client der Komponente ausgeführt, tauchen möglicherweise keine Fehler auf. Werden hingegen mehrere Clients verwendet, können mehrere Komponententhreads gestartet und damit Timingprobleme ausgelöst werden. Die Anforderungen an das Testen einer Komponente besteht somit in der Berücksichtigung der anderen, involvierten Komponenten und das mit einbeziehen von Multithreading-Ausführungen. Hieraus ergibt sich das Problem, welche Komponenten in diesen Test miteinbezogen werden sollen bzw. müssen.

2.4. Sequenz, in der Komponenten getestet werden

Im vorhergehenden Abschnitt wurde gezeigt, dass Komponenten mit anderen Komponenten zusammen getestet werden müssen. Dies zieht das Problem nach sich, in welcher Reihenfolge diese Komponenten getestet werden sollten. Man könnte mit den Komponenten beginnen, die keinen Service einer anderen Komponente beanspruchen (vgl. [GoMa99]). Nachfolgende Komponenten würden so gewählt, wie sie die anderen Dienste von schon zuvor getesteten Komponenten in Anspruch nehmen. Dies müsste aber voraussetzten, dass die Systemarchitektur eine Layerstruktur aufweist, so dass die am weitesten unten liegenden Komponenten keine Dienste von Komponenten in höheren Schichten in Anspruch nehmen müssen. Leider passen nach Gosh et al. [GoMa99] die Interaktionen zwischen den Komponenten in den meisten realen Fällen nicht in eine Layerarchitektur, da insbesondere Zyklen zwischen Komponenten auftreten können. Es ist somit nicht eindeutig, mit welcher Komponente man als erstes starten sollte. Alle Komponenten müssen demnach gleichzeitig getestet werden.

2.5. Ad-hoc Testsuiten für Softwarekomponenten

Komponentenhersteller beliefern ihre Kunden in der Regel nicht mit Testsuiten und Testinformationen zu ihren Komponentenprodukten. Darüber hinaus unterstützten sie weder Akzeptanztests noch Qualitätsreports (vgl. [Gao00]). Um die Qualität einer Komponente zu testen und um ihr Verhalten zu verstehen, müssen die Käufer somit zunächst hohe Aufwendungen investieren, um die gelieferten Dokumente zu verstehen und um anschließend selbst eine Testsuite und einen

Testtreiber erstellen zu können.
Für "in-house"-Komponenten erstellen die Entwickler normalerweise ihre eigene Testsuiten. Diese Testsuiten werden aber nach Gao [Gao00] überwiegend "ad-hoc" entwickelt, d.h. sie könnten mit inkonsistenten Testformaten, verschiedenen Technologien und Repositorien, sowie unterschiedlichen Werkzeugen erstellt worden sein. Gao [Gao00] ist der Auffassung, dass die Ursache hierfür darin liegt, dass sie in der Regel von unterschiedlichen Teams und in unterschiedlichen Projekten entwickelt wurden. Die so entstandenen Testtreiber und Stubs wurden somit für ein spezielles Projekt, basierend auf den dort gegebenen Anforderungen entwickelt. Diese Methode wird bei der Übertragung auf komponentenbasierte Softwareprojekte sehr ineffektiv und kostspielig. Meistens verfügen sie über keine wohldefinierten Standardinterfaces zwischen Komponenten und Testsuiten, sowie zwischen Komponenten und Testumgebung. Der größte Seiteneffekt ist nach Gao [Gao00] der, dass sie nur schwer wiederverwendet, integriert und auf systematische Weise verwaltet werden können um Komponententests, Komponentenintegration und Systemtests zu unterstützen.
Darüber hinaus zieht nach Harrold et al. [HaLiSi] die Korrektur eines Fehlers, der in einer Komponente von einem Komponentenanwender enthüllt wurde, üblicherweise wesentlich höhere Kosten nach sich, als die Korrektur eines ähnlichen Fehlers, der während der Integration eines nichtkomponentenbasierten Systems behoben wird, da eine solche Komponente in vielen verschiedenen System zum Einsatz kommen kann. Der Komponentenhersteller muss daher zwei Kriterien berücksichtigen (vgl. [HaLiSi]): Zunächst muss der Hersteller die Komponente effektiv als eine kontext-unabhängige Einheit testen. Effektives und adäquates Testen von Softwarekomponenten, unabhängig vom Kontext ihres Einsatzes erhöht das Benutzervertrauen in die Qualität der Komponenten und reduziert den Aufwand des Benutzers beim Testen dieser Komponenten. Zweitens müssen die Hersteller besseres Testen und Analysen der Benutzerapplikation unterstützen, so dass die Fehler bezüglich einer Komponente einfacher erkannt werden können, und zwar noch bevor die Benutzerapplikation veröffentlicht wird. Dies kann die Qualität eines komponentenbasierten Systems erhöhen und die Kosten eines Tests und den späteren Support des Systems verringern.
Als Resultat für das Testen von Komponenten folgt, dass "ad-hoc" Testsuiten niemals die komplette Qualität einer Komponente sicherstellen können und sogar hörere Kosten nach sich ziehen könnten. Eine Anforderung muss also sein, kontextunabhängige Testsuiten zu entwickeln, mit denen man die Komponenten sicherer testen kann (vgl. auch Kapitel 4.1.).

2.6. Ad-hoc Qualitätskontrollprozesse und Zertifizierungskriterien

Nach Gao [Gao00] haben viele Softwareteams "in-house"-Qualitätskontrollprozesse für Softwareprodukte etabliert. Bis jetzt bleibt allerdings unklar, ob der existierende Qualitätskontrollprozess und -Standard auf Softwarekomponenten appliziert werden kann. Das Fehlen von Komponenten-Standardqualitätskontrollprozessen führt somit regelmäßig zu "ad-hoc"-Komponententestprozessen und Zertifikationsstandards (vgl. [Gao00]). Wie schon bei den "ad-hoc-Testsuiten" folgt hieraus zwangsläufig eine verschlechterte Qualität der resultierenden Softwarekomponenten.
Ein weiteres Qualitätsproblem nennt Weyuker [Weyuker98]: Sofern sich der Hersteller einer Komponente dazu entschließen sollte, den Support für eine Komponente einzustellen oder sogar vom Markt verschwindet, hätte dies unabsehbare Konsequenzen für die von dieser Komponente abhängenden Projekte.

2.7. Komponentenüberwachung und -Verfolgung

Wie in [Gao99] beschrieben, zeigt die sog. "Komponenten-Beobachtbarkeit" ("component observability") nach Roy S. Freeman an, wie einfach es ist, ein Programm bzgl. seinem operationalen Verhalten und seinen Ein- und Ausgabeparametern, zu beobachten. Dieser Punkt betrifft insbesondere den Benutzer einer Komponente, der auf diese Weise während des Testvorganges genau beobachten kann, welche Daten in eine Komponente hinein- und welche wieder hinausfließen. Dies umfasst ebenfalls eine Verfolgungsmöglichkeit der externen Operationen, der Performance, der externen sichtbaren Ereignisse und der öffentlichen Eingeschaftsstatie einer Komponente. Hieraus folgt unmittelbar, dass das Design und die Definition eines Interfaces einer Komponente (wie Ein- und Ausgabe Interfaces) starke Auswirkungen auf diese Beobachtbarkeit haben. Man kann die Technik von Roy S. Freedman verwenden um ein Interface einer Komponente darauf zu testen, wie einfach es ist, seine Operationen und die Ausgabe im Verhältnis zur Eingabe zu beobachten (vgl. [Gao99]).

