Testverfahren und Strategien zum Integrationstest

Mario Winter

Institut für Informatik
Fachhochschule Köln, Campus Gummersbach
Steinmüllerallee 1
51643 Gummersbach
mario.winter@fh-koeln.de

Abstract: Der Integrationstest prüft das Zusammenspiel der Bausteine eines Softwaresystems. Hierzu sind Testfälle zu entwerfen, deren Eingaben die zu integrierenden Bausteine stimulieren und deren erwartete Ergebnisse die Interaktionen zwischen diesen Bausteinen sowie deren Parameter- und Rückgabewerte umfassen. Dabei bestimmt die gewählte Integrationsstrategie die Reihenfolge, in der die Bausteine integriert und in ihrem Zusammenspiel getestet werden. Zugunsten einer einfacheren Fehlerlokalisierung fügen schrittweise Integrationsstrategien immer nur eine begrenzte Anzahl weiterer Bausteine zur Menge bereits integrierter und integrationsgetesteter Bausteine hinzu. Der Beitrag skizziert relevante Testentwurfsverfahren sowie Verfahren zur Ermittlung von Integrationsstrategien.

Schlüsselworte: Test, Integrationstest, Integrationsstufen, Integrationsstrategie, Testentwurfsverfahren

1. Einleitung

Testen dient der Bewertung eines Softwareprodukts und dazugehöriger Arbeitsergebnisse mit dem Ziel, deren Anforderungserfüllung und Eignung festzustellen sowie ggf. Defekte zu finden. Je nach der „Konstruktionsphase“, in der die Arbeitsergebnisse angelegt wurden, und der Granularität der jeweiligen Testobjekte ergeben sich die korrespondierenden Teststufen Modul- bzw. Komponententest (unit test), Integrationstest, Systemtest und Abnahmetest (vgl. [GTB10]). Der Integrationstest ist klassischerweise als Teststufe zwischen Modul- bzw. Komponententest und Systemtest gelagert. Dabei werden Klassen, Module, Komponenten, Teilsysteme oder auch ganze Systeme zusammengefügt und in ihrem Zusammenspiel geprüft. Zur Vermeidung von Begriffskonflikten wird im Weiteren der allgemeine Begriff „Baustein“ verwendet. Für den Integrationstest sind Testfälle zu entwerfen, deren Eingaben die zu integrierenden Bausteine stimulieren und deren erwartete Ergebnisse die Interaktionen zwischen diesen Bausteinen sowie deren Parameter- und Rückgabewerte umfassen. Hierbei bestimmt die gewählte Integrationsstrategie die Reihenfolge, in der die Bausteine integriert und in ihrem Zusammenspiel getestet werden.

Der Beitrag ist wie folgt strukturiert. Kapitel 2 skizziert wichtige Testverfahren für den Integrationstest. In Kapitel 3 werden verschiedene Integrationsstrategien sowie Verfahren zur Ermittlung von Integrationsreihenfolgen beschrieben.

2. Testverfahren für den Integrationstest

Testverfahren für den Integrationstest helfen bei der Definition und Auswahl von Testfällen, mit denen Fehler im Zusammenspiel zwischen den Bausteinen eines Programms gefunden werden können. Man unterscheidet hierbei funktions- und wertebezogene , ablaufbezogene sowie fehlerbezogene, erfahrungsbasierte und weitere Testentwurfsverfahren.

2.1 Funktions- und wertebezogene Testentwurfsverfahren

Funktions- und wertebezogene Testentwurfsverfahren für den Integrationstest adressieren vornehmlich funktionale Anforderungen. Anwendungsfallbasiertes Integrationstesten prüft das Zusammenspiel der Bausteine im Rahmen aktueller Abläufe vollständiger, in sich abgeschlossener Funktionalitäten. Parameterbasiertes Testen richtet das Augenmerk auf die zwischen den Bausteinen ausgetauschten Daten. Wertebezogenes Testen betrachtet die Wertebereiche z. B. von Member-Variablen und Operationsparametern. Erstere tragen in ihrer Gesamtheit zum Zustand eines Bausteins bei und können dementsprechend auch „im Verbund“, also auf dem aktuellen Zustand basierend, zum Test herangezogen werden. Letztere erlauben es dabei, bestimmte Abläufe und Ausnahmebehandlungen zu erzwingen. Darüber hinaus stehen die strukturellen Abhängigkeiten der Bausteine im Fokus funktions- und wertebezogener Tests. So können Bausteine z. B. über mehr oder weniger lange Zeiträume miteinander verbunden sein.

