Reviews in der objekt-orientierten Softwareentwicklung

Mario Winter
Fachbereich Informatik
FernUniversität Hagen
e-Mail: mario.winter@fernuni-hagen.de

TAV 10, 6.-7.03.97, GADV / SBS Böblingen

Abstrakt: Bestimmte Eigenschaften objekt-orientierter Software stellen die Anwendbarkeit herkömmlicher Reviewtechniken in Frage. Dieser Beitrag stellt zuerst die bekanntesten Review-Techniken vor und gibt dann Techniken und Kriterien für Reviews objekt-orientierter Entwicklungsprodukte an. Den Abschluß bilden Anmerkungen zu unterstützenden Werkzeugen und zur Einführung von Reviews im Entwicklungsprozeß.

Keywords: Review, Inspektion, Walk-Through, Szenario, Analyse- und Entwurfsmuster, Werkzeuge

1 Einleitung

Angesichts der Tatsache, daß Computer - genauer gesagt die ablaufenden Programme - in immer mehr Bereichen zur einzig kontrollierenden Instanz werden, ist die Frage nach der Qualität der eingesetzten Software aktueller denn je. Für Qualitätssicherung in der Software-Entwicklung haben sich Reviews, hier besonders die von Fagan im Jahre 1976 eingeführten Inspektionen [Fagan76], als eines der wirkungsvollsten Mittel für die erwiesen [WheeBrykMees96]. Trotz ihrer proklamierten Unabhängigkeit vom zu inspizierenden Produkt bzw. vom der Entwicklung zugrundeliegenden Paradigma gibt es bis heute jedoch entgegengesetzt zu der Tatsache, das seit über zehn Jahren objekt-orientierte Software auch industriell entwickelt wird, keine Veröffentlichungen über Erfahrungen mit Inspektionen in der objekt-orientierten Software-Entwicklung [MacdEtAl96].

Schwierigkeiten bei der Inspektion objekt-orientierter Software mit herkömmlichen Inspektionstechniken resultieren hauptsächlich aus den folgenden beiden Punkten:

- Die Verlagerung der Komplexität von den einzelnen strukturellen Einheiten (Klassen, Dienste) in deren Beziehungsgeflecht „verschmiert“ bzw. „delokalisiert“ die zugrundeliegenden Konzepte.
- Durch die zusätzlichen Freiheitsgrade aufgrund des mit der Vererbung verbundenen Polymorphismus zerfasert die Struktur weiter und es besteht kaum noch eine Übereinstimmung zwischen der graphisch bzw. textuell repräsentierten Struktur und dem daraus resultierenden dynamischen Verhalten.

Im nächsten Abschnitt stellen wir die wichtigsten Review-Techniken vor. Im dritten Abschnitt geben wir dann Techniken und Kriterien für Reviews objekt-orientierter Analysen und Entwürfe an. Danach folgen Anmerkungen zur Inspektion objekt-orientierten Quellcodes. Zum Abschluß unserer Betrachtungen kommen wir noch auf unterstützende Werkzeuge zu sprechen und geben einige Tips zur Einführung von Reviews im Entwicklungsprozeß.

2 Reviews

Reviews gehören zu den manuellen, statischen Verfahren zur Prüfung von Entwicklungsprodukten bzw. ihren unterschiedlichen Repräsentationen. Sie leiten sich aus der „*Have a close look at the code*“-Vorgehensweise ab, mit der normalerweise jeder Programmierer seinen Code überprüfen sollte. Aus der freiwilligen, unstrukturierten Betrachtung von Code wird ein präzise definierter Vorgang, bei dem der zeitliche und inhaltliche Ablauf sowie die Anzahl und Qualifikationen der beteiligten Personen genau festgelegt sind und der nicht auf die Prüfung des Quelltextes beschränkt bleibt.

Reviews sollen Defekte im Entwicklungsprodukt aufzeigen, wobei wir unter einem *Defekt* die zu erwartende Abwesenheit eines Qualitätsmerkmals wie z.B. Korrektheit, Benutzerfreundlichkeit, Robustheit, Wartbarkeit oder Wiederverwendbarkeit verstehen. In Abhängigkeit von der beteiligten Personengruppe, den Defekten, die das Review aufdecken soll, ihrer Kritikalität und der des zu prüfenden Produktes werden verschiedene Review-Techniken eingesetzt; am weitesten verbreitet sind das persönliche Review, der Walk-Through und die Inspektion.