In der Praxis der Komponentenentwicklung stellt die sog. "Komponentenverfolgung" ("component traceability") einen weiteren wichtigen Faktor bei der Beobachtbarkeit von Komponenten ("component observability") dar. Die "Verfolgung" einer Softwarekomponente bezieht sich auf die Erweiterbarkeit ihrer eingebauten Überwachungsfähigkeiten. Dies umfasst z.B. das "mithören" der GUI-Ereignisse oder der Kommunikation, sowie die Verfolgung der komponenteninternen Operationen, der internen Objektstati, der Ereignisse, der Performance und das Mitlesen der internen Attribute einer Komponente (vgl. [Gao99]).
Gao [Gao99] unterscheidet bei der Verfolgung zum einen "wie" das Verhalten einer Komponente von außen verfolgt werden kann ("behavior traceability") und zum anderen ob und wie gut der Anwender einer Komponente Einfluss auf diese Verfolgungsmechanismen nehmen kann ("trace controllability").

a) behavior traceability
Diese "Verfolgung des Verhaltens" bezieht sich auf den Grad, mit dem eine Komponente die Verfolgung seines eigenen, internen und externen Verhaltens erleichtert. Nach [Gao99] versäumen die Entwickler bislang, ihre Softwarekomponenten bei der Auslieferung mit einem guten "Verfolgungsverhalten" auszustatten, da sie ein größeres Augenmerk auf das interne, als auf das externe Verhalten legen. Dadurch wird es den Komponententestern, den Integrationsentwicklern und den späteren Kunden erschwert, das externen Verhalten der Softwarekomponenten zu überwachen und zu prüfen.

b) trace controllability
Der zweite Punkt, die "Kontrollierbarkeit der Verfolgung" bezieht sich auf die Erweiterung der Steuerungsmöglichkeiten einer Komponente. Hierbei sollen die, von der Komponente zur Verfügung gestellten Verfolgungsfunktionen durch den Benutzer an dessen eigene Bedürfnisse anpassbar sein. Die Entwickler von komponentenbasierten Systemen können so die verschiedenen, eingebauten Verfolgungsfunktionen kontrollieren und einstellen. Dies umfasst z.B. das Aktivieren und Deaktivieren jeder einzelnen Verfolgungsfunktion und die Auswahl von Verfolgungsformaten und Verfolgungsrepositorien ("trace repositories"). Obwohl die aktuellen, kommerziellen und "in-house" - Komponenten diese Möglichkeiten nicht bieten, würden sie nach Gao [Gao99] eine ideale, kosteneffektive Eigenschaft für Tester und das Wartungspersonal darstellen, um das Debuggen und die Komponentenintegration zu erleichtern. Die "Verfolgungskontrollierbarkeit" ist

insbesondere bei komplizierten Komponenten mit benutzerdefinierbaren, funktionalen Eigenschaften in einem verteilten Umfeld nützlich und notwendig.
Um den Zugriff auf die eingebauten Verfolgungsfunktionen zu erleichtern, muss ein Standard-Verfolgungsinterface innerhalb der Komponente geschaffen bzw. definiert werden. Die Standardisierung des Komponentenverfolgungs-Interface und des Verfolgungsformates ist sehr wichtig, um eine systematische "Verfolgungslösung" ("tracking solution") für komponentenbasierte Software zu erstellen.

2.8. Komponentenkontrollierbarkeit

Jerry Gao fasst in [Gao00] und [Gao99] die "Kontrollierbarkeit" eines Programmes oder einer Komponente als eine wichtige Eigenschaft auf, die anzeigt, wie einfach ein Programm oder eine Komponente durch seine Eingaben, Operationen, Verhalten und Ausgaben kontrollierbar ist. Nach Gao [Gao00] umfasst diese "Kontrollierbarkeit" drei Punkte. Der Erste bezieht sich auf die Kontrollierbarkeit des Verhaltens und die Ausgabedaten im Bezug auf die Operationen und Eingabedaten. Der Zweite bezieht sich auf die eingebauten Möglichkeiten der Anpassung und Konfiguration der internen, funktionalen Eigenschaften einer Komponente. Der Letzte umfasst schließlich die Installationsmöglichkeiten. Die Kontrollierbarkeit einer Komponente hat starke Auswirkungen auf das Testen, da die Aktivierung bzw. Dekativerung der unterschiedlichen Funktionalitäten einer Komponente, verschiedene Auswirkungen auf das gesamte System haben kann.
Für Testingenieuere und Komponentenbenutzer muss nach Gao [Gao00] die Kontrollierbarkeit um zwei weitere Faktoren ergänzt werden. Der erste Faktor bezieht sich auf die Kontrollmöglichkeiten der eingebauten Verfolgungsmechanismen für das Komponentenverhalten ("component behavior tracking"), der Zweite auf die Kontrollmöglichkeiten der eingebauten Tests in einer Komponente (vgl. Kapitel 2.7.).

2.9. Komponentenpräsentation

Wie Jerry Gao in [Gao00] und [Gao99] beschreibt, setzt sich die Präsentation einer Komponente aus vier Faktoren zusammen. Die Präsentation für den Benutzer ist der erste Faktor. Sie umfasst das Komponentenbenutzerhandbuch, das Benutzerreferenzhandbuch und die Komponenten-Applikationsschnittstellen-Spezifikationen. Mit Hilfe dieser Dokumente lernen Benutzer, wie eine Komponente zu verwenden ist.
Die Präsentation des Designs und der Analyse einer Komponente ist der zweite Faktor. Entwickler benutzen Komponentenanalyse und Designdokumente um Interfaces, Eigenschaften, die funktionalen Fähigkeiten, das Verhalten und die Constrains einer Komponente zu verstehen.
Das eigentliche Komponentenprogramm ist der dritte Faktor. Es enthält den Komponentensourcecode und seine unterstützenden Elemente, wie den Installationscode, die Testtreiber und Stubs.
Der letzte Faktor umfasst Komponententests und Qualtiätsinformationen. Dies beinhaltet neben den Komponenten-Testplänen und/oder Testsuiten, auch die Testmetriken, Test- und Problemreports, sowie die Wartungsdokumente.
Jerry Gao kritisiert in diesem Zusammenhang, dass die meisten Komponenten von Drittanbietern dem Benutzer lediglich mit einem Komponenten-Benutzerhandbuch und den "Application Interface"-Spezifikationen geliefert werden. Es sollten aber mindestesten Akzeptanzpläne und Qualitätsinformationen, wie Testmetriken-Reports herausgegeben werden, da die Verständlichkeit einer Komponente davon abhängt, wie viel Informationen von einer Komponente bekannt sind und

wie gut diese präsentiert werden (vgl. [Gao00] und [Gao99]).

2.10. Zusammenfassung

Im Folgenden sollen die in Kapitel 2 genannten Kriterien, die auch für herkömmliche Software gelten, den Kriterien, die nur für Komponenten gelten, gegenüber gestellt werden.

a) Kriterien, die auch für herkömmliche Software gelten:

Kriterium/Problem	Übertragbarkeit auf Komponenten
Skalierbarkeit der Testadäquatskriterien	Sofern der Sourcecode vorhanden ist, sind die bisherigen Kriterien auch auf Komponenten anwendbar. Dabei bleiben aber die komplexen Beziehungen zwischen den Komponenten unbeachtet.
Testdatengenerierung und -Adäquadität	Das Problem geeignete Testdaten zu finden, existiert in ähnlicher Form auch bei großen, komplexen oder verteilten Softwaresystemen.
Ad-hoc Testsuiten für Softwarekomponenten	"Ad-hoc" Testsuiten sind aufgrund ihrer Kontextabhängigkeit nicht auf Komponenten übertragbar.
Ad-hoc Qualitätskontrollprozesse und Zertifizierungskriterien	Die bisherigen Qualitätssicherungsprozesse lassen sich nicht ohne Anpassungen auf Komponenten übertragen.

b) Kriterien, die nur für Komponenten gelten:

Kriterium/Problem	Anmerkung
Skalierbarkeit der Testadäquatskriterien	Es müssen neue Kriterien gefunden werden, die sowohl skalierbar sind, als auch anwendbar, wenn keine Sourcecode vorhanden ist.
Selektion von Untermengen für einen Test	Komponenten müssen zusammen mit anderen Komponenten getestet werden. Die hier auftretenden Wechselwirkungen sind in dieser Form bei herkömmlicher Software nicht zu finden.
Sequenz, in der Komponenten getestet werden	Bei komponentenbasierter Software treten, im Gegensatz zu herkömmlicher Software, Abhängigkeiten zwischen Komponenten auf, die den Test nachhaltig beeinflussen können.
Komponentenüberwachung und Verfolgung	Dies sind explizite Eigenschaften von Komponenten, die in den Testmethoden, Kriterien und Anforderungen berücksichtigt werden müssen.
Komponentenkontrollierbarkeit	Ebenfalls eine besondere Eigenschaft der Komponenten; durch die Konfigurierbarkeit werden alle Testvorgänge komplexer.
Komponentenpräsentation	Da Komponenten verwendet werden, um aus ihnen neue Softwaresysteme zu schaffen, stellen sie andere Anforderungen an die Dokumentation.