Die Art und konkrete Anzahl erlaubter Verbindungen steht im Mittelpunkt assoziationsbasierten Testens. Letztendlich rufen abhängige Bausteine oft Operationen unabhängiger Bausteine auf. Sind hierbei im Rahmen von Generalisierungsbeziehungen bzw. „Vererbung“ auch dynamisch gebundene Operationsaufrufe möglich, so führt dies oft zu explosionsartig steigenden Anzahlen möglicher Testfälle. Solchen Situationen kann mit dem generalisierungsbasierten und paarweisen Testen begegnet werden.

2.2 Ablaufbezogene Testentwurfsverfahren

Die zu den White-Box-Verfahren zählenden Verfahren des ablaufbezogenen Testens versuchen, aus der der Vielzahl möglicher Abläufe und Datenflüsse zwischen Bausteinen solche zu extrahieren, welche mögliche Fehlerwirkungen in den Aufrufreihenfolgen oder Parameter- bzw. Datenverwendungen aufdecken.

Die naheliegende Idee, ablaufbezogene Testentwurfsverfahren aus dem Komponententest durch einfache Verknüpfung z. B. der Ablaufbeschreibungen von abhängigen und unabhängigen Bausteinen auf den Integrationstest zu übertragen, schlägt aus mehreren Gründen fehl. Einerseits reicht nämlich eine Menge von Testfällen, mit denen ein bestimmtes Testendekriterium bzw. Ausgangskriterium für die Teilbausteine erreicht wurde, i. d. R. nicht dazu aus, auch den zusammengesetzten Baustein adäquat hinsichtlich dieses Kriteriums zu testen. Diese Beobachtung ist auch als sogenanntes Antikompositionsaxiom bekannt und wurde zusammen mit einigen anderen solcher „Axiome" bzw. Eigenschaften von Ausgangskriterien von Elaine Weyuker angegeben ([We86]). Andererseits steigt der Testaufwand bei einem naiven, herkömmliche White-Box-Testverfahren unverändert übernehmenden Ansatz im Integrationstest überproportional an. Während er sich im Komponententest lediglich durch Addition der Einzelaufwände für die Bausteine ergibt, müssten diese im Integrationstest miteinander multipliziert werden.

Die ablaufbezogenen Verfahren des Integrationstests versuchen daher, den Aufwand durch geschickte Eingrenzung der zu testenden Ablaufmöglichkeiten zu minimieren. Das Ziel kontrollflussbasierter Integrationstests ist die Prüfung, ob abhängige Bausteine die spezifizierte Aufrufreihenfolge von Operationen der Schnittstellen unabhängiger Bausteine einhalten. Spillner beschreibt dies in seiner Dissertation zum Integrationstest folgendermaßen:

Beim ablaufbezogenen kontrollflussbasierten Integrationstest wird ausgehend von der Spezifikation versucht, fehlerhafte Aufrufreihenfolgen externer Operationen festzustellen und betreffende Programmteile zur Ausführung zu bringen oder nachzuweisen, dass eine Aufrufreihenfolge nicht ausführbar ist. ([Sp90])

Das datenflussbasierte Testen untersucht die möglichen Nutzungen von Daten innerhalb von Programmen bzw. einzelnen Operationen. Im Integrationstest werden datenflussbasierte Tests in erster Linie dazu verwendet, Fehler aufgrund inkorrekter Datennutzungen zwischen den zu integrierenden Bausteinen zu finden. Ziel ist die Prüfung, ob abhängige Bausteine die spezifizierten Verwendungen von Daten und Operationsparametern der Schnittstellen unabhängiger Bausteine einhalten sowie Letztere diese Parameter richtig interpretieren und die korrekten Rückgabewerte liefern. Die Aufgaben des datenflussbasierten Tests fasst Spillner folgendermaßen zusammen:

Die Parameterverwendungen an den Schnittstellen externer Operationen sind beim Integrationstest zu untersuchen. Dabei wird versucht, anomalieträchtige Verwendungen zu entdecken. Dies sind beispielsweise die Nichtverwendung eines Parameterwertes in der aufgerufenen externen Operation oder die Nichtbelegung eines Parameterwertes und dessen lesende Verwendung innerhalb der Operation. Diese Fehler sind beim Modultest nicht nachweisbar. ([Sp90])

