

Ein Qualitätsmodell für den objektorientierten Entwurf

Ralf Reißing

reissing@informatik.uni-stuttgart.de

Abteilung Software Engineering, Institut für Informatik, Universität Stuttgart
Breitwiesenstr. 20-22, D-70565 Stuttgart

Betreuer der Arbeit: Prof. Dr. Jochen Ludewig
Art der Arbeit: Dissertation
Fachbereich GI: 2

Zusammenfassung

Eines der effektivsten, aber auch schwierigsten Mittel der Kostenminimierung bei der Software-Entwicklung ist die Erstellung eines guten Entwurfs. Ratschläge für einen guten Entwurf gibt es viele, doch ist die Bestimmung der Güte, also der Qualität eines Entwurfs, nicht leicht. Bestehende Ansätze zur Bewertung sind eher unbefriedigend, daher wird ein eigener Ansatz, basierend auf einem Qualitätsmodell, vorgeschlagen.

1 Einführung

Ein wichtiges Ziel des Software Engineering ist die Reduzierung der Kosten für Software über den gesamten Lebenszyklus. Der Entwurf ist einer der einflussreichsten Kostenfaktoren, weil er nicht nur die Kosten seiner eigenen Erstellung verursacht, sondern auch die Kosten der nachfolgenden Phasen (Implementierung und Wartung) wesentlich beeinflusst. Boehm (1976) und andere haben ermittelt, dass mindestens die Hälfte der Gesamtkosten in der Wartung anfällt, daher sind Kostenreduzierungen in der Wartung besonders lukrativ.

Die Erfahrung zeigt, dass ein schlechter Entwurf hohe Implementierungs- und Wartungskosten verursacht. Ein guter Entwurf hingegen führt zu geringeren Gesamtkosten, obwohl seine Erstellung teurer sein kann. Daher gibt es einen Bedarf für hohe Entwurfsqualität. Diese Arbeit betrachtet den objektorientierten Entwurf, da die objektorientierte Vorgehensweise zunehmend die traditionelle, strukturierte verdrängt, während gleichzeitig das Wissen über guten objektorientierten Entwurf geringer und weniger verbreitet ist.

Auf der Suche nach einem guten Entwurf stößt man meist auf mehrere Alternativen, unter denen die beste auszuwählen ist. Das folgende Beispiel zeigt, dass es bei der Entscheidung, welcher Entwurf für ein gegebenes Problem der beste ist, viele Kriterien zu berücksichtigen gibt, und dass Kriterien zueinander im Widerstreit stehen können.

2 Beispiel Videoverleih

Dieses Beispiel wurde von einem Beispiel von Fowler et al. (1999, Kap. 1) inspiriert. Ein Videoverleih soll ein System zur Rechnungsstellung erhalten. Der Kunde (Customer), eine Ausleihe (Rental) und ein Film (Movie) werden durch Klassen modelliert. Der Kunde hat Ausleihen, während die Ausleihe den ausgeliehenen Film und die Ausleihdauer (in Tagen) kennt. Ein Film hat einen Preiscode, der zusammen mit der Leihdauer den Preis bestimmt. Es gibt Preiscodes für normale Filme (regular), Kinderfilme (children's) und neue Filme (new release). Der Preiscode eines Films kann sich ändern, daher gibt es in Movie eine Methode setPriceCode.

2.1 Erste Stufe: Entkopplung und Kapselung

Entwurf A (vgl. Abb. 1) sieht vor, den Rechnungsbetrag in der Klasse Customer zu berechnen und dazu die erforderliche Information bei Rental und Movie einzuholen. Customer holt sich in der Methode statement von Rental die Ausleihdauer und den Film, von diesem wiederum den Preiscode. Dieses Vorgehen ist ein Verstoß gegen das Demetergesetz (Law of Demeter: Lieberherr, Holland, 1989), weil auf einem Objekt, das von einer anderen Klasse (Rental) als Resultat eines Methodenaufrufs (getMovie) geliefert wurde, eine Methode aufgerufen wird (getPriceCode). Dadurch entsteht eine unnötige Kopplung zwischen Customer und Movie (als Benutzt-Beziehung im Klassendiagramm sichtbar). Stattdessen sollte hier mit Delegation über Rental gearbeitet werden.