Tabelle 1 Gegenüberstellung der in Kapitel 2 genannten Kriterien

3. Probleme und Anforderungen beim Testen von komponentenbasierenden Systemen

Ein primäres Ziel der Komponentenentwicklung ist es, aus wiederverwendbaren Softwarekomponenten spezifische Softwaresysteme nach den Bedürfnissen der entsprechenden Benutzer zu bauen. Daher hängt die "Testbarkeit" ("testability") eines Programms stark von der Testbarkeit der involvierten Komponenten und deren Integration ab (vgl. [Gao99]). Nach Gao [Gao00] beschränken sich die bisherigen Systemtests auf das Prüfen des Systemverhaltens durch Performancetests, Stresstests, Wiederherstellungstests und Installationstests. In der Entwicklung eines komponentenbasierten Programms treten in diesem Zusammenhang allerdings verschiedene Probleme auf.
Im Gegensatz zu Kapitel 2 sollen in diesem Kapitel die Probleme aufgeführt werden, die sich nicht mehr auf das Testen von einzelnen Komponenten, sondern auf das Testen der aus Komponenten gebildeten Systeme beziehen. Diese Probleme entstehen dabei primär auf Seiten des Komponentenbenutzers. Die folgende Gliederung lehnt sich erneut an [GoMa99] und [Gao00] an.

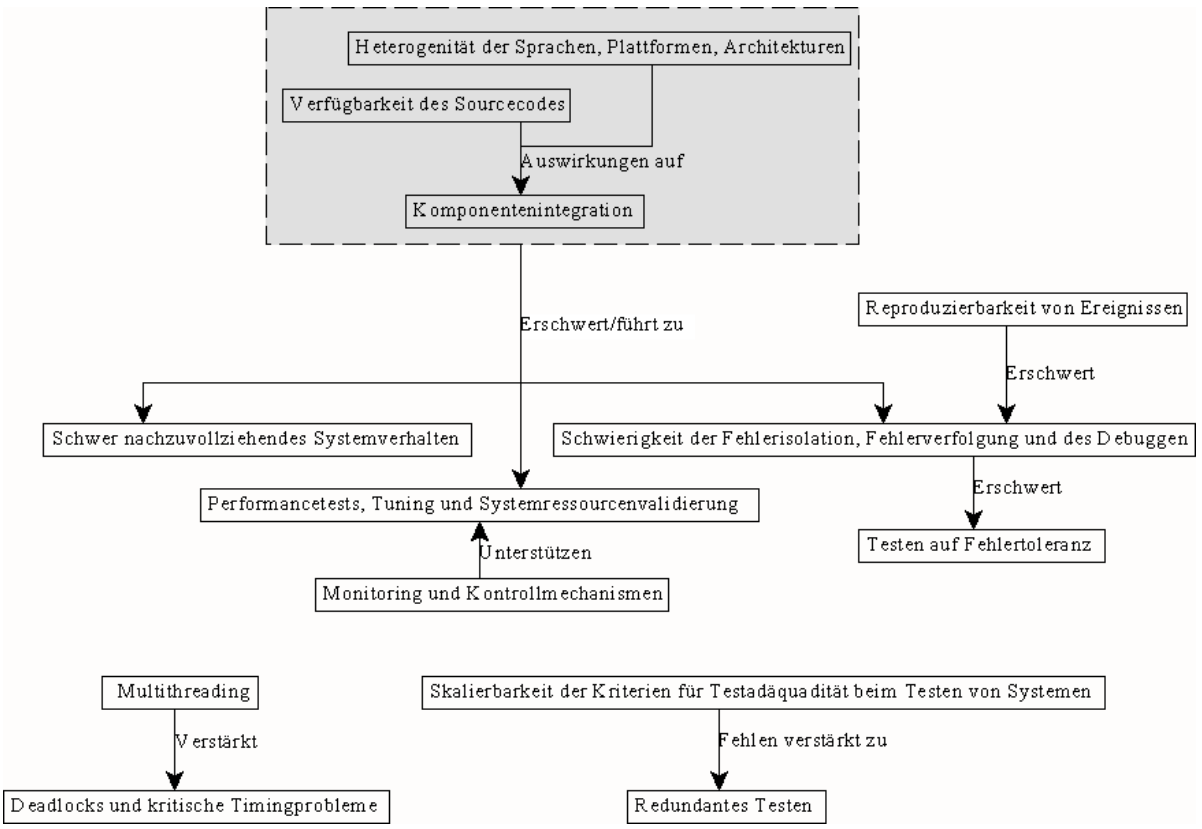

Abbildung 2 Übersicht über die in Kapitel 3 genannten Kriterien

3.1. Skalierbarkeit der Kriterien für Testadäquadität beim Testen von Systemen

Das Problem der Skalierbarkeit bei herkömmlichen Kriterien der Testadäquadität wirkt sich auch auf das Testen von Systemen aus. Die einzelnen Kriterien müssen mit der Anzahl der Komponenten, die über eine Anzahl von Hosts in einem System verteilt sein können, skalierbar sein (vgl. [GoMa99] und Kapitel 2.1.). Gerade im Hinblick auf dynamische Systeme, bei denen sich die Anzahl der Komponenten während der Laufzeit ändert, ist dieser Punkt äußerst wichtig.

3.2. Redundantes Testen

Nach Gosh et al. [GoMa99] werden Komponenten oft zunächst separat getestet und erst in einem anschließenden, inkrementellen Prozess, zusammen. Werden die Komponenten einzeln getestet, so können bisherige Testadäquaditätskriterien verwendet werden, um die Qualität der jeweiligen Testmengen sicherzustellen. Wird anschließend das gesamte System getestet, so werden nach Gosh et al. [GoMa99] in der heutigen Praxis genau dieselben Kriterien, unter Berücksichtigung der Überdeckungsbereiche ("coverage domains") der übrigen Komponenten verwendet. Dies führt aber zwangsweise zu vielen, größtenteils unnötigen Re-Tests der Komponenten.
Es müssen daher Methoden gefunden werden, die es den Testern erlauben, die Anzahl der redundanten Tests herauszufinden. Sind die Komponenten bereits separat gut getestet, muss man weiterhin wissen, wie viele zusätzliche Tests während der Integration notwendig sein werden.

3.3. Verfügbarkeit des Sourcecodes

Softwarekomponenten können entweder "in-house" oder “off-the-shelf” bezogen werden. Der Entwickler einer Komponente hat Zugang zum Sourcecode, wohingegen der Benutzer einer “COTS”-Komponente dies in der Regel nicht hat. Abhängig von der Verfügbarkeit des Sourcecodes sind deshalb verschiedene Testtechniken notwendig (vgl. [GoMa99]).
Falls der Sourcecode der Komponenten nicht verfügbar ist, können traditionelle Analyse- und Testverfahren, wie Kontroll- oder Datenflusstests, nicht auf komponentenbasierte Systeme adaptiert werden. Ein Weg um die Analyse ohne den Quellcode durchzuführen, stellt nach Harrold et al. [HaLiSi] die Verwendung von konservativen Approximationen über Analysebeziehungen innerhalb der Komponenten und über Analysebeziehungen, die in der Beziehung Benutzerapplikation mit dem Code in der Komponente begründet sind, dar. Ein wesentlicher Nachteil solcher Approximationen liegt in ihrer Ungenauigkeit.
Ein weiteres Problem, dass durch das Fehlen von Quellcode entstehen kann, nennt Weyuker [Weyuker98]: Die ausgewählten Komponenten könnten nicht zusammenpassen, wodurch in einigen Fällen sogar ein intensives Reengeniering notwendig werden würde. Diese Probleme haben direkte Auswirkungen auf die Qualtität und Zuverlässigkeit eines, auf Komponenten basierenden Systems. Die Entwicklung von Testsuiten kann für ein solches System ebenfalls zu einem Problem werden, da die hierfür notwendigen Einblicke in den Quellcode nicht zur Verfügung stehen.

