HANSER

Leseprobe

Mario Winter, Mohsen Ekssir-Monfared, Harry M Sneed, Richard Seidl, Lars Borner

Der Integrationstest

Von Entwurf und Architektur zur Komponenten- und Systemintegration

ISBN (Buch): 978-3-446-42564-4

ISBN (E-Book): 978-3-446-42951-2

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-42564-4

sowie im Buchhandel.

1 Einleitung

1.1 Worum geht es?

Heutzutage ist *Globalisierung* eines der am meisten verwendeten „Buzzwords“. Die Globalisierung ist ein vielschichtiger Prozess, der das ganze Spektrum des menschlichen (Zusammen-)Lebens verändert und von den Betroffenen – also von »uns allen«! – immer größere Integrationsleistungen fordert. Dieser Prozess beeinflusst vehement alle Bereiche wie Werte und Kultur, Umwelt, Politik, Wirtschaft und Technologie und damit auch die Software-Entwicklung.

Nun gibt es die Globalisierung nicht erst seit neuester Zeit. Man denke nur an die vielen Völkerwanderungen, die immer auch mit der Veränderung der Lebensgewohnheiten sowohl der wandernden als auch der besuchten (oder eroberten) Völker verbunden war. Nicht zuletzt umfasst der Begriff „Integration“ ja in der Soziologie *„… die Aus- und Umbildung einer Lebens- und Arbeitsgemeinschaft unter Einbezug von Menschen, die aus den verschiedensten Gründen wie z. B. Migrationshintergrund oder Behinderung bislang von dieser ausgeschlossen waren.“* ([Web Wikipedia], Stichwort: Integration). Ähnlich wie in den kontrovers diskutierten „Integrationstests“ für Einwanderer, welche die Verträglichkeit der Prüflinge mit dem sozialen Umfeld des Ziellandes überprüfen sollen, prüfen Integrationstests in der Software-Entwicklung die Schnittstellen und das reibungslose Zusammenspiel der Software-Systeme sowie der sie konstituierenden Bausteine untereinander.

Die Globalisierung heute fordert und fördert den Einsatz neuer Informations- und Kommunikationssysteme (IKS). Auf der anderen Seite beschleunigt deren Einsatz die Globalisierung („Technobalisierung“, s. [Ekssir-Monfared 2010]). Dieser Prozess ist keine Einbahnstraße. Durch die Globalisierung wird die Organisation der menschlichen Gesellschaft immer komplexer. Dies erfordert wiederum immer komplexere Lösungen und Software-Produkte, sodass auch immer mehr vorhandene Software-Systeme miteinander kombiniert und integriert werden. Die neuen, integrierten Lösungen können auch nicht mehr alleine von einem einzigen Software-Unternehmen gebaut werden. Selbst große »Player« wie z. B. Siemens, IBM, Microsoft und HP sind daher gezwungen, Partnerschaften einzugehen.

Die resultierenden Systeme sind aufgrund ihrer Komplexität und gegenseitigen Abhängigkeiten immer schwerer beherrschbar und entsprechend störanfällig geworden. Je nachdem, in welchem Anwendungsbereich ein (Software-)Fehler auftritt, sind unterschiedliche Auswirkungen mit unterschiedlichen Schadenshöhen möglich. So ist das Auftreten von Fehlern

besonders bei sicherheitskritischen Systemen natürlich anders einzustufen als das Auftreten von Fehlern in einem Spielgerät. Heute können sich z. B. die Folgen eines einfachen Stromausfalls in einem Ort lawinenartig und grenzüberschreitend über ein ganzes Gebiet ausbreiten und entsprechend verheerend sein. Ebenfalls kann das Auftreten eines einfachen Software-Fehlers wie einer Gleitkommaverschiebung durch Datenkonvertierung in einem Börsensystem weltweit große Schäden anrichten.

Aber auch solche mit der Globalisierung einhergehenden technischen Probleme gibt es nicht erst seit Kurzem. Ein Beispiel hierfür sind die unterschiedlichen Spurweiten und Signaltechniken der Eisenbahnen in den europäischen Ländern zu Beginn des 19. Jahrhunderts, die zu erheblichen Behinderungen im grenzüberschreitenden Bahnverkehr führten und erst durch kostspielige Integrations- und Standardisierungsmaßnahmen beseitigt werden konnten.

Fertige Anwendungen wurden und werden miteinander kombiniert und zusammengeschweißt, um geeignete Antworten auf die neuen Anforderungen zu geben. Dabei werden die verschiedenen Geschäftsprozesse und Funktionalitäten bzw. Informationen verschiedener Systeme miteinander integriert und zusammengeführt. Die Software-Herstellung geht in den letzten Jahren also immer mehr in die Richtung Systemintegration, um neue Funktionalitäten und Eigenschaften zu erstellen. Diese greifen oft über Unternehmensgrenzen hinaus auf andere „fremde“ Systeme zu, um bestimmte Anforderungen zu erfüllen.

Als Beispiele für die Integration von mehreren Anwendungen oder Geschäftsprozessen können die beliebten Mashup-Applikationen (dynamische Webanwendungen, welche größtenteils auf öffentlich verfügbaren Web Services basieren) oder auf dienstorientierten Architekturen (*Service Oriented Architecture*, SOA) basierte Anwendungen genannt werden. Dabei werden durch interne Interaktionen, Aggregationen oder Filterungen der kollaborierenden Teilsysteme oder -prozesse neuen Funktionalitäten oder Dienste geschaffen. In der Regel müssen bei der Systemintegration die in Kommunikation tretenden Schnittstellen der zu integrierenden Software-Systeme auf beiden Seiten angepasst werden oder es werden für diesen Zweck überhaupt neue gemeinsame Schnittstellen entwickelt. Der Integrationstest von integrierten IT-Anwendungen spielt daher eine immer wichtigere Rolle.

Leider ist die Störanfälligkeit zusammengesetzter Software-Systeme wesentlich höher als die mechanischer Systeme und auch als die von einfachen, als Einzelsystem *(standalone)* betriebenen Anwendungen. Die Systemintegration stellt daher hohe Anforderungen an die Qualität der Einzelsysteme sowie eine übergreifende Qualität über die Grenzen jedes einzelnen Systems hinaus. Diese Situation resultiert aus den gegenseitigen Abhängigkeiten zwischen den Systemen. Ein kleines Problem in einer Anwendung kann sofort zu einem großen Problem aller betreffenden Anwendungen eskalieren.

Fehler in den Schnittstellen tauchen naturgemäß erst dann auf, wenn die Teilsysteme beginnen, miteinander zu interagieren. Da die Entwickler eines bestimmten Teilsystems oft nicht wissen, was die Entwickler eines anderen Teilsystems annehmen, sind Missverständnisse aufgrund falscher Annahmen an der Tagesordnung. Jene Instanz, die validiert, ob die Summe aller einzelnen Ergebnisse eines Zusammenspiels auch den Anforderungen entspricht, ist der Integrationstest.

Aber nicht nur in solchen heterogenen Multisystemen spielt der Integrationstest eine große Rolle, nein, auch Einzelsysteme bis hin zu den beliebten Apps für den mobilen Einsatz erreichen heutzutage eine Komplexität, welche ohne die Zerlegung in einzelne Bausteine und damit einhergehende Integrationstests nicht beherrschbar ist.

1.2 Integrationstest in der heutigen Praxis

Der Integrationstest ist in der Praxis selten als separate Teststufe ausgebildet, obwohl dies in der Lehre der Testtechnologie schon immer vorgesehen war. Nach der reinen Lehre ist der Integrationstest im V-Modell eine Aktivität zwischen dem Ende des Komponententests und dem Beginn des Systemtests, bei der die Software-Bausteine wie z. B. Klassen bzw. Module, Komponenten, Teilsysteme usw. zusammengesetzt und getestet werden. Die Betonung liegt hier auf dem Test. Es wird geprüft, ob die zusammenzusetzenden Bausteine sich in ihrer Interaktion so verhalten wie erwartet.

Dabei gibt es verschiedene Ansätze der Zusammensetzung. Ein Ansatz ist, die Bausteine inkrementell, also einen nach dem anderen zusammenzufügen. Ein entgegengesetzter Ansatz ist, sämtliche Bausteine auf einmal zusammenzufügen und zu sehen, was passiert. Dieser Ansatz wird in der Testliteratur als „Big Bang"-Ansatz bezeichnet. Für die inkrementelle Zusammensetzung der Bausteine gibt es mehrere Ansatzpunkte. Man kann von oben bei den übergeordneten bzw. steuernden Bausteinen ansetzen und Top-Down von oben nach unten integrieren. Oder man beginnt mit den untergeordneten, passiven Bausteinen und arbeitet sich Bottom-Up von unten nach oben vor. Bei mehrschichtigen Software-Architekturen bietet sich auch die Möglichkeit, die Bausysteme von innen nach außen oder von außen nach innen zusammenzusetzen. Schließlich bietet sich der Ansatz an, einen Pfad nach dem anderen durch die Bausteine hindurch zu verfolgen und die Bausteine in der Reihenfolge hinzuzufügen, in der sie aufgerufen werden. Welcher Ansatz gewählt wird, hängt von der Architektur der Software und den Prioritäten des Projekts ab (siehe Bild 1.1).