Persönliche Reviews werden im Laufe der Entwicklung mit dem Ziel der Validierung und Verbesserung des Produktes durchgeführt. Möglichkeiten hierzu sind die „Ausführung von

Hand“, bei welcher der Entwickler ein Programm (-stück) auf dem Papier durchspielt und dabei die Werte von Variablen etc. tabellarisch nachhält, oder das Prüfen des Codes anhand persönlicher Check- und Fehlerlisten. Mit gutem Erfolg können die Mitarbeiter einer Projektgruppe ihre Produkte auch in gegenseitigen persönlichen Reviews prüfen.

Als *Walk-Through* bezeichnen wir eine Zusammenkunft mehrerer Personen mit meistens unterschiedlichen Aufgaben und Kenntnissen. Der Entwickler stellt das Produkt vor und erklärt es anhand einiger durchzuspielender Szenarien, wofür einige nicht zu komplizierte Testfälle vorbereitet werden. Ziel ist die Diskussion der Entwurfsentscheidungen und die Vermittlung von Entwurfswissen zum tieferen Verständnis des Produkts. Die Testfälle selbst spielen dabei keine kritische Rolle; sie dienen eher dazu, den Entwickler gezielt zu Entscheidungen (z.B. zur Programmlogik) zu befragen. In den meisten Fällen werden mehr Defekte in der Diskussion mit dem Entwickler als durch die Testfälle selbst entdeckt.

Eine *Inspektion* ist eine von mehreren Personen nach reglementierten Schritten durchgeführte, dokumentierte Prüfung eines Produktes gegen eine begrenzte Menge von Kriterien. Das Inspektionsteam besteht gewöhnlich aus vier bis acht Personen. Der Moderator, der ein erfahrener Softwareentwickler sein sollte, ist leitet hauptverantwortlich die Inspektionssitzung und protokolliert alle gefundenen Defekte. Außerdem muß er sicherstellen, daß anschließend alle während der Inspektion gefundenen Defekte behoben werden. Die weiteren Teilnehmer sind der Autor des Dokuments, der Programmdesigner (wenn es sich um ein Codedokument handelt) und ein Testspezialist. Eine Inspektion läuft in den folgenden vier Phasen ab.

1. **Überblick.** Der Moderator verteilt das zu prüfende Dokument inkl. sämtlicher Vorgänger-dokumente, die bindende und damit zu testende Vorgaben enthalten (z.B. die Spezifikation des codierten Moduls), an die Teilnehmer. (Für nicht mit dem Produkt vertraute Teilnehmer kann optional noch eine kurzer Informationsvortrag über das Produkt und das zu prüfende Dokument gegeben werden.)
2. **Vorbereitung.** Die Teilnehmer setzen sich intensiv mit dem Stoff auseinander.
3. **Sitzung.** Der Autor erläutert sein Dokument Zeile für Zeile. Alle in der Diskussion erkannten Defekte werden protokolliert, jedoch nicht korrigiert. Als Hilfsmittel dienen dabei z.B. Checklisten möglicher Defekte.[1]
4. **Nachbereitung.** In dieser Phase werden die gefundenen Defekte behoben.

Die Sitzung sollte nicht länger als 120 Minuten dauern, da danach die Konzentrationsfähigkeit der Teilnehmer und damit die Produktivität zu stark nachläßt. In dieser Zeit können ca. 250-300 Anweisungen Quelltext inspiziert werden.

Nach Parnas kann der Inspektionserfolg dadurch verbessert werden, daß unterschiedlich qualifizierte Reviewer (z.B. Benutzer, Entwickler, Applikationsspezialist...) Teile des Produkts anhand verschiedener, defektspezifischer Techniken untersuchen und dann „Fragebögen“ über den inspizierten Teil ausfüllen (*active design reviews*). In mehreren kleineren Treffen der Entwickler mit jeweils einem Reviewer werden die aufgrund der Fragebögen aufgeworfenen Punkte weiter diskutiert und so ein umfassendes Protokoll erstellt [ParnasWeiss85].