3.4. Heterogenität der Sprachen, Platformen und Architekturen

Die Komponenten eines Systems können in verschiedenen Programmiersprachen oder für die Verwendung auf verschiedenen Hardware- und Softwareplattformen geschrieben worden sein. Mit der Hilfe von Middelware, wie CORBA und DCOM, können Komponenten gegenseitig, unabhängig von der Sprache und der Plattform interagieren. Wenn ein solches System getestet wird, müssen die verwendeten Testmethoden und Tools unabhängig von der Sprache und den Plattformen sein (vgl. [GoMa99]). Ein Analysetool, das auf der Implementationssprache der Benutzerapplikation basiert, würde somit auf den anderen Komponenten fehlschlagen (vgl. [HaLiSi]).

3.5. Monitoring und Kontrollmechanismen in verteilten Softwaretests

Verteilte Softwaresysteme laufen auf vielen Computern in einem Netzwerk. Der Monitoring- und Kontrollmechanismus während des Tests ist daher weitaus komplexer als der für ein zentralisiertes Softwaresystem (vgl. [GoMa99]).
Mechanismen für zentralisiertes oder verteiltes Monitoring und Kontrollen müssen geplant werden. Des weiteren müssen verteilte Datensammlungen und Speichermechanismen so entworfen werden, dass sie mögliche Überläufe von Datenpuffern abfangen. Aufgrund der Anzahl der Komponenten und ihrer hohen Lebensdauer, ist die Anzahl der Daten, die beim Monitoring und bei Kontrollen gesammelt werden, normalerweise sehr hoch. Daher muss dafür Sorge getragen werden, dass die eingesetzten Monitore und Kontrollen keine Flaschenhälse ("Bottlenecks") währen des Monitoring- und Kontrollprozesses aufweisen (vgl. [GoMa99]).

3.6. Reproduzierbarkeit von Ereignissen

Konkurrierende Ausführung und asynchrone Kommunikation können in verteilten Systemen frequentiell auftreten. Zusammen mit dem Fehlen einer vollen Kontrollmöglichkeit über die Umgebung, führt dies zu dem Problem, dass ein Ausführungsverhalten möglicherweise nicht reproduzieren werden kann. Dies ist nach Gosh et al. [GoMa99] der Hauptgrund, warum das reine Testen von Komponenten nicht ausreicht, um die Verlässligkeit eines Systems sicherzustellen.

3.7. Deadlocks und “race conditions” (kritische Timingprobleme)

Verteilte oder konkurrierende Systeme können kritische Timingprobleme oder Deadlocks aufweisen. Diese Probleme sollten bereits beim Testen gefunden und anschließend direkt eliminiert werden (vgl. [GoMa99]).
CORBA besitzt z.B. die Möglichkeit "one-way" Aufrufe ("Einbahnstraßen-Aufrufe") zu erzeugen, die in einer "best-effort"-Semantik geliefert werden. Falls der Client mehrmals diese Aufrufe tätigt, könnten die im System auftretenden Timingprobleme die Reihenfolge, in der diese Aufrufe erzeugt wurden, verändern. Treffen Letztere schließlich in der falschen Rangfolge beim Server ein, könnte dies ein fehlerhaftes Verhalten des gesamten Systems zur Folge haben.
CORBA unterstützt darüber hinaus sog. "call-back"-Mechanismen, mit denen ein Server ein Client zurückrufen kann. Dies kann ebenfalls zu potentiellen Deadlocks führen (vgl. [GoMa99]).

3.8. Multithreading

Um die Performance zu erhöhen, können Komponenten unter Verwendung von Threads implementiert werden. Ein perfekt getestetes "single-threaded" System, muss nicht mehr einwandfrei arbeiten, wenn es unter der Verwendung von "muliple-threads" re-implementiert wurde. So könnten in einem solchen Fall sich überlagernde Bereiche, Timingprobleme und Deadlocks auftreten. Die verschiedene Threadingmodelle, die in Serverimplementierungen verwendet werden, wie bzw. Objektadapter in CORBA, können dabei unterschiedliche Folgen haben (vgl. [GoMa99]).

3.9. Testen auf Fehlertoleranz

Softwaresysteme müssen sehr oft fehlertolerant sein. Applikationen für Kernkraftwerke, Raumflüge, medizinische Ausrüstung und Abwehreinrichtungen etc., müssen fehlertolerant sein, da mit ihnen hohe Kosten sowohl an Leben, als auch an Ressourcen verbunden sind. Nach Gosh et al. [GoMa99] kann angenommen werden, dass die an fehlertolerante Systeme gestellen Anforderungen, zum Zeitpunkt des Testens bereits bekannt sind.
Um die Fehlertoleranz zu testen, ist es notwendig, das gesamte System zu testen, um die Auswirkungen von Fehlern zu beobachten, die in den individuellen Komponenten oder in den zugrunde liegenden Kommunikationsmechanismen auftreten. Die Fähigkeit eines Systems, in der Gegenwart von Fehlern gut zu arbeiten, hängt von der Korrektheit des Fehlerbehandlungscodes ab. Es ist daher notwendig, dass dieser Code vor der Weiterverwendung getestet wurde. Gosh et al. [GoMa99] schlagen hier die Verwendung von Fehlerinjektionstests vor, bei denen unter Testbedingungen Fehler in ein System eingefügt werden. Auf die Tester kommen durch diese Methode allerdings zwei Probleme zu: Zum einen reicht die reine Injektion von Fehlern in ein Programm nicht aus. Die Stelle, an der ein Fehler injiziert wurde, muss während irgendeiner Ausführung auch erreicht werden. Zum anderen sieht der Tester die Auswirkungen des Fehlers erst, nachdem die Daten durch verschiedene Module oder Komponenten gelaufen sind. Es könnte daher Tage dauern, bis sich ein injizierter Fehler auswirkt (vgl. [GoMa99]). Darüber hinaus wird eine Menge an Fehlern zum Zwecke der Injektion benötigt. Falls man eine allgemeine Menge an Fehlern für eine Klasse an Systemen hat, hilft dies bei der Automatisierung dieses Prozesses: die Fehler können in einem Fehlerinjektionstool angeboten und von jedem Tester für eine Injektion ausgewählt werden (vgl. [GoMa99]).

3.10. Schwer zu verstehendes Systemverhalten

Im Systemtest liegen die Schwierigkeiten beim Verstehen und Verfolgen des Systemverhaltens und der Funktionen. Dies hat nach Gao [Gao00] folgende Gründe:
Entwickler verwenden "ad-hoc" Mechanismen um das Verhalten von "in-house" Komponenten zu verfolgen, was aufgrund von inkonsitenten Verfolgungsnachrichten und -Formaten, sowie diversen Verfolgungsmethoden, zu einem schwerer verständlichen System führt. Desweiteren existieren in Komponenten von Drittherstellern keine eingebauten Verfolgungsmechanismen und -Funktionen, die das Überwachen des externen Verhaltens erleichtern könnten. Ebenso fehlen Funktionen zur Konfiguratuion der Verfolgungsmöglichkeiten in Komponenten oder Methoden und Technologien, um das externe Verhalten der Komponenten in einem komponentenbasierten Programm zu kontrollieren und zu überwachen (vgl. Kapitel 2.7).
Hieraus folgt, dass das Systemverhalten für den Systemtester nur schwer nachvollziehbar ist, da zum einen unterstützende Methoden, als auch eine direkte Unterstützung durch den Hersteller einer Komponente fehlen.