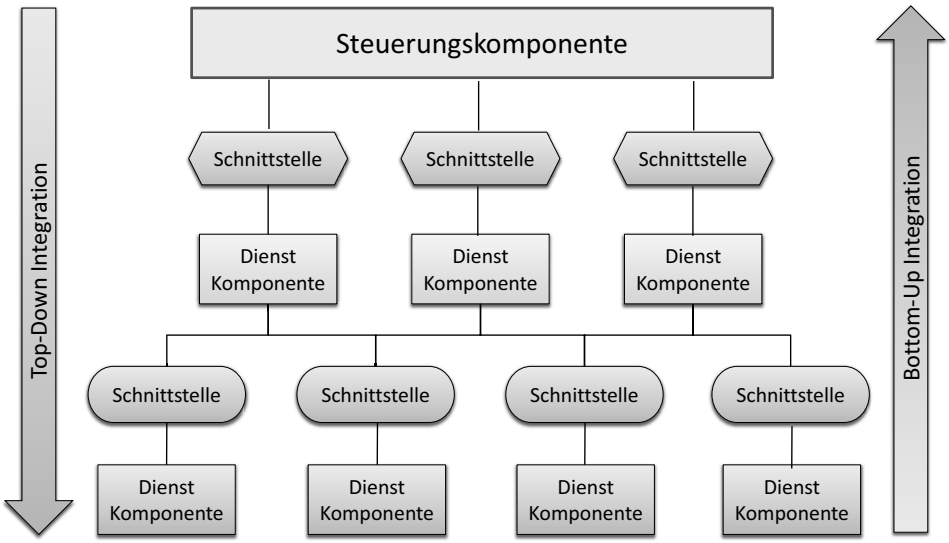

Bild 1.1 Strategien für Integrationstests

In der Praxis sieht dies anders aus. Oft gibt es überhaupt keinen Integrationstest. Der Entwickler einer Komponente testet sie für sich in einer Art Komponententest. Dann werden alle Komponenten gesammelt und zu einem Gesamtsystem zusammengefasst. Danach wird

mit dem Systemtest begonnen. Es kann sein, dass einige Bausteine nachgeliefert werden. Aber sie fließen direkt in den Systemtest. Einen Integrationstest einzelner Bausteingruppen gibt es nicht, allenfalls die Integration der Klassen in einer Komponente durch den einzelnen Entwickler. Das mag daran liegen, dass die meisten Systeme nicht groß genug sind, um eine stufenweise Integration zu rechtfertigen. Es mag auch daran liegen, dass die Zeit der Projekte für eine zusätzliche Teststufe zu knapp ist. Sie reicht oft nicht einmal für einen ordentlichen Systemtest aus. Drittens mag es an der Arbeitsteilung zwischen Entwickler und Tester liegen. Die Entwickler fühlen sich für „ihre“ Komponenten verantwortlich. Die Tester sehen sich für das Gesamtsystem verantwortlich. Für das, was dazwischen liegt, fühlt sich keiner verantwortlich. Hinzu kommt, dass die meisten Tester für einen echten Integrationstest nicht die erforderlichen technischen Kenntnisse mitbringen. Also bleibt der Integrationstest oft auf der Strecke. Viele Projekte gehen direkt vom Komponenten- zum Systemtest.

Nach Spillner ist der Integrationstest eine zwangsläufige Folge der Modularisierung. Je feiner Software-Systeme in einzelne Bausteine, sprich Module, Klassen, Komponenten und Teilsysteme aufgeteilt werden, desto schwieriger wird es, die Interaktion der Teile zu testen. Um der Schwierigkeit der Integration vieler Bausteine entgegenzuwirken, sollten die einzelnen Bausteine schrittweise integriert und nur die zusätzlichen Interaktionen getestet werden, bis das Gesamtsystem vollständig ist [Spillner 1990]. Demnach bedeutet jeder neue Baustein oder jede neue Bausteingruppe einen neuen Integrationstest. Es wird getestet, ob der neue Baustein zu den bereits integrierten Bausteinen passt. Damit kann der Tester seine Aufmerksamkeit auf den letzten dazu gekommenen Baustein und seine Interaktionen mit den bereits getesteten Bausteinen konzentrieren.

Aus Sicht des Testers ist diese Vorgehensweise durchaus logisch und praktikabel. Man muss sich nur die Zeit dafür nehmen. Aber da tritt das Problem auf. In den meisten IT-Projekten, die nach festgesetzten Kosten und zu einem bestimmten Termin abgewickelt werden müssen, gibt es keine Zeit dafür, vor allem dann nicht, wenn das Testen erst gegen Ende des Projekts begonnen wird. Die Versuchung ist allzu groß, die bis dahin fertigen Bausteine einfach zusammenzusetzen und zu beobachten, was passiert. Erst dann, wenn sich herausstellt, dass Probleme auftreten, geht man zurück und testet einzelne Zusammensetzungen in der Hoffnung, dort die Ursache des Problems zu finden. Es erfordert sehr viel Disziplin und auch genügend Zeit, um anders vorzugehen und jeden neuen Baustein einzeln zu integrieren. Die Erfinder dieser schrittweisen Teststrategie haben in einem IBM-Entwicklungslabor gearbeitet, wo sie scheinbar unter optimalen Bedingungen arbeiten konnten. Die Projekte hatten auch damals mehr Zeit und mehr Geld zur Verfügung. Wir sprechen über die 70er-Jahre. Inzwischen hat sich vieles geändert. Der Zeitdruck hat ebenso wie der Kostendruck enorm zugenommen. Der praktizierende Tester hat weder die Zeit noch die Mittel, so zu arbeiten, wie er nach der reinen Lehre arbeiten sollte. Dies erklärt die Kluft zwischen Praxis und Lehre.

Ein Versuch, mit den Anforderungen der Praxis umzugehen, ist die agile Software-Entwicklung. Dazu gehört auch der agile Test. Nach dem agilen Prinzip wird ein Software-System inkrementell gebaut: ein Baustein bzw. eine Bausteingruppe nach der anderen. Dadurch bekommt der Tester auch die Bausteine immer schrittweise bzw. kurz vor Ende jedes „Sprints“. Er hat nur bis zum Ende des Sprints Zeit, die neuen Bausteine mit den fertigen Bausteinen zu integrieren. Dann kommen die nächsten Bausteine. So gesehen ist der Integ-

rationstest ein unerlässlicher Bestandteil der agilen Vorgehensweise. Es wird permanent integriert und getestet. Die letzten Bausteine werden mit den bereits vorhandenen integriert, bis der letzte Baustein geliefert ist [Crispin 2009].

Das Problem hier ist die „Rekursion“ dieses Prozesses. Die Entwickler in einem agilen Entwicklungsprojekt entwickeln nicht nur neue Bausteine, sie überarbeiten und erweitern auch ständig die alten, bereits fertigen Bausteine. Diese müssen angepasst werden, damit sie zu den neuen Bausteinen passen. Die gleichen Bausteine werden immer wieder überarbeitet. Das heißt, es werden nicht nur neue Bausteine an die Tester geliefert, sondern auch jede Menge alter Bausteine, die bereits getestet wurden. Für die Tester bedeutet das, dass sie zurückgehen und die Tests der alten Bausteine wiederholen müssen. Ihr Arbeitsvolumen nimmt immer mehr zu, und zwar in dem Maße, wie das System wächst. Sie können die Masse an Bausteinen, die zu integrieren sind, kaum noch bewältigen. Eine Möglichkeit, dem entgegenzuwirken, ist der Einsatz einer durchgehenden Testautomatisierung von Beginn des Projekts an. Wenn dies nicht umsetzbar ist, bleibt das Projekt stehen oder die Qualität wird geopfert. Man könnte auch das Testteam verstärken, aber das bedeutet, das Projektbudget zu erweitern, was Projektverantwortliche ungern tun. Also wird allzu oft die Qualität geopfert [Black 2009].