Das Ziel des interaktionsbasierten Testens ist es, Fehler im Zusammenspiel einer Menge eng zusammenarbeitender Bausteine aufzudecken. Hierbei spielen die möglichen Reihenfolgen von Operationsaufrufen bzw. Nachrichten eine Rolle. Der Nutzen besteht darin, die Erfüllung der mit Interaktionsdiagrammen vorgegebenen Spezifikationen durch die Klassen eines Pakets zu prüfen. Ein Vorteil des Prüfens von Aufrufsequenzen besteht in dem geringeren Aufwand bei der Erstellung der Testumgebung sowie der Testfälle und Testdaten, da ein „Aufruf" eine ganze Reihe von Operationen prüfen kann ([Jo94]). Daneben werden nicht nur die einzelnen Operationen, sondern direkt auch ihr Zusammenspiel geprüft.

2.3 Fehlerbezogene, erfahrungsbasierte und weitere Testentwurfsverfahren

Fehlerbezogene und erfahrungsbasierte Testentwurfsverfahren setzen weniger auf systematische Verfahren zum Ableiten von Testfällen, sondern beziehen die explizit dokumentierten Erfahrungen und das implizite Wissen der Tester beim Testen mit ein. Grundlegend können hierbei zwei Verfahren unterschieden werden: Error Guessing und exploratives Testen.

Im Rahmen des fehlerbezogenen Testens (error guessing) werden die Testfälle auf Basis von möglichen Fehlern bzw. Fehlerklassen, die im Rahmen der Funktionalität auftreten können, abgeleitet. Hierbei werden die Testfälle so gewählt, dass sie bestimmte Fehler bzw. Fehlerklassen ausschließen bzw. diese Art von Fehler aufdecken würden.

Das explorative Testen (exploratory testing) nutzt die Intuition der ausführenden Tester, um Fehler im Software-System aufzudecken. Um den Tester nicht „irgendetwas" testen zu lassen, werden ihm Checklisten oder eine Test-Charta an die Hand gegeben, um ihm die funktionale Richtung, in die seine Tests gehen sollen, grob vorzugeben. Jedoch werden keine detaillierten Testfälle definiert, sondern nur grob das Testziel. Wie dieses Ziel erreicht wird, bleibt dem Tester bei der Testdurchführung selbst überlassen.

Neben den bislang betrachteten systematischen oder erfahrungsbasierten Testentwurfsverfahren werden auch im Integrationstest zur Generierung von Testfällen bzw. Testdaten heuristische Verfahren wie der Zufallstest oder der suchbasierte Test angewendet. Das in neuester Zeit vorgestellte konkolische Testen (concolic testing, [Se07]) versucht, durch die geschickte Verknüpfung statischer Analysen und symbolischer Programmausführungen mit den Ergebnissen dynamischer Analysen wie z. B. durchlaufener Pfade des Programmcodes oder realisierter Werte von Variablen und/oder Parametern, die oben skizzierten heuristischen Verfahren noch zielgerichteter zu steuern. Letztendlich kann auch im Integrationstest die Bestimmung der Testgüte und Optimierung der Tests mit Verfahren des Mutationstests erfolgen (z. B. [Vi01], [Be09]).

3. Integrationsstrategien und -reihenfolgen

Die Integrationsstrategie bestimmt die Reihenfolge, in der die Bausteine integriert und in ihrem Zusammenspiel getestet werden. Man unterscheidet schrittweise Integrationsstrategien von der sog. „Big-Bang-Strategie",

welche alle Bausteine eines Systems auf einmal integriert und dann (i.d.R. mit Systemtests) testet.

3.1 Schrittweise Integrationsstrategien

Zugunsten einer einfacheren Fehlerlokalisierung fügen schrittweise Integrationsstrategien immer nur eine begrenzte Anzahl weiterer Bausteine zur Menge bereits integrierter und integrationsgetesteter Bausteine hinzu. Weist hierbei ein Baustein eine Abhängigkeit zu einem noch nicht in dieser Menge befindlichen Baustein auf, wird letzterer durch einen extra für den Test zu erstellenden Stellvertreter (*stub*) ersetzt. Dieser stellt für die im Test geprüfte Nutzung eine rudimentäre Funktionalität sowie ggf. Protokollierungs- und Profiling-Funktionen bereit. Wird ein neu hinzugenommener Baustein von noch nicht integrierten Bausteinen genutzt, so sind letztere durch Treiber (*driver*) zu vertreten, wenn Testfälle für deren Nutzung des aktuell integrierten Bausteins notwendig sind. Dies ist dann der Fall, wenn der zu integrierende Baustein im Rahmen dieser Nutzungen seinerseits weitere Bausteine nutzt.