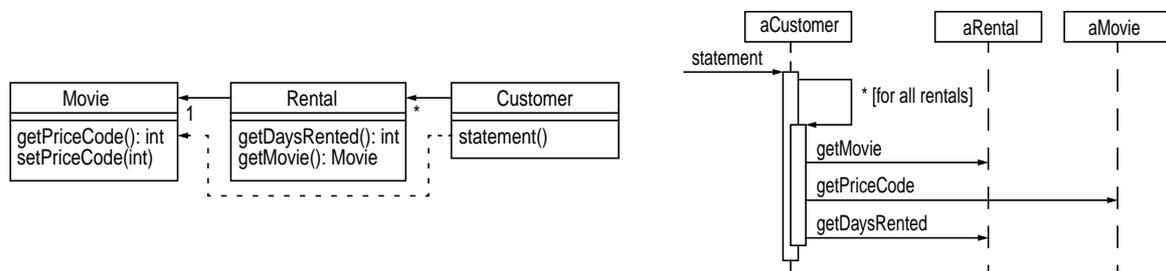


Abb. 1: Klassendiagramm und Sequenzdiagramm für Entwurf A.

Darüber hinaus lässt sich feststellen, dass die Preisberechnung für einen einzelnen Ausleihvorgang logisch gesehen eigentlich zu Rental gehört, da dort bereits alle erforderlichen Informationen vorliegen. In statement müssten die einzelnen Beträge dann nur noch aufsummiert werden. Die Preisberechnung anhand des Preiscode ist bei Movie am besten aufgehoben, da sich so neue Preiscode besser kapseln lassen (Verbergen einer Entwurfsentscheidung nach dem Information-Hiding-Prinzip: Parnas, 1972). Die zur Berechnung benötigte Leihdauer wird als Parameter übergeben. So gelangt man zu Entwurf B (vgl. Abb. 2). Vergleicht man Entwurf

B mit A, fällt positiv auf, dass zwischen den Klassen weniger Kopplung besteht (Customer weiß nichts mehr von Movie). Die Kapselung von Movie und Rental wurde ebenfalls verbessert, so dass nun auch die drei get-Methoden, die von statement vorher benötigt wurden, entfallen können (getPriceCode, getDaysRented, getMovie). Insgesamt ist der Entwurf weniger zentralistisch. Auf der anderen Seite ist die Berechnung des Rechnungsbetrags nun über drei Klassen „verschmiert“, so dass man drei Klassen statt nur einer betrachten muss, um den Algorithmus vollständig nachzuvollziehen.

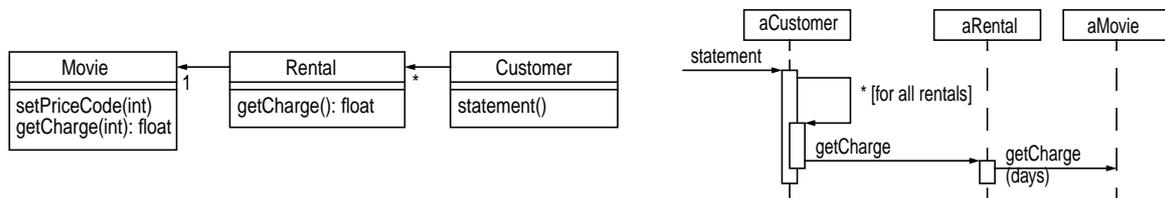


Abb. 2: Klassendiagramm und Sequenzdiagramm für Entwurf B.