Aus der Praxis der agilen Entwicklung wird berichtet, wie das Tempo beschleunigt wird und die Qualität der Software trotzdem gesichert ist [Link 2009]. Das entspricht aber in vielen Projekten nicht ganz der Wahrheit. Was passiert, ist, dass die leicht sichtbare Qualität gesichert wird. Die unsichtbare, innere Qualität der Software wird vernachlässigt und erst später im Zuge der Software-Evolution bzw. Refaktorisierung nachgebessert. Man nimmt vorübergehend Qualitätsrisiken in Kauf, um schneller fertig zu werden. Anderseits, da der Benutzer das Projekt selber steuert, bekommt er genau das, was er will. Er bekommt die richtige Lösung, bloß zunächst nicht optimal implementiert. In der klassischen Software-Entwicklung wird allzu oft die falsche Lösung richtig implementiert, was wiederum auch unbefriedigend ist. Tatsache ist, dass die Herausforderung des Software-Engineerings, richtige Produkte richtig zu bauen, noch lange nicht erfüllt ist [DeMarco 2009].

1.3 Eine kleine Geschichte

Wir befinden uns in einem kleinen Software-Haus, welches seit einiger Zeit die agile Software-Entwicklungsmethode Scrum einsetzt. In einem Projekt zur Entwicklung einer Client/Server-Anwendung mit Host-basiertem Backend und einer App für mobile Geräte als Frontend befindet man sich in den ersten Sprints. Das Grundgerüst der Software steht, ist aber noch instabil.

Drei Tage vor Sprint-Ende sitzen der Entwickler Klaus, die Testerin Anja und Claudia, die die Rolle des Product Owners innehat, beim Mittagessen. Klaus berichtet davon, dass er das neue Feature zur sekundengenauen Aktualisierung der Bewegungsdaten soeben in der Backend-Komponente fertiggestellt hat. Alle Komponententests zeigen den „grünen Balken“. Die Frontend-Komponente hatte Klaus bereits vorab entwickelt und mit einem einfachen Mock getestet.

Klaus: *„Ja, und nun war ich überzeugt davon, dass die Frontend-App und meine neue Backend-Funktion fehlerfrei zusammenarbeiten. Ich startete den Build und begann mit meinen Integrationstests. Leider ergab sich aber bereits beim dritten Testfall nicht das erwartete Ergebnis, sondern es wurde nur noch Schrott ausgegeben. Mir graut schon vor dem Debugging, weil ich nicht Frontend und Backend gleichzeitig im Debugger beobachten kann.“*

Anja: *„Hmm, du hast also deine Tests des Frontends mit dem fertiggestellten Backend wiederholt, und es hat nicht funktioniert?“*

Klaus: *„Ja, so war es!“*

Anja klärt Klaus auf, dass er keine Integrationstests durchgeführt hat, sondern lediglich Systemtests auf dem integrierten System. Daher wisse er nun auch nicht, wo er den Fehlerzustand suchen muss. Sie erklärt ihm, was ein Integrationstest ist.

Anja: *„Im Integrationstest wird gezielt das Zusammenspiel der zu integrierenden Bausteine geprüft. Dazu musst du die Daten und Abläufe an der Schnittstelle zwischen den Bausteinen betrachten. Für den Test musst du dir vorab überlegen, welche Dienste mit welchen Werten der Parameter in welcher Reihenfolge von der App auf dem Backend aufgerufen werden sollen und welche Werte dann die Rückgabeparameter des Backends haben müssen. Bei der Testausführung führst du die App mit solchen Eingabewerten aus, mit denen die Schnittstelle entsprechend deiner Testfälle angesteuert wird. Dann vergleichst du die tatsächlichen Werte an der Schnittstelle mit den erwarteten Werten. Falsche Aufrufreihenfolgen oder Dienstparameter zeigen dir, dass der Fehler in der App liegt. Falsche Rückgabewerte deuten auf Fehler in deiner neu entwickelten Backend-Funktion hin.“*

Claudia fährt dazwischen, dass das ja einen enormen Aufwand bedeuten würde: *„Unser Backend bietet 15 Dienste an, die von den acht App-Hauptfunktionen benutzt werden. Das würde ja zu 120 Integrationstests führen. Pro Test, sagen wir, zwei Personenstunden – nein, das gibt unser Budget nicht her!“*

Anja beruhigt Claudia und Klaus (der aber den Sinn schon selber sieht…): *„Claudia, deine Rechnung sieht alle Integrationstests separat und vernachlässigt auch die Möglichkeit, Testfälle der Komponententests für den Integrationstest wiederzuverwenden. Integrationstests können aber z. B. anhand zusammenhängender Abläufe miteinander kombiniert werden. Zur Wiederverwendung von Komponententests muss man sich nur überlegen, welche Werte an den Schnittstellen auftauchen sollen.“*

Klaus: *„Ja, richtig! Und meine Treiber und Stellvertreter aus dem Komponententest kann ich auch wiederverwenden. Die Daten an der Schnittstelle kann ich einfach an den RPC-Stubs abgreifen. Ich weiß auch schon, welchen meiner Komponententestfälle ich für den Integrationstest aufbohren kann, um den Fehler einzugrenzen.“*

Als erstes Fazit aus dieser kleinen Geschichte folgt, dass der Komponententest die Teile einzeln testet, während der Systemtest das Ganze unter die Lupe nimmt. Erst der Integrationstest testet gezielt das Zusammenspiel der Teile. Solche Tests sind nicht nur aus der praktischen Erfahrung heraus, sondern auch aufgrund theoretischer Überlegungen notwendig. Das sogenannte Anti-Kompositionstheorem besagt nämlich, dass ausreichende Tests für die Teile keine ausreichenden Tests für das Ganze darstellen. Umgekehrt sagt das Anti-Dekompositionstheorem, dass ausreichende Tests für das Ganze eben keine ausreichenden Tests für die Teile ergeben [Weyuker 1986]. Genau in diese Lücke zielt der Integrationstest.

1.4 Integrationstest und Software-Qualität

Integrationstests zielen auf verschiedene Aspekte der Software-Qualität. Die Qualitätsmodelle der ISO-Standards 9126-1 [ISO 9126 2001] bzw. ISO 25010 [ISO 25010 2011] unterteilen diese in Gebrauchsqualität sowie innere und äußere Qualität. Im Allgemeinen steht beim Komponententest die innere Qualität wie Wartbarkeit oder Übertragbarkeit der Software aus der technischen Sicht der Entwickler im Fokus (Entwicklertest). Beim System- und Abnahmetest werden eher die äußeren Qualitätsmerkmale wie Funktionalität, Benutzbarkeit oder Zuverlässigkeit aus der fachlichen Sicht der Anwender untersucht und analysiert. Der Integrationstest betrachtet nun den Grenzbereich zwischen technischer und fachlicher Sicht. Anhand der zu integrierenden Bausteine werden mehrere Stufen unterschieden. So findet man Integrationstests „im Kleinen“ z. B. auf der Stufe der in einer Klasse oder einem Modul gekapselten und miteinander zusammenspielenden Member-Funktionen und -Variablen oder auch auf der Stufe mehrerer zusammenspielender Klassen und Module. Diese werden oft von den Entwicklern gemeinsam mit eher technisch orientierten Testern durchgeführt. Der Integrationstest von Komponenten und Teilsystemen oder sogar von ganzen Systemen stellt den Integrationstest „im Großen“ dar und wird in der Regel von den technischen und fachlichen Testanalysten durchgeführt.

Auf allen Stufen des Integrationstests werden die Schnittstellen zwischen den jeweils zu integrierenden Bausteinen in Bezug auf das Zusammenspiel und den Datenaustausch getestet. In diesem Fall steht die innere Qualität im Vordergrund. Werden jedoch die öffentlichen Schnittstellen von Standard-Software oder von im Web öffentlich angebotenen Diensten getestet, steht eher die äußere Qualität im Fokus. Generell jedoch geht es beim Integrationstest eher um die innere Qualität der Software.

1.5 Für wen ist dieses Buch geschrieben?

Das vorliegende Buch richtet sich vorrangig an die Personen im Entwicklungsprozess, die mit der Planung, Vorbereitung und Durchführung des Integrationstest betraut sind. Dies sind in erster Linie Testmanager, Systemintegratoren, Integrationstestdesigner und Integrationstester. Aber auch Entwickler im weitesten Sinne sowie Projektmanager können wertvolle Informationen und praktische Hinweise für die tägliche Arbeit finden.

Den Testmanagern liefert das Buch den notwendigen Überblick zur Planung und Steuerung von Integrationstests. Die entsprechenden Kapitel beziehen sich dabei auf den allgemeinen Testprozess, sodass die entsprechenden Besonderheiten des Integrationstests immer vor dem Hintergrund entsprechender Erfahrungen anderer Tests eingeordnet und umgesetzt werden können.