3.11. Schwierigkeit der Fehlerisolation, der Fehlerverfolgung und des Debuggen

In einem System verwenden Komponenten, die von verschiedenen Teams hergestellt wurden, möglicherweise unterschiedliche Verfolgungsmechanismen und Verfolgungsformate. Inkonsitente Fehlerverfolgungsmechnismen in den Komponenten sind möglicherweise eine, inkonsitente Verfolgungsformate und Fehlercodes in Fehlerverfolgungsnachrichten die andere Folge (vgl. [Gao00]). Um dieses Problem beseitigen zu können, sind somit einheitliche Fehlerverfolgungs-

mechanismen und -Formate gefordert.
Darüber hinaus kann das Beseitigen von Fehlern signifikant schwerer werden, falls das Entwicklerteam nicht mehr verfügbar oder nicht mehr mit dem Code vertraut ist (vgl. [Weyuker98]). Des weiteren kann sich das Entwicklungsteam nicht mehr zuständig für das Modifizieren des Codes fühlen und solche Aufgaben an andere Entwickler weiterleiten, die nicht mit dem Code vertraut sind.

3.12. Performancetests, Tuning und Systemressourcenvalidierung

Performancetests für komponentenbasierte Programme sind eine wesentliche Herausforderung bei den Systemtests, da Komponentenhersteller normalerweise keine Performanceinformationen über ihre Komponenten bereitstellen (vgl. [Gao00]). Als Folge müssen Systemtester und Integrationsentwickler großen Aufwand beim Identifizieren von Performanceproblemen und den betroffenen Komponenten leisten. Mit der Weiterentwicklung der Komponenten und dem System steigen nach Gao [Gao00] als Folge die Performancetestkosten und der Tuningeinsatz kontinuierlich an.
Weitere Performanceprobleme können sich nach Weyuker [Weyuker98] zum einen aus der Größe und zum anderen aus der Komplexität des Systems ergeben.
Da nach Gao [Gao00] die meisten Komponenten ihre Systeminformationen nicht anbieten, wird es für Systemtester schwer, die Gründe für Systemressourcenprobleme während des Testens zu finden. Jerry Gao ist der Auffassung, dass aus diesem Grund Softwarekomponenten mit konsistenten Mechanismen und Interfaces entworfen werden müssen, so dass das Verfolgen des Verhaltens, der Funktionen, der Performance und der Ressourcen möglich wird.

3.13. Komponentenintegration

In einer komponentenbasierten Softwareproduktlinie, werden die Programme auf einer Menge von Softwarekomponenten aufgebaut. Nach Gao [Gao00] gibt es zwei Faktoren, die die Komplexität dieser Komponentenintegration beeinflussen. Der Erste betrifft die Anzahl von involvierten Komponenten, der Andere die Anpassungsfähigkeit von Komponenten. Neben den existierenden Integrationsmethoden (wie die inkrementelle Integration), die auf die Komponentenintegration applizierbar sind, benötigen Entwickler neue, effektive Integrationsmethoden und systematische Tools, um die Komponentenintegration zu unterstützen.

3.13.1. Schwierigkeit des Erstellens von Integrationstestsuiten

Für die Schwierigkeit des Erstellens von Integrationstestsuiten, die auf Komponententestsuiten aufbauen, gibt es nach Gao [Gao00] zwei Gründe: Der Erste betrifft die Konstruktion von "ad-hoc"-Testsuiten, was in inkonsistenten Testinformationsformaten, diversen Zugangsmechanismen und verschiedenen Datenspeichertechnologien endet. Dies betrifft insbesondere die Wiederverwendbarkeit der Testsuiten für die Komponentenintegration. Der andere Grund ist das Fehlen von Testselektionsmethoden, um Integrationstestsuiten, basierend auf Komponententestsuiten, zu erstellen. Die meisten existierenden Techniken beschränken sich auf die Auswahl von "white-box"-Tests, die nicht auf die Auswahl von "black-box"-Tests für die Komponentenintegration und -Reintegration applizierbar sind. Die Gründe hierfür liegen nach Gao [Gao00] zum einen darin, dass die Testgenerations- und Testselektionsmethoden ihre gesamten integrierten, funktionalen Eigenschaften auf den Komponentenlevel konzentrieren müssen und zum anderen sind die Interaktionen und Beziehungen zwischen Komponenten wesentlich komplizierter als die Interaktionen zwischen Prozeduren. Daher werden neue systematische Methoden benötigt, um sowohl

Tests von "black-box"-Komponententestsuiten zu identifizieren und zu selektieren, um dann daraus Integrationstestsuiten bilden zu können, als auch Tests von einer Integrationstestsuite für Regressionstests selektieren zu können (vgl. [Gao00]).

3.13.2. Hohe Kosten durch "ad-hoc"-Testtreiber
In der komponentenbasierenden Softwareentwicklung ist jedes Produkt das Integrationsergebnis einer Menge von angepassten Komponenten an die gegeben Anforderungen. Wenn sich die Anzahl der Komponenten erhöht, erhöht sich auch die Anzahl der Komponentenintegrationen. Der Bau einer Integrationsumgebung für ein komponentenbasiertes Softwaresystem ist nach Gao [Gao00] aufgrund der "ad-hoc"-Konstruktion von Komponententesttreibern und Stubs schwierig und langatmig. Mittlerweile komplizieren Komponentenanpassungs-Eigenschaften die Integrationsinfrastruktur und -Umgebung.

3.13.3. Fehlen von Komponentenintegrationstestmodellen und Testkriterien
Eine Komponente mit angepassten Eigenschaften wird in einem integrierten Komponentensystem eingesetzt, um eine spezielle Untermenge von ihren implementierten Funktionen bereit zu stellen. Ein komponentenbasiertes Programm besteht aus vielen, dieser angepassten Komponenten (vgl. [Gao00]). Ohne die Identifizierung des Teils der Funktionalitäten, die von der speziellen Applikation verwendet werden, kann nach Harrold et al. [HaLiSi] ein Test- oder Analysetool unpräzise Ergebnisse liefern. Integrationstests werden, durch die Anpassungsmöglichkeiten innerhalb einer Komponente und ihrer unterschiedlichen Integrationen erschwert. Um die Komponentenintegration zu unterstützen, fordert Gao [Gao00] neue Integrationsmodelle und Kriterien. Die Modelle sollten die Komponentenintegrationsstruktur und Integrationsfunktionen auf Komponentenebene repräsentieren, Komponenten "einfangen" und ihre Anpassungseigenschaften in der "Black-Box"-Sicht und die verschiedenen Interaktionen zwischen den Komponenten berücksichtigen. Integrationstestkriterien sollten die Datenbereichsüberdeckungen ("data domain coverages") im Hinblick auf die Interaktion von Komponenten, integrierte funktionale Überdeckungen ("functional coverage") und integrierte Strukturüberdeckungen ("structure coverage") berücksichtigen.

3.14. Zusammenfassung

Im Folgenden sollen die in Kapitel 3 genannten Kriterien, die auch für herkömmliche Software gelten, den Kriterien, die nur für Komponenten gelten, gegenüber gestellt werden.

a) Kriterien, die auch für herkömmliche Software gelten:

Kriterium/Problem	Übertragbarkeit auf Komponenten
Skalierbarkeit der Kriterien für Testadäquadität beim Testen von Systemen	Bei herkömmlichen, verteilt laufenden Anwendungen entstehen ähnliche Probleme, wie bei Systemen, die auf Komponenten basieren.
Monitoring und Kontrollmechanismen in verteilten Softwaretests	Im Gegensatz zu herkömmlichen Systemen, entstehen bei komponentenbasierenden Systemen wesentlich mehr Monitoring- und Kontrolldaten.