2.2 Zweite Stufe: State-Muster für Movie

Entwurf C entsteht aus B durch die Anwendung des State-Musters (Gamma et al., 1995). Die Preisberechnung für einen Film wird in eine abstrakte Klasse Price ausgelagert. Zu Price gibt es für jeden Preiscode eine konkrete Unterklasse, die den spezifischen Preis berechnen kann (vgl. Abb. 3). Der Vorteil dieser Lösung ist, dass getCharge in Movie leichter zu implementieren ist, da die Fallunterscheidung nach dem Preiscode nun durch dynamisches Binden erledigt wird. Die Lösung ist daher einfacher um neue Preiscodes zu erweitern, und Änderungen in der Preisberechnung für einzelne Preiscodes sind leichter durchzuführen. Die Trennung zwischen den Geschäftsregeln und der restlichen Programmlogik (Separation of Policy and Implementation: Buschmann et al., 1996, S. 401) ist hier am besten umgesetzt. Der Preis dafür ist allerdings hoch: Vier neue Klassen, so dass sich die Anzahl der Klassen im gezeigten Ausschnitt mehr als verdoppelt. Die Verständlichkeit, die auch von der Anzahl der Klassen abhängt, wird dadurch aber verschlechtert.

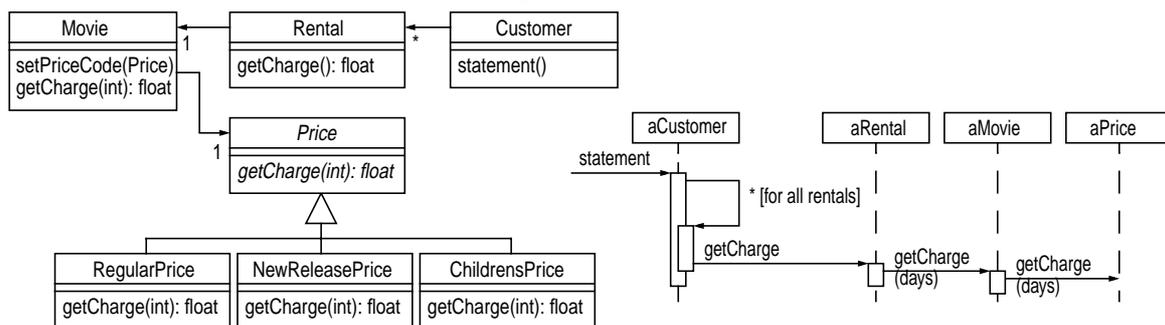


Abb. 3: Klassendiagramm und Sequenzdiagramm für Entwurf C.

2.3 Schlussfolgerungen

Die Änderungen von Entwurf A nach B und von B nach C dienen der Entkopplung und damit der Änderbarkeit. Bei Entwurf C nimmt die Zahl der Klassen aber stark zu und verschlechtert so die durch die Entkopplung verbesserte Verständlichkeit wieder. Es wird Flexibilität eingebaut, die aber nur dann wirklich sinnvoll ist, wenn sich die Preiscode tatsächlich ändern werden. Die Güte des Entwurfs hängt also auch vom Kontext der Software ab.

Man kann feststellen, dass es Grundsätze gibt, die fast immer sinnvoll sind, zum Beispiel Entkopplung sowie Kapselung von Details und sich wahrscheinlich ändernden Entwurfsentscheidungen. Diese Grundsätze sollten immer beachtet werden, wenn dabei nur geringe Mehrkosten entstehen. Flexibilität, insbesondere durch Entwurfsmuster, ist oft teuer und sollte erst dann eingebaut werden, wenn sie benötigt wird. Häufig muss dazu auch der vorhandene Entwurf geändert werden: Zum Beispiel kann das State-Muster auf Entwurf A nicht ohne vorhergehende Änderungen angewendet werden.

Jeder am Entwurf Beteiligte oder von ihm Betroffene (Entwickler, Projektmanager, Projekteigentümer, Kunde und Anwender) hat andere Ansprüche an den Entwurf. So ist es nur natürlich, dass es unterschiedliche Auffassungen über Entwurfsqualität gibt. Selbst unter den Entwicklern gibt es unterschiedliche Interessensgruppen: Entwerfer, Tester, Codierer und Wartungsentwickler.