Ebenso soll das Buch den Testern sowohl auf methodischer als auch technischer Ebene bei der Durchführung und Verbesserung der Integrationstests in ihrem Umfeld helfen. Die Aufgabe bei der Systemintegration ist der Test der Schnittstellen von Software-Systemen, die zwar selbständig ablauffähig sind, aber im Verbund und integriert in Betrieb genommen

werden müssen. Systemintegratoren müssen daher einen guten Überblick über das Verhalten der einzelnen Systeme, ihre Konfiguration sowie deren Datenaustausch mit anderen integrierten Systemen haben. Es ist durchaus möglich, dass die Systemintegratoren neben der Durchführung des Systemintegrationstests noch für die Durchführung von Multisystemtests und Multisystemabnahmetests (End-to-End-Test) verantwortlich sind. Systemintegratoren sind in der Regel ausgebildete und erfahrene Tester mit Domänen-Know-how. Für den Systemintegrationstest ist normalerweise eine gesonderte Testumgebung notwendig. Die Systemintegratoren müssen daher nach Bedarf von den jeweiligen Testteams der einzelnen Software-Systeme unterstützt werden.

Projektmanager erhalten einen Überblick, wie sich der Integrationstest in den Entwicklungsprozess einordnet, um ihn in der Planung des Projekts zu berücksichtigen. Sie können die Inhalte des Buches nutzen, um sich einen Überblick über die durchzuführenden Aufgaben des Integrationstest zu verschaffen und sich mit den prinzipiellen Methoden und Verfahren vertraut zu machen.

Die weiteren Rollen eines Entwicklungsprojekts können dieses Buch nutzen, um zu erfahren, wie sich ihre Ergebnisse direkt oder indirekt auf die Aufgaben des Integrationstests auswirken. Ihnen liefert das Buch Einblicke in die entsprechenden Voraussetzungen und Probleme von Integrationstests sowie die dort anzuwendenden Techniken und die zu findenden Fehler:

- Enterprise-Architekten: Deren modellierte Anwendungslandschaft ist Basis für den Integrationstest auf oberster Ebene
- Anforderungsingenieure: Deren Anforderungsspezifikationen beeinflussen den Integrationstest auf oberster Ebene
- Architekten und Entwurfsspezialisten: Mit ihrer festgelegten Architektur und ihrem Entwurf bieten sie einen unschätzbaren Wert für die Integrationstests auf unterster Ebene
- Unit- und Systemtester: Deren bereits spezifizierte (und eventuell automatisierte) Testfälle können unter Umständen im wieder verwendet werden

Um dem Leser den Einstieg in das Buch und die einzelnen Kapitel zu erleichtern, fasst Tabelle 1.1 die Meinung der Autoren zur Relevanz der einzelnen Kapitel bzgl. der betrachteten Rollen zusammen. Hierbei bedeuten ++ = *unbedingt lesen*, + = *sicher interessant*, # = *sollte bekannt sein* und o = *mal überfliegen*. Ein leeres Feld weist auf ein für die entsprechende Rolle eher nicht relevantes Kapitel hin. So muss sich beispielsweise ein technischer Tester detailliert mit dem Aufbau der „Integrationstestumgebung“ auskennen, aber nur Grundlagenwissen im Themengebiet „Integrationstestprozess“ besitzen. Ein Testmanager hingegen sollte die einzelnen Aufgaben des „Integrationstestprozesses“ stets vor Augen haben, aber ihm genügt die grobe Kenntnis über die „dynamische Testentwurfsverfahren“.

Tabelle 1.1 Rollenbezogene Relevanz der Kapitel

	Projekt-manager	Test-manager	Integra-tionstest-manager	Geschäfts-prozess-modellierer	Anforde-rungs-ingenieur	Software-Architekt	Software-Entwickler	Fachlicher Tester (Test-analyst)	Technischer Tester (Technischer Testanalyst)	Integra-tions-tester
2 Einführendes Fallbeispiel	+	++	++	o	o	+	+	+	++	++
3 Grundlegendes zum Softwaretest	o	#	#	o	o	O	+	#	#	#
4 Grundlagen des Integrationstests	+	++	++	+	+	++	++	++	++	++
5 Modellierung im Integrationstest	o	+	++	#	o	#	+	+	++	++
6 Software-Abhängigkeiten		o	++	o	o	#	#	+	++	++
7 Integrationsfehler-arten		+	++	o	o	+	++	+	++	++
8 Fallstudien zum Integrationstest	o	++	++	o	+	++	++	++	++	++
9 Integrationsstufen	o	++	++		o	++	++	++	++	++
10 Integrations-strategien	o	++	++	o	o	o	+	+	++	++
11 Integrationstest-prozess	+	++	++		o	+	o	+	+	++
12 Statische Analysen		o	+		o	+	o	o	++	++
13 Funktions- und wertebezogene Testentwurfs-verfahren		+	++	+	++	o	o	++	+	++

Tabelle 1.1 Rollenbezogene Relevanz der Kapitel *(Fortsetzung)*

	Projekt-manager	Test-manager	Integra-tionstest-manager	Geschäfts-prozess-modellierer	Anforde-rungs-ingenieur	Software-Architekt	Software-Entwickler	Fachlicher Tester (Test-analyst)	Technischer Tester (Technischer Testanalyst)	Integra-tions-tester
14 Ablaufbezogene Testentwurfsver-fahren		+	++	o	o	++	+	+	++	++
15 Fehlerbezogene, erfahrungsbasierte und weitere Test-entwurfsverfahren		+	++	o	o	o	+	++	++	++
16 Nicht-Funktionale Integrationstests		+	++	o	o	++	o	o	++	++
17 Integrationstest-umgebung	o	+	++			+	o	o	++	++
18 Integrationstest-automation und dynamische Analysen	o	+	++			o	o	+	++	++

4 Grundlagen des Integrationstests

Der Begriff „Integrationstest“ setzt sich aus den zwei Worten „Integration“ und „Test“ zusammen. Der (Software-)Test wurde im vorangegangenen Kapitel beleuchtet, sodass in diesem Kapitel zunächst der Begriff „Integration“ zu klären ist, bevor erläutert wird, was integriert wird, d. h. was genau die Ziele und Testobjekte des Integrationstests sind.

4.1 Was ist Integration?

Das Wort Integration stammt vom lateinischen Verb „integrare“ ab, was übersetzt in etwa „wiederherstellen“ oder „erneuern“ heißt. Der Duden gibt folgende drei umgangssprachliche Bedeutungen an:

> *Integrieren (1) zu einem übergeordneten Ganzen zusammenschließen; (2) in ein übergeordnetes Ganzes aufnehmen; (3) vereinheitlichen.* [Web Duden]

In der Mathematik bedeutet „integrieren“ die Berechnung des Integrals einer Funktion, also die Ermittlung ihrer Stammfunktion z. B. zur Berechnung der von ihr umschriebenen Fläche anhand des Zusammensetzens vieler infinitesimaler Flächenstücke. In der Technik bedeutet Integrieren das Ein- oder Zusammenfügen mehrerer (Einzel-)Teile zu einem größeren Teil oder einem System. Anstelle von „Teilen“ spricht man dabei je nach technischer Disziplin auch von Komponenten, Modulen, Einheiten oder Baugruppen. Einzelteile auf der untersten Stufe der Hierarchie, wie z. B. in der Elektronik Widerstände oder Transistoren, sind dabei oft genormt bzw. standardisiert und werden als frei auf dem Markt verfügbar und nicht weiter zerlegbar angesehen. Auch die Mittel zum Verbinden der Teile sind in weiten Bereichen der Technik genormt, man denke nur an Schrauben und Muttern in der Mechanik oder Stecker und Buchsen in der Elektrotechnik und Elektronik.

Das IEEE-Glossar zu System und Software Engineering definiert die Integration im informationstechnischen Sinn:

> *Integration (1) the process of combining software components, hardware components, or both into an overall system.* [Web SEVOCAB]

Bei der Integration geht es also darum, die einzelnen Komponenten eines Software-Systems zu einem funktionsfähigen Ganzen zusammenzusetzen.

4.2 Bausteine der Integration

Ein zentraler Begriff in den Erläuterungen zur Integration ist der eines Teils bzw. einer Komponente. Leider ist dieser Begriff in der IT-Welt durchaus unterschiedlich belegt. Das Spektrum reicht von der ganz allgemeinen Bedeutung als „Teil von irgendetwas Größerem“ über die immer noch recht generische Bedeutung als „separat zu entwickelndes Teil eines Programms oder Systems“ bis hin zu der sehr präzisen Bedeutung von „in sich abgeschlossenen, separat liefer- und installierbaren Software-Bausteinen, die gemäß eines bestimmten, präzise definierten Komponentenmodells erstellt und mit anderen Komponenten zu Programmen zusammengesetzt (komponiert) werden können“. Wir haben uns aufgrund dieser Vielfalt dazu entschlossen, in diesem Buch anstelle von Komponenten allgemein von den „Bausteinen“ der Integration zu sprechen.