Multithreading	Auch die bisherigen Softwaresystene können sich anders verhalten, wenn Sie durch "multiple-threads" re-implementiert werden. Durch die starken Wechselwirkungen, die in komponentenbasierenden Systemen herrschen, können die Auswirkungen dort aber wesentlich stärker sein.
Testen auf Fehlertoleranz	Herkömmliche Software muss ebenfalls Fehlertolerant sein. Bei komponentenbasierten Systemen können aber die Auswirkungen bisheriger Testmethoden (wie Fehlerinjektionstests) nur sehr viel schwerer, bzw. erst wesentlich später entdeckt werden.

b) Kriterien, die nur für Komponenten gelten:

Kriterium/Problem	Anmerkung
Redundantes Testen	Würde man die bisherigen Kriterien und Methoden ohne Anpassungen auf Komponenten übertragen, würden einige Tests doppelt ausgeführt. Parallelen zu herkömmlichen Systemen tauchen nur noch im Zusammenhang mit objektorientierter Software auf.
Verfügbarkeit des Sourcecodes	In der Regel (insbesondere bei COTS-Komponenten), steht der Quellcode nicht zur Verfügung. Dies erfordert neue Testverfahren und -Technologien auf "black-box"-Basis.
Heterogenität der Sprachen, Plattformen und Architekturen	Herkömmliche Verfahren werden für eine bestimmte, feste Architektur und in der Regel in nur einer bestimmten Sprache geschrieben. Dort können sich die Tests auf die jeweiligen Gegebenheiten stützen, was bei komponentenbasierten Systemen nicht möglich ist.
Reproduzierbarkeit von Ereignissen	Durch die starken Wechselwirkungen zwischen den Komponenten, ist es nicht mehr Möglich, ein bestimmtes Verhalten auf Wunsch exakt zu reproduzieren.
Deadlocks und kritische Timingprobleme	Durch die starken Wechselwirkungen zwischen den Komponenten, können Deadlocks und Timingprobleme auftreten; diese Probleme werden durch Multithreading noch verschärft.
Schwer zu verstehendes Systemverhalten	Aufgrund des fehlenden Sourcecodes und der mangelhaften Unterstützung durch den Hersteller, ist das gesamte Systemverhalten des aus Komponenten erzeugten Systems wesentlich schwerer zu verstehen.
Schwierigkeit der Fehlerisolation, Fehlerverfolgung und des Debuggen	Aufgrund des Fehlens von Standards, des Quellcodes, sowie dem Bezug von Komponenten aus verschiedenen Quellen, ist das Isolieren und das Verfolgen von Fehlern wesentlich schwieriger und umfangreicher als bei herkömmlicher Software.
Performancetests, Tuning und Systemressourcenvalidierung	Auch hier fehlen Standards, der Quellcode, und die Unterstützung durch den Hersteller, um Performance- und Ressourcenprobleme einfacher zu identifizieren und zu beseitigen.
Komponentenintegration	Probleme bereiten hier die Anzahl der Komponenten, deren Anpassungsfähigkeiten, das Problem der fehlenden Erstellungsmöglichkeiten von Integrationstestsuiten, sowie das Fehlen von Komponentenintegrationstestmodellen und Kriterien.

Tabelle 2 Gegenüberstellung der in Kapitel 3 genannten Kriterien

4. Probleme und Anforderungen an eine Testumgebung

Neben den Problemen und Anforderungen, die in Kapitel 2 und 3 genannt wurden, existieren weitere Probleme und Anforderungen, die sich auf das Erstellen und den Einsatz einer Testumgebung beziehen.

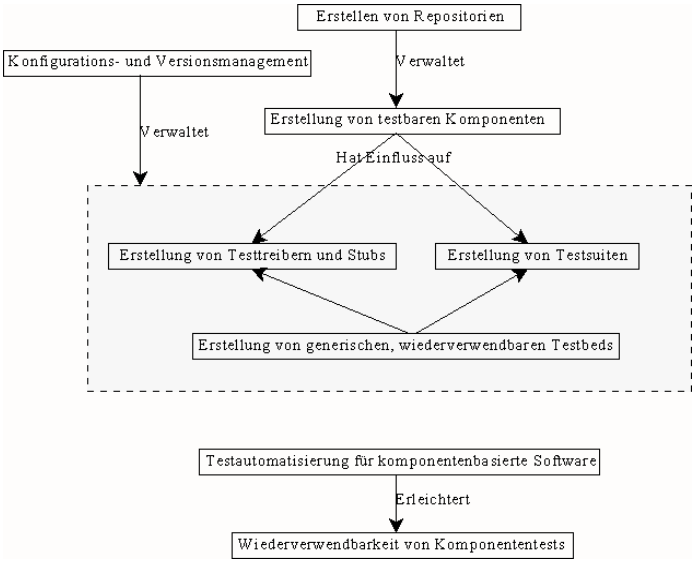

Abbildung 3 Übersicht über die in Kapitel 4 genannten Kriterien

4.1. Erstellung von testbaren Komponenten, Testsuiten, Testtreibern, Stubs und Testbeds

4.1.1. Erstellung von testbaren Komponenten

Anders als normale Komponenten, haben nach Gao [Gao00] testbare Komponenten die Eigenschaften, dass ihr Verhalten verfolgbar ist und sie über eine eingebaute Menge an Schnittstellen verfügen, um mit einer Menge an wohldefinierten Testmöglichkeiten (wie z.B. Testsuiten oder Stubs) zu interagieren. Darüber hinaus besitzen testbare Komponenten ein wohldefiniertes Test-Architekturmodell und eingebaute Test-Interfaces um ihre verschiedenen, unterschiedlichen funktionalen Eigenschaften, Daten und Interfaces bei einem Test durch Komponententestsuiten und einem Komponenten-Testbed zu unterstützen. Weiterhin müssen diese testbaren Komponenten einen standadisierten Mechanismus verwenden, der die eingebauten Tests einschließt. Hiermit wird der konsistente Zugriff auf die "build-in"-Tests ermöglicht. Allerdings ist es nicht einfach, die eingebauten Tests für andere Tests wiederzuverwenden, insbesondere, wenn ein "built-in"-Zugangsinterface fehlt (vgl. [Gao00]).
Drei essentielle Probleme bleiben hierbei nach Gao [Gao99] ungelöst: das Design und die Definiton einer allgemeinen Architektur und Test-Interfaces für testbare Komponenten, die Generation von testbaren Komponenten auf eine systematische Art und Weise und die Schwierigkeit der Kontrolle und Minimierung von Programm-Overheads und Ressourcen für die Testunterstützung.

4.1.2. Erstellung von Testsuiten

Das Testen von komponentenbastierten Systemen macht die Erstellung von Testsuiten notwendig. Die Basisidee ist hierbei, ein externes Tool zu verwenden, das dem Entwickler helfen soll, Testfälle zu erstellen, zu aktuallisieren, zu verwalten und zu unterstützen, sowie die in einem Test

benötigten Informationen zu verwalten (vgl. [Gao00]). Bekannt sind bereits sehr effektive Methoden für die Unterstützung von "black-", als auch "white-box" Tests für "in-house"-Komponenten. Die Entwickler können später selbst die Tests in einer Testsuite selektieren und für Integrations- und Regressionstests wiederverwenden. Um diese Methoden auf allgemeinere, komponentenbasierte Systeme übertragen zu können, benötigen die Entwickler ein gutes Softwaretest-Managemanttool und wohldefinierte Standards für alle Testinformationen, wie Testfälle, Testprozeduren und Testergebnisse. Nach Gao [Gao00] ist die Konsistenz der Schlüssel, um Komponententestsuiten bei der Entwicklung von komponentenbasierter Software erfolgreich einsetzen zu können. Um die "ad-hoc"-Konstruktion von Testsuiten zu vermeiden, müssen Standardformate, kompatible Technologien und konsistente Repositorien verwendet werden.
Nach Weyuker [Weyuker98] sollten in einer Testsuite für jede erstellte Softwarekomponente folgende Punkte gespeichert und aktuallisiert werden, sobald die Komponente verändert wird: Zeiger zwischen den Anforderungen und ihren Implementierungen, Zeiger zwischen individuellen Testfällen und die Teile des Codes, die für einen Test entworfen wurden (zwecks Verwaltung von Testfällen für einen Regressionstest) und Zeiger zwischen den Spezifikationen und Teilen der Testsuite (welcher Teil der Spezifikation wurde wie gut getestet). In allen Fällen sollte die Testsuite sowohl Eingaben, als auch die erwarteten Ausgaben verwalten. Dies erleichtert Regressionstests bei der einer Änderung einer Komponente.