Schrittweise Integrationsstrategien lassen sich weiter unterteilen in von der Softwarestruktur abhängige und von ihr unabhängige Strategien. Für strukturabhängige Strategien wird oft der Abhängigkeitsgraph herangezogen, dessen Knoten die Bausteine und dessen Kanten ihre Abhängigkeiten darstellen. Prominente Vertreter sind z.B. die Bottom-Up und die Top-Down Strategie, welche von vollkommen unabhängigen (also nur „genutzten“) Bausteinen hin zu vollkommen abhängigen (also nur „nutzenden“) Bausteinen bzw. umgekehrt integrieren. Als Mischformen sind auch die Inside-Out und die Outside-In bzw. Sandwich-Strategie bekannt. Strukturunabhängige Strategien wählen den bzw. die als nächstes zu integrierenden Bausteine z.B. nach deren Risiko, Komplexität oder Verfügbarkeit aus.

Weist ein zu integrierender Baustein eine Nutzungs- oder eine Generalisierungs-Abhängigkeit zu einem noch nicht integrierten Baustein auf, ist letzterer durch einen Test-Stellvertreter zu ersetzen. Kann ein neu hinzugenommener Baustein nicht über bereits integrierte Bausteine angesteuert werden, so ist für jede entsprechende Aufruf-Abhängigkeit ein Treiber zu erstellen. Zusätzlich fällt immer ein Treiber für die Ansteuerung des Systems nach dem letzten Integrationsschritt an.

Die Aufwände hierfür lassen sich in drei Klassen einteilen. $KS(i, j)$ bzw. $KG(i, j)$ beziffern den Aufwand, wenn für eine Nutzungs- bzw. Generalisierungs-Abhängigkeit von Baustein i zu Baustein j ein Stellvertreter als Ersatz für j erstellt werden muss. $KD(i, j)$ gibt den Aufwand an, für eine Abhängigkeit von i nach j anstelle des aufrufenden Bausteins i einen Treiber einzusetzen. Für eine konkrete Integrationsreihenfolge fallen diese Aufwände natürlich nur dann an, wenn Baustein i also bzgl. o vor Baustein j integriert wird.

Für den in Abbildung 2 gezeigten einfachen „generalisierungsfreien“ Abhängigkeitsgraphen sind z.B. $KS(1, 2) = 2$, $KD(1, 2) = 0$ und $KD(2, 1) = 1$. Die „Kosten“ für den ersten Schritt der „Top-Down“ Integrationsreihenfolge $o_{TD} = <1, 2, 3, 4, 5, 6, 7>$ betragen 7 (ein Treiber für Komponente 1 und drei Stellvertreter für die Komponenten 2 bis 4, also 1+3*2=7). Als Gesamtkosten ergeben sich $K(o_{TD}) = 19$. Besser ist die „Bottom-Up“ Reihenfolge $o_{BU} = <7, 6, 5, 4, 3, 2, 1>$ mit den Gesamtkosten $K(o_{BU}) = 10$.

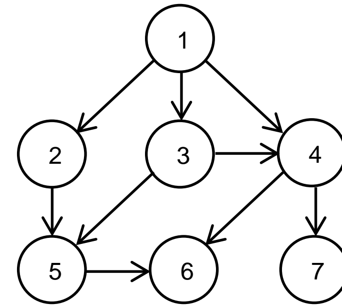

Abbildung 1: Einfacher (azyklischer) Abhängigkeitsgraph

Die Arbeiten zur Ermittlung von (schrittweisen) Integrationsreihenfolgen lassen sich grob in solche klassifizieren, die auf deterministischen graph-basierten Verfahren aufbauen, und solche, die heuristische Ansätze wie z.B. Suchverfahren oder genetische Algorithmen verwenden. Aus Platzgründen werden nur einige wenige repräsentative Arbeiten näher beleuchtet, jeweils stellvertretend für ihre Klasse. Umfassendere Ausführungen hierzu finden sich z.B. in [BLW03], [Ba09], [BP09] oder [Wi12] und [Wi13].