Bausteine bilden die Bestandteile eines Software-Systems. Ein Baustein ist eine abgeschlossene Software- oder Hardware-Einheit mit festgelegter Schnittstelle (vgl. Abschnitt 6.1.1). Über diese definiert der Baustein die Interaktionsmöglichkeiten mit anderen Bausteinen und seiner Umwelt.

Ein Baustein muss nicht zwangsläufig für sich operabel bzw. lauffähig sein. Auch ist ein Baustein oft wiederum aus kleineren Bausteinen zusammengesetzt. Beispielsweise umfassen Klassen als elementare Bausteine objektorientierter Programme Features bzw. Member wie öffentliche und private Daten bzw. Instanzvariablen *(attributes, member variables)* und Prozeduren bzw. Methoden *(operations, member functions)*. Die öffentlichen, das heißt von außen sicht- und verwendbaren Features stellen die Schnittstelle der Objekte dieser Klasse dar. Eine Menge von Klassen wiederum kann ein Paket *(package)*, eine Funktionsbibliothek (*library*, *Dynamic Link Library* DLL), ein Web Service oder auch ein komplettes Teilsystem sein. Diese Menge kann ihrerseits als einzelner Baustein betrachtet werden und Bestandteil eines großen Softwaresystems sein, welches wieder als Baustein, wenn auch ein sehr großer, betrachtet werden kann. Im Rahmen einer großen Anwendungslandschaft, die wiederum aus vielen Systemen besteht, ist das soeben beschriebene Softwaresystem vielleicht nur ein kleiner Baustein von vielen.

Zur besseren Übersicht werden die Bausteine eines Systems oft grafisch als Rechtecke bzw. „Kästen“ dargestellt. So zeigt z. B. Bild 4.1 drei Bausteine `A`, `B` und `C` eines Systems.

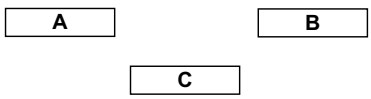

Bild 4.1 Drei Bausteine

Natürlich gibt es auch spezifischere Darstellungsformen für die Software-Entwicklung. So zeigt Bild 4.2 in einem Komponentendiagramm der UML (*Unified Modeling Language*, s. Abschnitt 5.1) die Struktur eines Systems aus drei Teilsystemen `Teilsystem_1` bis `Teilsystem_3`. `Teilsystem_2` wiederum besteht aus drei Komponenten, und `Komponente_3` umfasst die Klassen `Klasse_1` und `Klasse_2`. Für `Klasse_1` sind als Features die Instanzvariable `attribut_1` und die Methode `operation_1()` dargestellt. `Klasse_2` ist Unterklasse oder Spezialisierung von `Klasse_1` und erbt deren Instanzvariablen und Methoden.

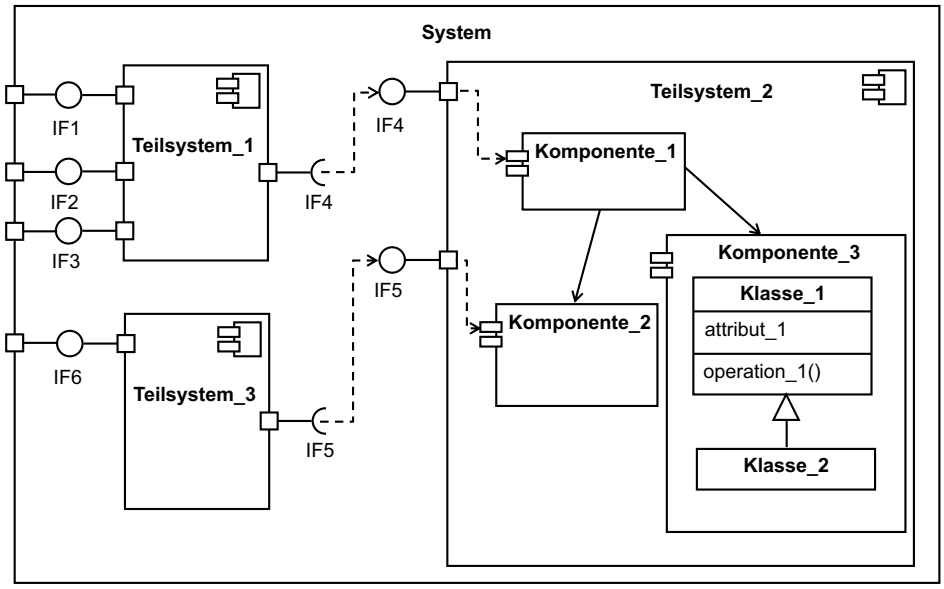

Bild 4.2 Bausteine der Integration

Die Verschachtelung von Bausteinen kann beliebig tief erfolgen und hat direkte Auswirkung auf den Integrationstest, denn die „Teilbausteine" eines großen Bausteins müssen ja ihrerseits zunächst zusammengesetzt und dabei in ihrem Zusammenspiel getestet werden, bis sie als gesamter Baustein funktionieren. Um die Verwirrungen über die unterschiedlichen, eher technisch bedingten Bausteinbezeichnungen (z. B. Komponente, Web Service etc.) ein wenig zu reduzieren, werden im Buch folgende Arten von Bausteinen für den Integrationstest betrachtet:

- System,
- Komponente bzw. Teilsystem,
- Klasse bzw. Modul sowie
- Member, d. h. Funktionen bzw. Methoden sowie Variablen.

Bild 4.3 zeigt den Zusammenhang dieser Bausteintypen in Form eines UML-Klassendiagramms (vgl. Abschnitt 5.1.2).

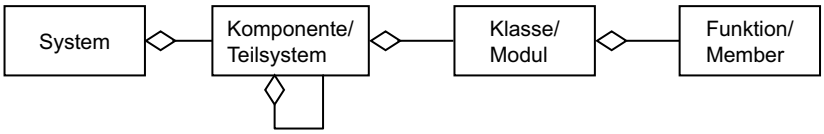

Bild 4.3 Bausteine im Integrationstest

Systeme setzen sich aus einer Menge von Teilsystemen bzw. Komponenten zusammen und können als Bausteine sogenannter Multisysteme oder Anwendungslandschaften betrachtet werden, in denen sie ihrerseits mit anderen Systemen integriert werden. Multisysteme selbst sind keine Bausteine der Integration, sondern ihr Ergebnis. Ob es sich bei einem

Baustein um ein System oder ein Teilsystem handelt, ist oft eher eine organisatorische denn eine technische Frage. In der Regel sind Systeme für sich alleine einsatzfähig, Teilsysteme jedoch nicht.

Komponenten sind eher programmiertechnische Artefakte mit i. d. R. einer individuellen Schnittstelle, über die ihre fachlichen Funktionen aufgerufen werden, sowie ggf. eine oder mehrere von Komponentenrahmenwerken wie z. B. Enterprise Java Beans (EJB) vorgegebenen Schnittstellen, über welche sie geladen, initialisiert, aktiviert, ggf. persistiert, deaktiviert und wieder entladen werden können.

Im Gegensatz zu Komponenten sind Teilsysteme eher auf architektonischer Ebene angesiedelt und oft durch mehrere, fachlich teilweise unabhängige Schnittstellen spezifiziert. Diese können auf programmiertechnischer Ebene von einer relativ losen Sammlung von Komponenten und/oder Klassen realisiert sein. Teilsysteme fassen nach logischen Kriterien und unter Beachtung des Geheimnisprinzips verschiedene Bausteine wie Klassen und/oder Komponenten unter einem Dach zusammen. Teilsysteme können geschachtelt werden. Sie definieren eine Enthält-Beziehung zwischen sich und den in ihnen organisierten Bausteinen. Sie kontrollieren lediglich die Benutzung dieser Bausteine, steuern aber keine eigenen Ressourcen bei. Wie eine Klasse besteht ein Teilsystem aus einer Schnittstelle und einer Realisierung bzw. einem Rumpf. Der Rumpf umfasst die zum Teilsystem gehörenden Bausteine, von denen eine Untermenge über die Schnittstelle des Teilsystems angeboten wird. Nur diese Klassen bzw. Teilsysteme sind außerhalb des Teilsystems sichtbar, alle anderen sind nur lokal benutzbar.

Einzelne Module bzw. Klassen bestehen aus Funktionen bzw. Methoden sowie Member-Variablen, die in der Klasse bzw. dem Modul gekapselt sind. Die Werte der Member-Variablen werden von den Funktionen manipuliert und bilden die möglichen Zustände von Instanzen bzw. Objekten ab. Funktionen bzw. Methoden wiederum umfassen einzelne Anweisungen, die letztendlich bei der Ausführung gemäß des Kontrollflusses abgearbeitet werden.