4.1.3. Erstellung von Testtreibern und Stubs

In der Praxis verwenden Entwickler "ad-hoc"-Methoden um modulspezifische oder produktspezifische Testtreiber und Stubs, basierend auf den gegebenen Anforderungen und Designspezifikationen, zu erstellen. Nach Gao [Gao99] liegt der Hauptnachteil dieser Methode darin, dass die generierten Testreiber und Stubs nur für ein bestimmtes Projekt (oder Produkt) brauchbar sind, wodurch sie wiederum höhere Kosten bei der Konstruktion von Komponententesttreibern verursachen.
Erschwerend kommt hinzu, dass Komponenten Anpassungsfunktionen besitzen können, die es den Entwicklern ermöglichen sollen, die enthaltenen Funktionalitäten an ihre gegebenen Bedürfnisse anzupassen. Hierdurch kann der traditionelle Weg zur Erstellung von Komponententesttreibern und Stubs sehr teuer und ineffizient werden. Man benötigt daher neue, systematische Methoden, um Testtreiber und Stubs für diverse Komponenten und verschiedene Komponenten-Konfigurationen konstruieren zu können (vgl. [Gao99]). Jerry Gao schlägt deshalb in [Gao00] und [Gao99] vor, dass Testtreiber einer Softwarekomponente skriptbasierte Programme sein sollten, die nur "black-box"-Funktionen einer Komponente ausführen.
Teststubs einer Softwarekomponente simulieren ihre "black-box"-Funktionen und/oder Verhalten. Nach Gao [Gao00] gibt es zwei Mehtoden um Teststubs zu generieren. Bei der ersten Methode wird das Verhalten der Komponente durch ein formales Modell repräsentiert, wie z.B. einem endlicher Automat. Diese Methode hat den Vorteil der Wiederverwendbarkeit und der automatischen Stub-Erzeugung. Bei der zweiten Methode werden skriptbasierte Teststubs erzeugt, um spezifische, funktionale Verhalten einer Komponente in der "black-box" Ansicht zu simulieren. Ein Hauptvorteil dieser Methode ist die Flexibilität bei der Unterstützung verschiedener Anpassungen funktionaler Eigenschaften in einer Komponente. Der anderer Vorteil ist die Wiederverwendbarkeit von Teststubs, da jeder Teststub eine spezifische, zugängige Funktion einer Komponente simuliert. Es bleibt allerdings ein ungelöstes Problem, wie diese funktionsspezifischen Teststubs auf eine systematische Weise generieren werden können (vgl. [Gao00]).

4.1.4. Erstellung von generischen, wiederverwendbaren Testbeds
Im allgemeinen besteht eine Programm-Testumgebung ("Testbed") aus verschiedenen, die Tests unterstützenden Funktionen: Testwiederherstellung, Testausführung, Testergebnisprüfung und Testreport. Eine Testumgebung für Komponenten muss diese entsprechenden Funktionen bereitstellen. Auch hier stellt sich die Frage, wie man eine neue Testbed-Technologie erstellen kann, die auf verschiedene Komponenten, die mit verschiedenen Programmiersprachen und -technologien erstellt wurden, applizieren kann (vgl. [Gao99]). Gao [Gao00] schlägt hier vor, für die involvierten Interfaces zwischen Komponenten und Testbeds (Testausführungsinterfaces, Testinformationszugangsinterfaces und Interaktionsinterfaces), Standards einzuführen.

4.1.5. Erstellen von Repositorien
Wenn es ein primäres Ziel von komponentenbasierter Softwareentwicklung ist, Komponenten wiederzuverwenden, dann muss nach Weyuker [Weyuker98] ein signifikates Augenmerk auf das Design des Komponenten-Repositoriums gelegt werden, so dass Komponenten von verschiedenen Projekten aus genutzt werden können. Man muss sich entscheiden, wie Komponenten beim Entwurf eines neuen Systems für den sofortigen Einsatz gespeichert und identifiziert werden können.

4.2. Konfigurations- und Versionsmanagement

Das Konfigurationsmanagement ist nicht nur für die Softwareentwicklung, sondern auch für den Softwaretest von Bedeutung. Im Lebenszyklus von komponentenbasierter Software hängt dessen Weiterentwicklung von den Modifikationen und Upgrades der beteiligten Komponenten ab. Nach dem Ändern wiederverwendbarer Softwarekomponenten müssen Entwickler verbundene Testsuiten, Testtreiber und Teststubs aktuallisieren, um sukssesive Regressionstests durchführen zu können. Um die Evolutionskosten bei Regressionstests zu reduzieren, müssen Konfigurationsentwickler deshalb nach Gao [Gao00] Komponententesttreiber und Stubs ihrer Aufgabenliste hinzufügen. Die Versionen und Releases von Komponententestsuiten und Problemdatenbanken müssen sich für jede Komponente beobachten lassen. Manager müssen Versionskontrollen und Releasekontrollen bzgl. der betreffenden Ressourcen aller beteiligten Komponenten, einschließlich Dokumenten, Sourcecode und ausführbaren Code, Testsuiten, Testtreiber, Stubs und Problemdatenbanken erzwingen (vgl. [Gao00]).

4.3. Testautomatisierung für komponentenbasierte Software

Um die Testkosten von komponononentenbasierter Software zu reduzieren, stellt nach Gao [Gao00] die Testautomatisierung eine Hilfsmöglichkeit dar. Das systematische Verwalten von Komponententestinformationen ist der entscheidende Schritt um eine Testautomatisierung für komponentenbasierte Softwareentwicklung zu realisieren (vgl. [Gao00]).
Um dies zu erreichen, ist die Verwendung eines wohldefinierten Testinformationsstandards bei allen Projekten in einer Organisation notwendig, um die Automatisierung in den Bereichen Testdesign, Testdokumentation, Testplanung und Testausführung und Reporting zu unterstützen. Darüber hinaus kann ein Managementsystem zur Kontrolle, Verwaltung und Unterstützung aller Komponententestsuiten auf eine systematische Weise entwickelt werden. Weiterhin benötigt man Komponenten, die unter einem Komponententestbed oder einer Komponententestsuite getestet werden können (vgl. Kapitel 4.1.1.).

4.4. Wiederverwertbarkeit von Komponententests

Im Rahmen der Evolution der Softwarekomponenten, muss nach Gao [Gao99] eine große Aufmerksamkeit auf die Wiederverwendbarkeit von Komponententests gelegt werden. Dazu müssen einige systematische Testmethoden und Tools entwickelt werden, damit wiederverwendbare Komponententestsuiten eingerichtet, verschiedenartige Test-Resourcen, wie Testfälle, Testdaten und Test-skripte verwaltet und gespeichert werden können. Nach Gao [Gao99] verwenden Entwicklungsteams in der Praxis eine "ad-hoc" Methode um über ein Testmanagementtool die Komponenten-Testsuiten zu erstellen. Solange existierende Tools von verschiedenen Testinformationsformaten, Repositorien-Technologien, Datenbankschemata und Testzugangs-Interfaces abhängen, bleibt es für die Entwickler schwierig mit Softwarekomponenten aus den verschiedensten Quellen zu arbeiten. Dieses Problem betrifft insbesondere die Wiederverwendbarkeit von Komponententests bei Akzeptanztests und der Integration der Komponenten.
Gao [Gao99] schlägt zwei Möglichkeiten zur Lösung dieses Problems vor: Die Erste wäre, eine neue Testsuite-Technologie für Komponenten mit "plug-in-and-test" Techniken zu entwickeln. Die alternative Methode wäre, Komponententests innerhalb der Komponenten als "build-in" Tests ablaufen zu lassen (vgl. Kapitel 4.1.1.). Hierdurch wird das Testen von Komponenten vereinfacht und die Testkosten auf der Kundenseite reduziert, sofern ein benutzerfreundliches Testinterface zur Verfügung gestellt wird. Dabei besteht wiederum die Notwendigkeit, die entsprechenden Interfaces zu standadisieren.