3.2 Graph-basierte Ansätze

In einer der ersten Arbeiten zum Integrationstest für objektorientierte Software gibt Overbeck ein Verfahren zur Ermittlung einer Integrationsstrategie auf der Grundlage des Klassendiagramms an ([Ov94]). Das Verfahren orientiert sich primär an den Generalisierungsbeziehungen Top-Down, also von Basisklassen zu abgeleiteten Klassen, und sekundär an den Interaktions- bzw. Nutzungsabhängigkeiten. Die mit diesem Verfahren ermittelten Integrationsstrategien minimieren die Anzahl der für den Test benötigten Testtreiber und -stellvertreter. Probleme, die aus zyklischen Nutzungsbeziehungen resultieren, sollen durch „Aufbrechen“ der Zyklen anhand einer Tiefensuche mit entsprechenden Teststellvertretern behandelt werden. Ebenso durch Aufbrechen zyklischer Abhängigkeiten und nachfolgende topologische Sortierung wird die Integrationsreihenfolge von Kung et Al. ermittelt ([Ku95]).

[Je99] und [Wi00] stellen unabhängig voneinander eine Integrationsstrategie vor, welche die Betrachtung von ganzen Klassen auf deren konstituierende Bausteine verfeinert. Die Knoten der resultierenden Abhängigkeitsgraphen modellieren also Member der Klassen (Methoden, in [Wi00] auch Variablen), die Kanten stellen Aufrufe zwischen Operationen dar, in [Wi00] darüber hinaus auch Redefinitionen von Operationen sowie die Verwendung der Member-Variablen (def, use). Die feingranulare Betrachtung führt dazu, dass viele auf Klassenebene zu beobachtende zyklischen Abhängigkeiten auf der Ebene von Methoden in azyklische Teilgraphen bzw. Wälder zerfallen und somit oft eine „kanonische“ Integrationsreihenfolge gefunden werden kann.

Abdurazik und Offutt beschreiben ein graph-basiertes Verfahren, das sowohl die Knoten als auch die Kanten des Abhängigkeitsgraphens mit Gewichten belegt, welche durch unterschiedliche Code-Metriken ermittelt werden ([AO09]).

Unter Berücksichtigung beider Arten von Gewichten erreichen sie in den Experimenten bessere Ergebnisse als vergleichbare Arbeiten.

3.3 Such-basierte Ansätze

Le Hanh et Al. sowie Briand et Al. experimentieren mit evolutionären Algorithmen zur Ermittlung optimaler Integrationsreihenfolgen ([Le01], [BFL02]). Als primäres Ziel- bzw. „Fitness“-Kriterium wird die Kopplungs-Komplexität zwischen Klassen sowie – in [Le01] – auch zwischen Methoden betrachtet. [Le01] verwenden als sekundäres Zielkriterium auch die Komplexität der einzelnen Klassen selbst. Experimentelle Vergleiche verschiedener evolutionärer und graph-basierter Algorithmen ergaben für erstere tw. bessere Ergebnisse, während letztere besser für große Systeme skalierten.

Assunção et Al. betrachten die verschiedenen Arten von Abhängigkeiten separat und formulieren ein Optimierungsproblem mit mehrfacher Zielsetzung ([As11]). Zur Lösung verwenden sie evolutionäre Mehrziel-Optimier-Algorithmen. Experimente mit zwei unterschiedlichen Algorithmen hinsichtlich vier Zielkriterien (Attribute, Methoden, Anzahl unterschiedlicher Typen der Parameter und Rückgabewerte) zeigen die Eignung des Ansatzes anhand vier realer Systeme.

Literaturverzeichnis

[AO09] Abdurazik, A. & Offutt, J.: Using Coupling-Based Weights for the Class Integration and Test Order Problem. Comput. J., Oxford University Press, 2009; S. 557-570.

[As11] Assunção, W. K. G.; Colanzi, T. E.; Pozo, A. T. R. & Vergilio, S. R.: Establishing integration test orders of classes with several coupling measures. Proc. 13th Conf. on Genetic and evolutionary computation, ACM, 2011; S. 1867-1874.