Eine weitere Verbesserung durch die Unterteilung der Reviews in kleine, aufeinander aufbauende Schritte (Phasen) schlagen Knight und Myers mit den *phased inspections* vor [KnightMyers93]. Gefundene Defekte werden nach jeder Phase behoben, so in den anschließenden Phasen davon ausgegangen werden kann, daß das Produkt alle durch die vorhergehenden Phasen zugesicherten Qualitätsmerkmale hat. In Einzelinspektorphasen wird das Produkt zuerst von jeweils einem Reviewer gegen relativ einfache, aber wichtige applikationsunabhängige Defekte (z.B. Entwurfs- bzw. Programmierrichtlinien) geprüft. Danach prüfen mehrere Reviewer in sog. Multiinspektorphasen das Produkt unabhängig voneinander gegen domänen- oder applikationsspezifische Defekte (z.B. Realzeitanforderungen, Robustheit, Benutzerfreundlichkeit). In einer jede Multiinspektorphase abschließenden Sitzung vergleichen die Reviewer ihre Ergebnisse und erstellen ein gemeinsames Protokoll. Im Unterschied zu den anderen Review-Techniken, die jeweils einen Teil des Produktes gegen viele Kriterien prüfen, wird in jeder Phase das gesamte Produkt gegen eine oder einige wenige Eigenschaften geprüft. Experimentell wurden nach der Anwendung solcher Verbesserungen Effizienzsteigerungen von bis zu 35% gemessen [PortVottBasi95].

Neben dem Hauptzweck „Prüfen des inspizierten Dokuments“ sollten die in den Inspektionen gewonnenen Daten auch dazu benutzt werden, die Prüfmethoden im einzelnen sowie den gesamten Entwicklungsprozeß zu optimieren [Fagan76]. Insbesondere zeigen Defektstatisti-

1. Checklisten finden sich in [FreedmanWeinberg90].

ken (in welcher Phase werden wie viele Defekte welcher Art aus welcher Phase gefunden?) Schwachpunkte und die Auswirkungen von Verbesserungsmaßnahmen auf.

3 Objekt-orientierte Reviews

Grundsätzlich sollten Reviews in jeder Entwicklungsphase auf verschiedenen Granularitätsstufen (z.B. Gesamtmodell, Teilsysteme/Frameworks, Gruppen eng gekoppelter Klassen, einzelne Klassen) durchgeführt werden (Abbildung 1). Die in objekt-orientierten Produkten hinzukommenden Freiheitsgrade können prinzipiell durch getrennte Reviews bezüglich der Vererbungshierarchie und der Objektbeziehungen[1] berücksichtigt werden [Siegel96]. Wir empfehlen jedoch, mit einem Review der gesamten Architektur zu beginnen, da oftmals erst das Zusammenspiel von vererbten und redefinierten Eigenschaften die „Essenz des Entwurfes“ ausmacht. Wir gehen deshalb im nächsten Unterabschnitt auf Architekturreviews ein und geben dann in den beiden darauffolgenden Unterabschnitten Kriterien für getrennte Reviews der statischen Struktur und der Dynamik an.

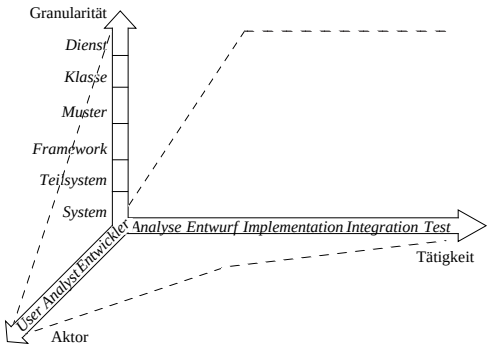

Abbildung 1 SW-Entwicklung

3.1 Architektur

In den letzten Jahren ist sehr viel „Entwicklerwissen“ in Form von sogenannten Mustern (Patterns) katalogisiert worden. So lassen sich in bestimmte Analysemuster auch Domänenübergreifend verwenden [Fowler97] und Entwürfe in ihrer Groben Struktur oft bestimmten, wohlbekannten Architektur-Mustern wie z.B. der 3-Schichtenstruktur oder der Client-Server-Architektur zuordnen [BuschmannEtAl96]. Für bestimmte, engumgrenzte Anwendungsbereiche gibt es Frameworks, welche die Architektur von Teilsystemen wie z.B. der Benutzungsschnittstelle vorgeben. Innerhalb von Teilsystemen finden sich oft bestimmte Entwurfsmuster, d.h. stereotype Arten des Zusammenspiels einiger weniger Klassen, wieder [GoF94].