4.5. Zusammenfassung

Im Folgenden sollen die in Kapitel 3 genannten Kriterien, die auch für herkömmliche Software gelten, den Kriterien, die nur für Komponenten gelten, gegenüber gestellt werden.

a) Kriterien, die auch für herkömmliche Software gelten:

Kriterium/Problem	Übertragbarkeit auf Komponenten
Erstellung von Testsuiten	Bei herkömmlicher Software werden in der Regel projektabhängige Testsuiten erstellt. Bei Komponententests müssen diese aber kontextunabhängig sein.
Erstellung von Testtreibern und Stubs	Die speziellen Eigenschaften von Komponenten, wie deren Anpassbarkeit bzw. Konfigurierbarkeit, fordern von den Testtreiber und Stubs zusätzliche Eigenschaften.
Erstellung von Testbeds	Testbeds für Komponenten müssen, im Gegensatz zu herkömmlicher Software, Plattform-, Sprachen-, und Technologieunabhängig sein.
Konfigurations- und Versionsmanagement	Das Konfigurations- und Versionsmanagement muss beim Einsatz von Komponenten ausgeweitet werden, da jede Komponente, sowie die zu ihr gehörenden Testtreiber und Suiten selbst Modifikationen und Upgrades unterworfen sind.
Testautomation	Sofern die Komponenten sich in einer Testumgebung testen lassen und Teststandards zwischen allen Komponenten eingehalten werden, lassen sich die Tests, wie auch schon bei herkömmlicher Software, automatisieren
Wiederverwendbarkeit der Tests	Sofern Testsstandards zwischen allen Komponenten eingehalten werden, lassen sich die verwendeten Tests wiederverwenden.

b) Kriterien, die nur für Komponenten gelten:

Kriterium/Problem	Anmerkung
Erstellung von testbaren Komponenten	Da jede Komponente eine eigene Software-Einheit darstellt, die mit anderen Komponenten in einem System agiert, muss zunächst sichergestellt werden, dass jede Komponente selbst testbar ist.
Erstellen von Repositorien	Die Eigenschaft der Wiederverwendbarkeit von Komponenten legt den Aufbau eines Repositoriums nahe.

Tabelle 3 Gegenüberstellung der in Kapitel 4 genannten Kriterien

5. Fazit

Testverfahren und Methoden für komponentenbasierte Softwaresysteme müssen zwei Grobanforderungen erfüllen. Zum einen müssen alle in den Kapiteln 2 und 3 genannten Probleme berücksichtigt, eingebunden und beseitigt werden und zum anderen muss erlaubt werden, eine Testsuite zu erstellen, die die in Kapitel 4 aufgeführten Kriterien erfüllt.

Zusammengefasst sind dies im einzelnen folgende Anforderungen:
- Neue, skalierbare Testkriterien müssen entworfen werden. Dies bedeutet insbesondere, dass diese Kriterien sich sowohl auf kleinere Teile einer Komponente, als auch auf das gesamte System anwenden lassen müssen.
- Die Testmengen müssen so gewählt werden, dass sie zum einen Wiederverwendbar sind, zum anderen aber auch alle Teile einer Komponente überdecken.
- Beim Test muss die Zusammenarbeit mit anderen Komponenten, gerade im Hinblick auf die Reihenfolge, in der man Komponenten testet und eine etwaige "multithread"-Ausführung, sowohl in der Komponente, als auch im gesamten System, berücksichtigt werden.
- Neue, kontext-unabhängige Testsuiten müssen entworfen werden. Darüber hinaus muss der Hersteller einer Komponente intensiver mit den Kunden zusammenarbeiten.
- Es werden neue, standardisierte Qualitätsprozesse sowohl für Komponenten, als auch für die entstehenden Systeme benötigt.
- Komponenten müssen standardisierte Beobachtungs- und Verfolgungsmechanismen aufweisen, die von außen steuer- und erweiterbar sein müssen.
- Die Kontrollierbarkeit und Anpassbarkeit der involvierten Komponenten muss bei einem Test berücksichtigt werden.
- Die Tests und die Komponenten selbst müssen umfassend dokumentiert werden, da gerade bei COTS-Komponenten hiervon die Verständlichkeit essentiell abhängt.
- Bei einem Test müssen Kriterien und Methoden gefunden und verwendet werden, die vermeiden, dass man bestimmte Bereiche doppelt oder unnötig testet.
- Die verwendeten Testmethoden müssen unabhängig von der verwendeten Sprache und Plattform sein. Darüber hinaus müssen sie auch ohne Quellcode auskommen können.
- Man benötigt Monitoring- und Kontrollmechanismen, die auch auf großen, verteilten Systemen arbeiten. Insbesondere müssen Deadlocks und Timingprobleme durch ein Verfahren erkannt und beseitigt werden können.
- Die involvierten Komponenten müssen fehlertolerant sein, was wiederum durch ein geeignetes Verfahren getestet werden muss.
- Man benötigt einheitliche, standardisierte Fehlerverfolgungsformate und -Mechanismen.
- Performance- und Ressourcenprobleme müssen durch geeignete Methoden identifizierbar sein.
- Testsuiten und -Verfahren müssen die Komponentenintegration unterstützen und berücksichtigen. Gleichzeitig müssen sie die Konfigurationsmöglichkeiten der Komponenten mit einbeziehen.

Man benötigt also neue Methoden zum Testen und zur Unterstützung von Softwarekomponenten, um sie hoch zuverlässig und wiederverwendbar zu machen (vgl. [Gao00]). Darüber hinaus benötigen Tester adäquate Testwerkzeuge, die den Prozess der Testsatzgenerierung und Testausführung automatisieren (vgl. [GoMa99]). Man benötigt weiterhin Analysetechniken, die effizient arbeiten und zwar sowohl auf den Komponenten, als auch auf dem System, das aus ihnen konstruiert wird. Tools werden benötigt, die zum einen die Auswirkungen von Veränderungen in Komponenten auf Systemen, die sie verwenden, verfolgen lassen und zum anderen helfen,

Systeme wiederholt zu testen und es erlauben, modifizierte Komponenten zu verwenden (vgl. [HaLiSi]).

Abschließend lässt sich sagen, dass ein Hauptaugenmerk auf die Schaffung von Standards in allen genannten Bereichen gelegt werden muss. Diese Forderung ging aus allen genannten Quellen eindeutig hervor.

Literaturverzeichnis

[Gao00] Gao, Jerry: Testing Component-Based Software, San Jose State University, San Jose 2000, hwww.engr.sjsu.edu/gaojerry/techreport/

[Gao99] Gao, Jerry: Component Testability and Component Testing Challenges, San Jose State University, San Jose 1999, hwww.engr.sjsu.edu/gaojerry/techreport/

[GoMa99] Gosh, Sudipto; Mathur, Aditya P.: Issues in Testing Distributed Component-Based Systems, Purdue Univertiy, West Lafayette, 1999

[HaLiSi] Harrold, Mary Jean; Liang, Donglin; Sinha, Saurabh: An Approach To Analyzing and Testing Component-Based Systems, Ohio State University, Columbus

[Weyuker98] Weyuker, Elaine J.: Testing Component-Based Software: A Cautionary Tale, IEEE Software, Ausgabe September/Oktober 1998