Bausteine alleine machen natürlich noch kein Software-System aus! Wie bereits skizziert werden bei der Integration Bausteine zusammengesetzt und dahingehend geprüft, ob ihr Zusammenspiel korrekt spezifiziert und implementiert ist. Ein Blick zurück zu Bild 4.1 zeigt zwar drei zu integrierende Bausteine, sagt aber nichts über deren Zusammenspiel aus. Bild 4.2 zeigt immerhin, dass die dort dargestellten Bausteine bzgl. unterschiedlicher Beziehungen wie z. B. der Benutzung von Schnittstellen oder der Generalisierung zusammenhängen, damit das Software-System seine Anforderungen erfüllen kann. So ist z. B. `Teilsystem_1` über die für seine Funktion erforderliche Schnittstelle `IF4` von `Teilsystem_2` abhängig, welches die Schnittstelle `IF4` realisiert. Diese Abhängigkeit wird in `Teilsystem_2` intern auf eine Schnittstelle von `Komponente_1` abgebildet. Solche Abhängigkeiten stehen im Mittelpunkt des Integrationstests, sie werden daher im Folgenden detailliert charakterisiert, modellierungstechnisch in Kapitel 5 behandelt und hinsichtlich ihrer softwaretechnischen Realisierungsmöglichkeiten in Kapitel 6 beleuchtet.

4.3 Abhängigkeiten

Abhängigkeiten zeigen, dass Bausteine zusammenspielen oder aufeinander aufbauen bzw. verweisen. Im Weiteren geht es dabei nicht nur um ausführbare Software-Bausteine, sondern ganz allgemein um alle bei der Software-Entwicklung erstellten Artefakte wie z. B. Programme, Daten und Dokumente. Diese Verallgemeinerung des Bausteinbegriffs ist wichtig, weil auch solche Artefakte zusammenspielen bzw. voneinander abhängen können, z. B. wenn ein Dokument oder ein Modell einen Querverweis auf ein anderes enthält.

4.3.1 Elementare Bausteine

Jungmayr definiert Abhängigkeiten in seiner Dissertation zum Thema Testbarkeit *(testability)* folgendermaßen:

> *A dependency is a directed relationship between two entities where changes in one entity may cause changes in the other (depending) entity.* ([Jungmayr 2003], S. 9)

Auch Savernik definiert Abhängigkeiten in diesem ganz allgemeinen Sinn:

> *Eine Abhängigkeit zwischen zwei Artefakten liegt vor, wenn Artefakt A Artefakt B zu seiner korrekten Funktionsweise benötigt.* ([Savernik 2007], S. 3)

Wichtig dabei ist, dass jede Abhängigkeit immer zwischen genau zwei Bausteinen besteht. Im Rahmen der Abhängigkeit spielen beide Bausteine unterschiedliche Rollen: Es gibt einen abhängigen und einen unabhängigen Baustein, wobei die Abhängigkeit vom abhängigen hin zum unabhängigen Baustein zeigt. Bild 4.4 (a) skizziert z. B. eine Abhängigkeit von Baustein `A` zu Baustein `B`, die als `A_B` bezeichnet ist. Der abhängige Baustein `A` benötigt den unabhängigen Baustein `B` für seine korrekte Funktionsweise. Insofern wird der unabhängige Baustein auch als „benötigter Baustein“ bezeichnet (z. B. [Web UML], [Linz 2012]).

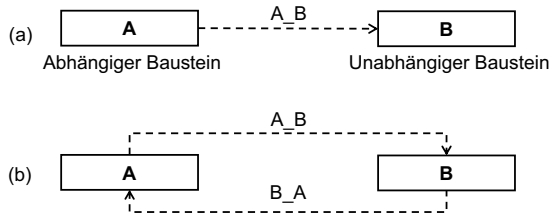

Bild 4.4 Abhängigkeiten zwischen Bausteinen

Diagramme wie in Bild 4.4 werden Abhängigkeitsgraphen genannt und beinhalten die wesentliche Information zur Ermittlung einer Integrationsstrategie, also der Reihenfolge, in der die einzelnen Bausteine integriert werden (Kapitel 10).

Für den Fall, dass sowohl Baustein `A` das Funktionieren von Baustein `B` als auch Baustein `B` das Funktionieren von Baustein `A` voraussetzt, werden beide Abhängigkeiten getrennt voneinander betrachtet. Es ergeben sich die zwei in Bild 4.4 (b) dargestellten Abhängigkeiten. Für die Abhängigkeit `A_B` ist Baustein `A` der abhängige und Baustein `B` der unabhängige Baustein, für die Abhängigkeit `B_A` verhält es sich umgekehrt. Solche gegenseitigen (zykli-

schen) Abhängigkeiten werden immer getrennt voneinander betrachtet und nicht etwa als eine (ungerichtete oder bidirektionale) Abhängigkeit. Daraus ergibt sich, dass es zwischen zwei Bausteinen nicht mehr als zwei Abhängigkeiten geben kann.

Nur höchstens zwei Abhängigkeiten zwischen zwei Bausteinen? Wie kann das ausreichen, wenn z. B. eine Klasse aufgrund der Aufrufe verschiedener Operationen von einer anderen Klasse abhängig ist?

4.3.2 Zusammengesetzte Bausteine

Die Antwort auf obigen – berechtigten! – Einwand liefert ein erneuter Blick auf die Bilder 4.2. und 4.3. Sie zeigen, dass viele Bausteine hierarchisch zusammengesetzt sind. Daher ist es oft so, dass eine Abhängigkeit zwischen zwei Bausteinen je nach Granularität der Betrachtung bzw. je nach Integrationsstufe (Abschnitt 1.5 sowie Kapitel 9) in mehrere Abhängigkeiten zwischen den jeweiligen Teilbausteinen zerfallen kann. So könnte z. B. die in Bild 4.2 gezeigte Abhängigkeit zwischen `Komponente1` und `Komponente3` in mehrere Abhängigkeiten zwischen den in diesen Komponenten enthaltenen Klassen zerfallen. Jede dieser Abhängigkeiten zwischen zwei Klassen kann dann weiter in solche zwischen den einzelnen Operationen und/oder Member-Variablen der Klassen unterteilt werden. Eine Abhängigkeit zwischen zwei Operationen kann letztendlich in mehrere Abhängigkeiten zwischen den Aufrufanweisungen der abhängigen Operation und dem „Kopf“ der aufgerufenen, unabhängigen Operation zerfasern.

Bei einer zyklischen Abhängigkeit zwischen zwei zusammengesetzten Bausteinen kann dabei der für den Integrationstest „günstige“ Fall auftreten, dass die jeweils enthaltenen Bausteine voneinander insofern unabhängig sind, dass sich der Zyklus auf Ebene der enthaltenen Bausteine zu mehreren azyklischen Abhängigkeiten auflöst. Auf diesen Sachverhalt wird u. a. in Abschnitt 10.5 eingegangen.

4.3.3 Syntaktische und semantische Abhängigkeiten

Jungmayr nennt die oben beschriebenen Abhängigkeiten i. S. v. „A benötigt B für seine korrekte Funktion“ auch semantische Abhängigkeiten. Er unterscheidet diese von syntaktischen Abhängigkeiten [Jungmayr 2003]. Letztere sind unmittelbar textuell oder grafisch im abhängigen Baustein zu erkennen, z. B. als entsprechende Namen oder Verweise. Ob eine syntaktische Abhängigkeit auch eine semantische Abhängigkeit impliziert, muss i. d. R. durch weitere Analysen entschieden werden. Wichtig ist, dass eine semantische Abhängigkeit auch ohne syntaktische Abhängigkeit vorliegen kann. Dies kann z. B. der Fall sein, wenn in einem Baustein implizite Annahmen über die Funktionsweise eines anderen Bausteins gemacht werden oder der unabhängige Baustein erst zur Laufzeit bestimmt und dynamisch geladen wird (Komponente bzw. *Plugin* oder bei Bedarf nachgeladene Bibliothek, *Dynamic Link Library*, DLL). Semantisch ist eine Abhängigkeit also immer dann, wenn sie tatsächlich etwas zur Bedeutung eines Artefakts beiträgt oder „zur Laufzeit“ bewirkt.

Welche unterschiedlichen Möglichkeiten von Abhängigkeiten es zwischen softwaretechnischen Artefakten und insbesondere Programmbausteinen geben kann, wird in Kapitel 6 näher beschrieben. Als kleiner Vorgeschmack seien hier aber schon einige Beispiele solcher Abhängigkeiten genannt:

- Ein Objekt einer Klasse `Bestellung` ruft auf einem Objekt der Klasse `Artikel` die Methode `Bestellung_prüfen()` auf (syntaktische Abhängigkeit, Interaktionsabhängigkeit).
- Eine Klasse `Spezialkundenbestellung` ist Unterklasse der Klasse `Kundenbestellung` (syntaktische Abhängigkeit, Vererbungsabhängigkeit).
- Ein Teilsystem `Warenverwaltung` und ein Teilsystem `Kundenverwaltung` greifen gemeinsam auf die gleiche Datenbank zu (semantische Abhängigkeit, datenorientierte Abhängigkeit).