Neben den weiter unten aufgeführten rein struktur- bzw. verhaltensorientierten Review-Kriterien sollte dementsprechend auch der Einsatz von Standardarchitekturen, Frameworks und Mustern berücksichtigt werden [PagelWinter96]. Fragestellungen sind z.B.:

- Welche Architekturen/Frameworks/Muster sind (nicht) benutzt und warum (nicht)?
- Finden sich alle relevanten Teile der Architekturen/des Frameworks/des Musters wieder?
- Ist der Einsatz der Architekturen/des Frameworks/des Musters notwendig? Welches Kosten/Nutzenverhältnis ergibt sich?

Weitere, auf das jeweils verwendete Muster zugeschnittene Kriterien finden sich in den Abschnitten „Absicht“, „Anwendbarkeit“ und „Konsequenzen“ der Musterbeschreibungen (nach [GoF94]) selbst.

3.2 Statische Struktur

Nachdem in den Architektur-Reviews die Aufteilung des Systems in Teilsysteme sowie die Benutzung von Frameworks und Entwurfsmustern berücksichtigt wurden, wird nun die Feinstruktur der Komponenten bis hin zur einzelnen Klasse untersucht. Einige Kriterien für die Vererbungshierarchie sind z.B.

- Ist dokumentiert, ob es sich um Vererbung im Sinne einer Generalisierung/Spezialisierung oder um Vererbung für Wiederverwendung handelt?
- Ist die Vererbungsbeziehung notwendig oder läßt sich der erwünschte Effekt auch durch eine Benutzungsbeziehung bzw. mittels Aggregation erreichen?
- Sind Blätter der Vererbungshierarchie abstrakte[2] Klassen? Wenn ja, warum? Können Sie eliminiert werden?
- Erben abstrakte Klassen von konkreten Klassen? Wenn ja, warum? Können Sie eliminiert werden?

1. Objektbeziehungen sind Konstruktions- oder Behälterbeziehungen (Aggregation, das Ganze ist verantwortlich für seine Teile) sowie Benutzungsbeziehungen („gleichberechtigte“ Partner ohne gegenseitige Verantwortung, das „Kundenobjekt“ benutzt das „Lieferantenobjekt“).

2. Eine Klasse heißt abstrakt, wenn für einige oder alle ihrer Dienste nur eine Schnittstelle ohne Implementierung angegeben ist.

- Gibt es Klassen, die keine Attribute und/oder Dienste neu definieren bzw. redefinieren? Wenn ja, warum? Können Sie eliminiert werden?

Zusätzlich bei mehrfacher Vererbung:

- Ist der gewünschte Effekt einer wiederholten Vererbung klar dokumentiert?
- Realisiert die Klasse mehrere Konzepte bzw. Aufgaben? Wenn ja, warum?

Anhand der Objektbeziehungen teilen wir das System in mehrere Gruppen gekoppelter Klassen (Cluster) auf. Zwischen Klassen aus unterschiedlichen Clustern sollten dabei nur Benutzungsbeziehungen existieren. Nach folgenden Kriterien kann inspiziert werden:

- Sind die erkennbaren Cluster explizit als solche dokumentiert?
- Sind die Cluster entkoppelt, d.h., bestehen zwischen Klassen aus verschiedenen Clustern keine Konstruktions- und Behälterbeziehungen?
- Existieren zwischen Klassen aus verschiedenen Clustern wenige, klar dokumentierte Benutzungsbeziehungen, welche die notwendigen dynamischen Sichtbarkeiten[1] (zum Aufbau der Botschaftskanäle) ermöglichen?