[Ba09] Bansal, P.; Sabharwal, S. & Sidhu, P.: An investigation of strategies for finding test order during integration testing of object-oriented applications. Proc. Int. Conf. on Methods and Models in Computer Science (ICM2CS 09), 2009; S. 1-8.

[Be09] Belli, F., Hollmann, A., Padberg, S., Communication Sequence Graphs for Mutation-Oriented Integration Testing, Proc. SSIRI, 2009, S. 387–392.

[BFL02] Briand, L. C.; Feng, J. & Labiche, Y.: Experimenting with Genetic Algorithms to Devise Optimal Integration Test Orders. Technical Report SCE-02-03, Carleton University, 2002.

[BLW03] Briand, L. C.; Labiche, Y. & Wang, Y.: An Investigation of Graph-Based Class Integration Test Order Strategies. IEEE Transactions on Software Engineering, Vol. 29, IEEE Computer Society, 2003; S. 594-607.

[BP09] Borner, L. & Paech, B.: Integration Test Order Strategies to Consider Test Focus and Simulation Effort. Proc. 1st Int. Conf. on Advances in System Testing and Validation Lifecycle (VALID 09), 2009; S. 80-85.

[GTB10] ISTQB/GTB Standardglossar der Testbegriffe, Version 2.1, Deutsch/Englisch, ISTQB/GTB, 2010.

[Je99] Jeron, T.; Jezequel, J.-M.; Le Traon, Y. & Morel, P.: Efficient strategies for integration and regression testing of OO systems. Proc. 10th Int. Symp. onSoftware Reliability Engineering, IEEE, 1999; S. 260-269.

[Jo94] Jorgensen, P. C., Erickson, C., Object-Oriented Integration Testing, Commun. ACM, Vol. 37, ACM, 1994, S. 30 – 38.

[Ku95] Kung, D., Gao, J., Hsia, P., Toyoshima, Y. & Chen, C.: A test strategy for object-oriented programs. Proc. 19th Computer Software and Applications Conf. (COMPSAC 95), IEEE Computer Society Press, 1995; S. 239-244.

[Le01] Le Hanh, V.; Akif, K.; Le Traon, Y. & Jézéque, J.-M.: Selecting an Efficient OO Integration Testing Strategy: An Experimental Comparison of Actual Strategies. In: Knudsen, J. (Ed.): Proc. ECOOP 2001, LNCS 2072, Springer, Berlin & Heidelberg, 2001; S. 381-401.

[Ov94] Overbeck, J.: Integration Testing for Object-Oriented Software. PhD Dissertation, Vienna University of Technology, 1994.

[Se07] Sen, K., Concolic testing, Proc. 22. IEEE/ACM Int. Conf. on Automated Software Engineering, ACM, 2007, S. 571 – 572.

[Sp90] Spillner, A., Dynamischer Integrationstest modularer Softwaresysteme, Dissertation,Universität Bremen, 1990.

[Ve10] Vergilio, S.; Pozo, A.; Arias, J.; da Veiga Cabral, R. & Nobre, T.: Multi-objective optimization algorithms applied to the class integration and test order problem. International Journal on Software Tools for Technology Transfer (STTT), Springer Berlin & Heidelberg, 2010; S. 1-15.

[Vi01] Vincenzi, A. M. R., Maldonado, J. C., Barbosa, E. F., Delamaro, M. E., Unit and integration testing strategies for C programs using mutation, Software Testing, Verification and Reliability, Vol. 11, 2001, S. 249 – 68.

[We86] Weyuker, E. J., Axiomatizing Software Test Data Adequacy, IEEE Transactions on Software Engineering, Vol. SE-12, 1986, 1128 – 1138.

[Wi00] Winter, M.: Ein interaktionsbasiertes Modell für den objektorientierten Integrations- und Regressionstest. Informatik - Forschung und Entwicklung, Bd. 15, Springer Berlin / Heidelberg, 2000; S. 121-132.

[Wi12] Winter, M.; Ekssir-Monfared, M.; Sneed, H.M.; Seidl, R.; Borner; L.: Der Integrationstest – Von Entwurf und Architektur zur Komponenten- und Systemintegration. Hanser Verlag, München, 2012.

[Wi13] Winter, M.: Optimale Integrationsreihenfolgen. Erscheint in: Proc. GI Software Engineering 2013, LNI, Aachen, 2013.