Jede diese Abhängigkeitsarten birgt unterschiedliche Fehlerquellen, die zu einem Fehlerverhalten der Software führen können. So kann ein Vertauschen von übergebenen Parametern bei einer Interaktionsabhängigkeit dazu führen, dass eine Berechnung falsch ausgeführt wird. Die unterschiedlichen Fehlerarten, die im Integrationstest für die jeweiligen Abhängigkeitsarten auftreten können, werden in Kapitel 7 näher beschrieben.

4.4 Ziele, Fokus und pragmatische Definition des Integrationstests

Der Integrationstest prüft die verschiedenen Arten der Abhängigkeiten zwischen den Bausteinen. Das Ziel des Integrationstests ist es, mögliche Fehler im Zusammenspiel der Bausteine aufzudecken und – vorsichtig ausgedrückt – das Vertrauen in deren korrektes Zusammenspiel zu erhöhen [Spillner 2010]. Eickelmann und Richardson z. B. schreiben, der Integrationstest *„… insures the consistency of component interfaces and whether the components pass data and control correctly, which results in successful integration of dependent components“* ([Eickelmann 1996], Seite 65).

Das IEEE-Glossar zu System und Software Engineering definiert den Integrationstest wie folgt:

> **Integration testing:** *(1) testing in which software components, hardware components, or both are combined and tested to evaluate the interaction among them.* [Web SEVOCAB]

Mit anderen Worten: Der Integrationstest ist ein Test zwischen bereits getesteten Bausteinen bzw. Komponenten oder Modulen, um *„… Fehlerzustände in Schnittstellen und im Zusammenspiel zwischen integrierten Komponenten zu finden“* ([Spillner 2010], S. 243).

Eine wichtige Voraussetzung für den Integrationstest ist, dass die zu integrierenden Bausteine bereits isoliert geprüft sind. Boris Beizer formuliert dies so:

> *Integration testing is aimed at showing inter-element consistency under the assumption that the elements themselves satisfy element requirements and have passed element-level testing.* [Beizer 1984]

Diesen Sachverhalt verdeutlicht der Abhängigkeitsgraph in Bild 4.5. Unter der Annahme, dass die vier Bausteine A bis D bereits isoliert ihre Komponententests erfolgreich absolviert haben, konzentriert sich der (Komponenten-)Integrationstest nun auf die vier mit A_B, A_C, B_C und B_D bezeichneten Abhängigkeiten bzw. Kanten des Abhängigkeitsgraphen. Im Fokus steht dabei das auf Grundlage dieser Abhängigkeiten mögliche Zusammenspiel der Komponenten.

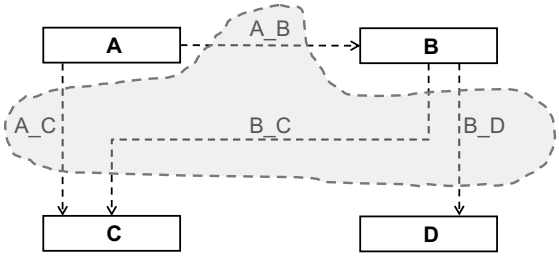

Bild 4.5 Fokus des Integrationstests

Aus diesen Überlegungen ergibt sich die folgende pragmatische Definition für den Integrationstest:

Im Integrationstest werden die Abhängigkeiten und das Zusammenspiel zwischen einzeln bereits getesteten Bausteinen eines Software-Systems getestet.

Eine der größten Herausforderungen für den Integrationstest ist die oft sehr große Anzahl von Abhängigkeiten zwischen den Bausteinen. Ein Beispiel: In der Entwicklungsumgebung Eclipse in der Version 2.0 (s. [Web Eclipse]) existierten 6747 Bausteine, zwischen denen 56765 Abhängigkeiten gefunden werden konnten (vgl. [Borner 2010]). Das sind im Durchschnitt 8 Abhängigkeiten pro Baustein. Von diesen Abhängigkeiten waren 5,28 % reine Vererbungsabhängigkeiten (ohne zusätzliche Interaktionsabhängigkeiten), 5,91 % Abhängigkeiten, die sowohl Vererbungs- als auch Interaktionseigenschaften besaßen, und 88,94 % waren reine Interaktionsabhängigkeiten. In der darauffolgenden Version (Version 2.1) stieg die Anzahl der Abhängigkeiten auf 71182.

Nur durch eine präzise Beschreibung sowohl der Testbasis als auch der Testobjekte und ein gut durchdachtes Vorgehen können solche in realen Systemen auftretenden Abhängigkeiten sinnvoll im Integrationstest überprüft werden. In Kapitel 5 werden daher Modellierung von Abhängigkeiten mit der Unified Modeling Language (UML) sowie die die Graphentheorie als mathematisches Werkzeug zur abstrakten Darstellung von Abhängigkeiten vorgestellt. Die unterschiedlichen Stufen der Integration sowie das prinzipielle Vorgehen werden im Folgenden nur kurz skizziert, ihnen widmen sich dann die Hauptkapitel des Buches.

4.5 Stufen der Integration

Weiter oben wurde bereits gezeigt, dass im Integrationstest Bausteine unterschiedlicher Granularität zu betrachten sind. Der Integrationstest ist dementsprechend auf unterschiedlichen Stufen angesiedelt, angefangen von den kleinsten Bausteinen eines Software-Systems, den Methoden und Funktionen, über Klassen/Module und Teilsysteme bis hin zu ganzen Software-Systemen, die voneinander abhängig sind.

Bashir und Paul unterscheiden für objektorientierte Software fünf mögliche Stufen der Integration [Bashir 1999]:

- Integration der Methoden einer Klasse;
- Integration zweier Klassen aufgrund Generalisierung bzw. Vererbung;
- Integration zweier Klassen aufgrund Komposition bzw. „Enthaltensein“;
- Integration mehrerer Klassen zu einer Komponente;
- Integration mehrerer Komponenten zu einem System.

Für die Benennung solcher Integrationsstufen kann die Unterscheidung zwischen den Bausteinen bzw. Eingabeartefakten der Integration und dem Ergebnis bzw. dem Ausgabeartefakt der Integration herangezogen werden.

Binder z. B. nennt Erstere die Komponenten und spricht vom Fokus *(focus)* der Integration, Letzteres bezeichnet er als (integriertes) System und nennt dies den Bereich *(scope)* der Integration (vgl. [Binder 1999], S. 627 ff.). Leider gibt er keine Bezeichnungen der resultierenden Stufen an.

Die Autoren des TMapNext®-Buches definieren die folgenden zwei Stufen des Integrationstests und benennen sie nach den jeweils zu integrierenden Bausteinen:

> **Unit Integration Test (UIT):** *A unit integration test is a test carried out by the developer in the development environment, with the aim of demonstrating that a logical group of units meets the requirements defined in the technical specifications.*
>
> **System Integration Test (SIT):** *A system integration test is a test carried out by the future user(s) in an optimally simulated production environment, with the aim of demonstrating that (sub)system interface agreements have been met, correctly interpreted and correctly implemented.* [Koomen 2008]

Auch Rex Black bezeichnet die Integrationsstufen nach den Bausteinen der Integration. Er betrachtet dabei auch die Integration mehrerer Systeme zu einem „System aus Systemen“ und schreibt:

> *„Integration testing can mean component integration testing—integrating a set of components to form a system, testing the builds throughout that process. Or it can mean system integration testing – integrating a set of systems to form a system of systems, testing the system of systems as it emerges from the conglomeration of systems.* […] *you need to keep in mind additional test levels that you might need for your projects.“* ([Black 2011], S. 4)

Insgesamt überwiegt in der Literatur eine Benennung der Integrationsstufen nach den Bausteinen bzw. Eingabeartefakten der Integration. Die in diesem Buch betrachteten vier Stufen des Integrationstests sind gemäß dieser Konvention

- der Member-Integrationstest, auf dem das Zusammenspiel der Member-Methoden und -Variablen einer Klasse bzw. eines Moduls geprüft wird,

- der Klassen- bzw. Modulintegrationstest, welcher das Zusammenspiel der Klassen bzw. Module von Komponenten oder Teilsystemen *(cluster)* prüft,
- der Komponenten- bzw. Teilsystemintegrationstest, bei dem man das Zusammenspiel der Komponenten bzw. Teilsysteme eines Systems bzw. einer Anwendung untersucht, und
- der Systemintegrationstest, in dessen Fokus das Zusammenspiel eines Systems bzw. einer Anwendung mit seiner Plattform bzw. Systemumgebung sowie das Zusammenspiel mehrerer Systeme bzw. Anwendungen in einer Anwendungslandschaft steht.