3.3 Dynamik

Zusätzlich zur Inspektion der statischen Struktur sollten in Reviews alle essentiellen Aufgaben durchgespielt werden, da so der Aufbau der Sichtbarkeitsbereiche für die Benutzungsbeziehungen am deutlichsten wird. Das dem weiteren zugrundeliegende „Ausführungsmodell“ ist synchron und sequentiell: vereinfacht gesprochen wird das sogenannte Wurzelobjekt erzeugt und einer seiner Dienste ausgeführt, wobei dann weitere Objekte erzeugt und, ggf. angestoßen durch externe Ereignisse wie z.B. Benutzeraktionen oder Botschaften von anderen Prozessen, zur Ausführung von Diensten angeregt werden. Ursache und Ausdruck dynamischen Verhaltens im Geflecht kommunizierender Objekte sind also Botschaften, die in den Zielobjekten die Ausführung bestimmter Dienste bewirken und oft zu einer Lawine weiterer Botschaften führen, welche die Zielobjekte an assoziierte Objekte senden.

1. Nicht zu verwechseln mit den in C++ durch die Keywords public, protected und private angegebenen statischen Sichtbarkeiten von Diensten- und Attributen.

Dies berücksichtigend schlagen wir die folgende Form des *objekt-orientierten Walk-Through*‘s vor: Die Objekte jeder beteiligten Klasse werden von einer Person vertreten, die auch den Zustand der Objekte (Werte der atomaren Instanzvariablen, Referenzen auf andere Objekte) nachhält. Beginnend mit dem Ereignis, welches die durchzuspielende Aufgabe auslöst, werden Botschaften zwischen den Objekten bzw. den sie vertretenden Personen gesendet. Die zugrundeliegende Objektkonstellation kann entweder vorgegeben oder - besser, aber zeitraubender - vom Wurzelobjekt ausgehend innerhalb des Walk-Through aufgebaut werden. Folgende Punkte sind zu beachten:

- Sind alle verwendeten atomaren Instanzvariablen (Attribute) initialisiert?
- Bestehen Referenzen zu den Zielobjekten der vom Dienst ausgesendeten Botschaften (Instanzvariable bzw. Parameter der den Dienst auslösenden Botschaft)?
- Sind die aktuellen Parameter der vom Dienst ausgesendeten Botschaften bekannt? Stimmen Ihre Typen (bei statisch getypten Sprachen) bzw. verstehen die Zielobjekte die Botschaft (bei dynamisch getypten bzw. ungetypten Sprachen)?
- Werden die Rückgabewerte benutzter Dienste adäquat weiterverarbeitet?

Voraussetzung hierzu ist, das die von einem Dienst verwendeten Dienste (aus der „eigenen“ oder aus anderen Klassen) im Entwurf spezifiziert sind (statisch oder ggf. dynamisch in Form von Szenarien).

4 Code-Inspektionen

Quellcode kann getrennt nach verschiedenen inneren Qualitätsaspekten wie z.B. Wiederverwendungspotential, Wartbarkeit, Portabilität und Effizienz inspiziert werden. Codeinspektionen sind jedoch nur begrenzt nach allgemeinen Richtlinien ausrichtbar, da die Möglichkeiten und Eigenarten verwendeten Programmiersprache sowie Firmeninterna berücksichtigt werden müssen. Wir geben deshalb im weiteren nur einige wenige, allgemein zu beachtende Punkte an. Für spezielle Inspektionslisten z.B. als Grundlage für firmeneigene Programmierrichtlinien siehe z.B. für C++ in [Coplien92], [Baldwin92], für Smalltalk-80 in [Johnson92], [Beck97].

- **Wiederverwendbarkeit.** Sind (abstrakte) Klassen in Bezug auf Pattern/Frameworks dokumentiert? Sind die von der Anwendung bereitzustellenden bzw. zu überschreibenden

Klassen/Methoden (Hook-Klassen bzw. Hook-Methoden [PagelWinter96]) gekennzeichnet? Gibt es eine geeignete Schnittstelle für benutzende Objekte? für spezialisierende Nachfahren? Kann die Klasse/der Dienst verallgemeinert werden?