Auf die jeweiligen Besonderheiten dieser Stufen wird in Kapitel 9 ausführlich eingegangen.

An dieser Stelle werden viele Praktiker (und ebenso viele Autoren, z. B. Bashir und Paul in [Bashir 1999]) einwenden, dass das Zusammenspiel der Member-Funktionen einer Klasse in der objektorientierten Programmierung nicht zum Integrationstest, sondern eher zum Komponententest *(unit test)* gehört. In den meisten Veröffentlichungen zum Software-Test wird der Komponententest jedoch überwiegend als Test einzelner, voneinander unabhängiger Funktionen oder Prozeduren dargestellt, und auch die meisten dort behandelten Testentwurfsverfahren haben diesen Fokus. Die einzelnen Member-Funktionen und -Variablen einer Klasse sind aber gerade nicht voneinander unabhängig, sondern erfüllen gemeinsam eine gewisse Funktionalität, ja, man fordert hier ja gerade einen hohen Zusammenhalt *(high cohesion)*. Darüber hinaus ist die Generalisierung bzw. Vererbung eine spezielle Form der Abhängigkeit, die für den Integrationstest objektorientierter Software eine große Rolle spielt (s. z. B. die Abschnitte 5.1.2, 6.3 und 10.5). Wir vertreten daher die Ansicht, dass in der objektorientierten Programmierung fast alle Komponenten- bzw. Klassentests mehr oder weniger auch als Integrationstests anzusehen sind, welche auf das Zusammenspiel der Funktionen und Variablen der Klasse selbst sowie ihrer geerbten Eigenschaften fokussieren.

4.6 Vorgehen im Integrationstest

Als prinzipielle Vorgehensweise im Integrationstest wird mit dem Überprüfen des Zusammenspiels der Funktionen bzw. Methoden innerhalb einer Klasse begonnen. Anschließend werden die Abhängigkeiten zwischen Klassen und Modulen überprüft, um im Anschluss daran die entstandenen Komponenten oder Teilsysteme zu weiteren Teilsystemen oder einem vollständigen Software-System zusammenzusetzen. Das Zusammensetzen von Bausteinen zu Bausteinen höherer Abstraktionsebenen (z. B. Methoden zu Klassen) sollte stets schrittweise geschehen und idealerweise immer möglichst wenig Bausteine (idealerweise nur ein Baustein) zur Menge der bereits integrierten Bausteine hinzugefügt werden. Die Anzahl der Bausteine, die während eines Integrationsschritts zur Menge der bereits integrierten Bausteine hinzugefügt werden, definiert die Integrationsschrittgröße.

In jedem der oben beschriebenen Schritte der Integration sollte idealerweise immer nur ein Baustein (Integrationsschrittgröße = 1) hinzugefügt und die Abhängigkeiten zwischen dem neu integrierten und den bereits integrierten Bausteinen überprüft werden (vgl. z. B. [Beizer 1984], [Spillner 1990], [Binder 1999]). Das Ziel des schrittweisen Vorgehens ist es, einen gefundenen Fehler in der Abhängigkeit schnell den fehlerhaften Bausteinen zuord-

nen zu können. Dieser Fehler wird mit großer Wahrscheinlichkeit im neu hinzugefügten Baustein oder in den Bausteinen, von denen der neue Baustein abhängig ist, zu finden sein. Integriert man das gesamte System in einem Schritt, d. h. werden alle Bausteine gleichzeitig zu einem System zusammengesetzt (Big-Bang-Integration, s. Kapitel 10), können die Abhängigkeiten nur unzureichend getestet und gefundene Fehlerwirkungen schwer dem verursachenden Baustein zugeordnet werden. Die Suche nach den fehlerhaften Bausteinen wird drastisch erschwert. Zusätzlich ermöglicht es die schrittweise Integration, Testkonstellationen zu erstellen, die im gesamten Software-System mit allen Bausteinen nur mit sehr viel Aufwand (oder gar nicht) geschaffen werden können.

Dies bedeutet aber unter Umständen, dass in den ersten Integrationsschritten Bausteine, die noch nicht integriert worden sind, aber für die Lauffähigkeit bereits integrierte Bausteine benötigt werden, simuliert werden müssen. Die simulierenden Bausteine werden Platzhalter (*stub*, vgl. [Spillner 2011]) genannt und in Kapitel 17 näher beschrieben. Ebenso müssen die unabhängigen Bausteine im Test gezielt aktiviert werden, wofür ggf. sog. Treiber (driver) notwendig sind. In den nachfolgenden Integrationsschritten werden diese Platzhalter und Treiber durch die „echten“ Bausteine ersetzt. Dabei ist es wichtig, eine Integrationsreihenfolge zu finden, die möglichst wenig Platzhalter und Treiber benötigt. Diese definiert, wann welcher Baustein integriert und seine Abhängigkeiten getestet werden. Sie ergibt sich aus der gewählten Integrationsstrategie (vgl. Kapitel 10).

Um beim Integrationstest frühzeitig die wichtigen Fehler aufzudecken, sollte die Integration mit den Bausteinen und ihren Abhängigkeiten beginnen, die ein hohes Risiko in sich tragen. Dies sind Bausteine, die eine höhere Wahrscheinlichkeit besitzen, fehlzuschlagen und/oder die im Fehlerfall zu weitreichenden Schäden (Personenschäden, finanzielle Schäden …) führen. Auch Eickelmann et al. und Orso weisen darauf hin, die Integrationsreihenfolge so zu wählen, dass kritische Bausteine möglichst früh integriert werden ([Eickelmann 1996], [Orso 1998]).

Zum Aufdecken der Fehler stehen zwei unterschiedliche Verfahren zur Verfügung, welche sich hervorragend ergänzen. Zum einen können Fehler über die statische Analyse (vgl. Kapitel 12) und zum anderen mithilfe von dynamischen Tests (vgl. Kapitel 13 – 15) aufgedeckt werden.

Als statische Analysen werden Verfahren zur automatisierten Prüfung von Bausteinen und ihren Abhängigkeiten bezeichnet, welche die Bausteine unmittelbar untersuchen, ohne sie auszuführen. Statische Analysen können sowohl manuell als auch automatisiert durchgeführt werden. Im automatisierten Fall analysiert ein Werkzeug, z. B. ein Parser oder Scanner, den Text und registriert mögliche syntaktische Fehler und Regelverletzungen. Solche Analysen können u. a. fehlerhafte Parametertypen aufdecken, z. B. wenn ein Parameter vom Typ `char` erwartet, aber ein Parameter vom Typ `int` übergeben wird. Auch mögliche Fehler beim Zugriff auf nicht initialisierte Parameter bzw. Variablen sind hiermit aufdeckbar. Jedoch lassen sich über die statische Analyse nicht alle Fehler in den Abhängigkeiten aufdecken.

In vielen Fällen ist es notwendig, die Bausteine und ihre Abhängigkeiten auszuführen um das Fehlerverhalten aufzudecken. Um die Ausführung strukturiert und systematisch durchführen zu können, werden Testentwurfsverfahren angewendet. Diese erlauben es, mithilfe der Informationen über die Bausteine und ihre Abhängigkeiten Testfälle abzuleiten, um das Zusammenspiel systematisch testen zu können. Beispielsweise können Anwendungsfälle

(use cases), die das Verhalten der Software aus Sicht des Anwenders beschreiben, verwendet werden, um mögliche Eingabesequenzen für das System zu ermitteln. Parallel dazu werden die beteiligten Bausteine, die durchlaufenen und betroffenen Abhängigkeiten und die erwarteten Reaktionen der Bausteine identifiziert. Anhand dieser Informationen können die Testfälle beschrieben werden. Diese Testfälle enthalten Informationen über die zu verwendenden Testdaten, die aufzurufenden Dienste, das erwartete Verhalten der Bausteine und deren Abhängigkeiten sowie Hinweise über die zu beobachtenden Systemreaktionen. Hierbei spielen jedoch nicht nur die funktionalen Aspekte der Software eine Rolle, sondern auch ihre nicht-funktionalen Aspekte wie z. B. Performance oder Security (vgl. Kapitel 16).

Den Integrationstest bzw. die Integrationstests auf den unterschiedlichen Stufen zu organisieren, zu planen und durchzuführen, bedarf eines wohldefinierten Prozesses. Darin wird festgelegt, welche Aufgaben durch welche Rollen durchzuführen sind und welche Ergebnisse dabei erstellt werden. Einen groben Überblick gibt der oben in Abschnitt 3.7 skizzierte allgemeine Testprozess. Welche einzelnen Aufgaben diese Aktivitäten umfassen, wird in Kapitel 11 näher beschrieben.