- **Wartbarkeit.** Wurden alle Programmierrichtlinien bzw. allgemeine konstruktive Qualitätsvorschriften eingehalten („Law of Demeter“ etc.)? Sind die Dienste minimal (Signatur/ Kontrollflußkomplexität)?
- **Portabilität.** Wurden maschinennahe bzw. maschinenabhängige Befehle (Cast, Bit-Shift...) verwendet? Wenn ja, sind diese wirklich notwendig? Sind sie geeignet dokumentiert/gekapselt? Sind die Bibliotheksabhängigkeiten dokumentiert?
- **Effizienz.** Implementiert der Code den Algorithmus innerhalb der theoretischen Zeit/ Raumkomplexität? Welche Kompromisse wurden warum getroffen? Gibt es in zeitkritischen Diensten unvorhergesehene Verluste aufgrund der Botschaftsmenge? durch dynamisches Binden? durch doppelten oder mehrfachen „Methoden-Dispatch“ [Beck97]?

Auch für Code-Inspektionen empfehlen wir, zusätzlich zu den eher „syntaktischen“ Checks objekt-orientierte Walk-Throughs nach der im vorigen Kapitel genannten Technik durchzuspielen. Da Botschaftsfolgen auf dieser Ebene jedoch sehr komplex werden können, sollten sie spätestens bei Diensten wiederverwendeter (Bibliotheks-) Klassen beendet werden, sofern diese keine weiteren Dienste der zu inspizierenden Klassen benutzen.

5 Werkzeuge

In [Gerrard94] wird ein Werkzeug zur interaktiven Prüfung textuell spezifizierter, strukturierter Anforderungen (Requirements Review) beschrieben. In [PagelWinter96] haben wir ein Konzept für eine „intelligente Musterbibliothek“ vorgestellt. Der Entwickler kann Beispiele von Mustern in seinen Entwurf kopieren und abändern, oder aber Elemente eines bereits bestehenden Entwurfs Elementen von Mustern zuordnen. In Inspektionen kann das (als Prototyp in Smalltalk-80 [GoldbergRobson89]) implementierte Werkzeug helfen, Musterspezifische Fragen zu beantworten. Zur Zeit arbeiten wir an einer Erweiterung des Werkzeuges, mit der Botschaftsfolgen für einen Entwurf spezifiziert und dann in objekt-orientierten Walk-Throughs durchgespielt werden können.

Für Codeinspektionen bieten sich zuerst statische Analysatoren wie z.B. QA-C++, Logiscope oder STW/Advisor an, die automatisch Verstöße gegen die Programmierrichtlinien sowie weitere Qualitätseigenschaften bis hin zu Datenflußanomalien ermitteln. Manuelle Codeinspektionen werden von graphisch-interaktiven Browsern unterstützt, welche die gleichzeitige, verknüpfte Ansicht mehrerer Klassen bzw. Dienste erlauben.

Werkzeuge sind konstruktiv und analytisch einsetzbar. Der Einsatz von Code-Inspektionswerkzeugen sollte jedoch auf den analytischen Bereich beschränkt werden, um eine hochgradig iterative, auf die Kriterien des Werkzeuges zielende Entwicklertätigkeit zu verhindern und die Effizienz der Inspektionen zu erhalten.

6 Und wie weiter?

In diesem Beitrag haben wir einige Fragestellungen für Inspektionen objekt-orientierter Software beleuchtet. Wir haben objekt-orientierte Walk-Throughs für Design- und Codereviews eingeführt. Ob und wie die vorgeschlagenen Techniken und Werkzeuge in der Praxis eingesetzt werden können, hängt selbstverständlich vom jeweiligen Entwicklungsprozeß und den dort angefertigten Dokumenten ab. Hier muß sicherlich einiges „maßgeschneidert“ werden.

Wir empfehlen Einsteigern in die Inspektionstechnik, sozusagen bottom-up mit Checklistengestützten Code-Inspektionen zu beginnen. Für die meisten Programmiersprachen sind fertige Checklisten erhältlich, darüber hinaus sind Erfolge hier am einfachsten meßbar. Mit dem zu inspizierenden Quellcode können erläuternde Entwurfs-Dokumente verteilt werden. In einer die Inspektion abschließenden Runde werden die Entwurfs-Dokumente bewertet und auf fehlende bzw. wünschenswerte weitere Informationen hingewiesen. Mit diesen Informationen werden dann Checklisten für erste Entwurfs-Inspektionen erstellt, nach dann wiederum analog in Analyse-Inspektionen eingestiegen wird.

7 Literatur

[Baldwin92] John T. Baldwin *An Abbreviated C++ Code Inspection Checklist* ftp://cs.uiuc.edu:/pub/testing/baldwin-inspect.ps

[Beck97] Kent Beck *Smalltalk Best Practice Patterns, Vol. 1: Coding* Prentice Hall, Upper Saddle River, N.J. 1997

[BuschmannEtAl96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal *Pattern-Oriented Software Architecture - A System of Patterns* - J. Wiley & Sons, Chichester, 1996

[Coplien92] James O. Coplien *Advanced C++ Programming Styles and Idioms* Addison-Wesley, Reading, Massachusetts 1992

[Fagan76] Michael E. Fagan *Design and Code Inspections to Reduce Errors in Program Development* IBM Systems Journal, Vol. 15, No. 3, 1976, pp. 182-211

[Fowler97] Martin Fowler *Analysis Patterns - Reusable Object Models* Addison-Wesley, Reading, Massachusetts 1997

[FreedmanWeinberg90] Daniel P. Freedman, Gerald M. Weinberg *Handbook of Walkthroughs, Inspections, and Technical Reviews* 3rd. Ed., Dorset House Publishing, New York 1990

[Gerrard94] Paul Gerrard Testing Requirements Proc. EuroSTAR 1994, 10-13 Okt. 1994, Brüssel, Belgien http://www.ftech.net/~evolutif/TestReqs/TESTREQS.html

[GoF94] Erich Gamma, Richard Helm, Ralph E. Johnson, John Vlissides (the Gang of Four) *Design Patterns - Elements of Object-Oriented Reusable Software* Addison-Wesley, Reading, Massachusetts 1994

[GoldbergRobson89] Adele Goldberg, David Robson *Smalltalk-80 - The Language* Addison-Wesley, Reading, Massachusetts 1989

[Johnson92] Ralph E. Johnson *Smalltalk Classic Bugs* Univ. of Illinois, Urbana-Champaign, 1992 http://st-www.cs.uiuc.edu/ftp/pub/Smalltalk/st-docs/classic-bugs.txt

[KnightMyers93] John C. Knight, E. Ann Myers *An Improved Inspection Technique* Comm. ACM, Vol. 36, No. 11, 1993, pp. 51-61

[MacdEtAl96] F. Macdonnald, J. Miller, A. Brooks, M. Roper, M. Wood *Applying Inspection to Object-Oriented Code* Software Testing, Verification, and Reliability, Vol. 6, 1996, pp. 61-82

[PagelWinter96] Bernd-Uwe Pagel, Mario Winter *Towards Pattern-Based Tools* EuroPLoP‘96 Conf. Proc., Irsee, 1996 http://voss.fernuni-hagen.de/pi3/abstracts/pi3.EuroPLoP96.html

[ParnasWeiss85] D. L. Parnas and D. M. Weiss *Active design reviews: principles and practices* Proc. 8. International Conference on Software Engineering, IEEE Computer Society Press, Aug. 1985, pp. 132--136 http://www.itd.nrl.navy.mil/ITD/5540/publications/a7/misc/icse8.ps

[PortVottBasi95] Adam A. Porter, Lawrence G. Votta, Victor R. Basili *Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment* IEEE Transactions on SE, Vol. 21, No.6, June 1995, pp. 563-575

[Siegel96] Shel Siegel *Object-Oriented Software Testing - A Hierarchical Approach* John Wiley & Sons, New York 1996

[WheeBrykMees96] David A. Wheeler, Bill Brykczinski, Reginals N. Meeson, Jr. *Software Inspektion - An Industrial Best Practice* IEEE Press, Los Alamitos 1996

Umfangreiches Material zu Reviews und Inspektionen findet sich im WWW auch unter http://www.cs.strath.ac.uk/research/efocs/reports.html sowie unter http://www.ics.hawaii.edu/~johnson/FTR/.