3 Eine Theorie des guten Entwurfs: erste Schritte

3.1 Gute Praktiken

Bis heute sind viele erwünschte Eigenschaften eines Entwurfs vorgeschlagen worden, zum Beispiel Konsistenz, Zurückverfolgbarkeit (traceability), Änderbarkeit und Wiederverwendbarkeit. Es wurden Unmengen von Ratschlägen publiziert, wie man diese Eigenschaften erreichen kann, z. B. Methoden, Prinzipien, Heuristiken und Entwurfsmuster. Klassische Beispiele sind die Forderung, Kopplung zu minimieren und Zusammenhalt zu maximieren (Stevens et al., 1974) oder das Information Hiding von Parnas (1972), das das Verbergen von Entwurfsentscheidungen in Modulen empfiehlt. Beide Prinzipien streben u. a. eine bessere Änderbarkeit an.

Einen guten Entwurf zu erstellen ist nicht leicht. Wesentliche Schwierigkeiten dabei sind instabile Anforderungen und die essentielle Komplexität des Entwerfens, das viel Erfahrung bei den Entwerfern voraussetzt. Schließlich hängt die Entwurfsqualität stark von den Fähigkeiten der Entwerfer ab. Brooks (1987) stellt fest: "good designs can be achieved by following good practices", doch es gelte auch: "great designs come from great designers". Nach Brooks wäre es am besten, großartige Entwerfer auszubilden und vor allem diese einzusetzen. Leider haben aber nur wenige Entwickler das Potential, großartige Entwerfer zu werden. Deshalb brauchen wir die guten Praktiken, die Brooks anspricht, um normalen Entwerfern dabei zu helfen, gute Entwürfe zu erschaffen. Die angesprochenen Ratschläge sind Teil dieser guten Praktiken.

Das Problem mit den Ratschlägen ist, dass sie nicht recht in einen gemeinsamen Rahmen passen, einige widersprechen sich sogar. Das liegt zum Teil daran, dass jeder Ratschlag bestimmte Ziele verfolgt, die sich bei mehreren Ratschlägen natürlich unterscheiden oder sogar im Widerspruch zueinander stehen können. Eine „Große Vereinigende Theorie des Entwurfs“, die brauchbare Ratschläge in einen gemeinsamen Rahmen stellt und gleichzeitig die unbrauchbaren aussortiert, fehlt noch. Eine solche Theorie hat sich noch nicht herausgebildet, was darauf hindeutet, dass es sehr schwierig ist, eine solche aufzustellen und zu validieren.

3.2 Ein Qualitätsmodell für den objektorientierten Entwurf

Allerdings sind Schritte in diese Richtung trotzdem hilfreich und möglich. Ein Zwischenschritt zu der allgemeinen Entwurfstheorie ist ein Qualitätsmodell für den Entwurf, das die erwünschten Eigenschaften definiert. Dieses Modell beantwortet zunächst die Frage: Was macht einen guten Entwurf aus?

Nun kann jeder Ratschlag in Relation zu diesem Modell evaluiert werden, und man kann herausfinden, ob und wo er passt. Das sich schließlich herausbildende Rahmenwerk macht es dem Entwerfer leichter zu ermitteln, welche Ratschläge verfügbar sind und wann ein bestimmter Ratschlag sinnvoll eingesetzt werden kann. Das Modell kann schließlich die Frage beantworten werden: Wie komme ich zu einem guten Entwurf?

Das Vorgehen bei der Erstellung des Modells ist wie folgt: Zunächst werden alle relevanten Qualitätskriterien für das Modell identifiziert. Mögliche Quellen dafür gibt es viele; zum Beispiel einige Vorschläge für Kriterien zur Bewertung objektorientierter Entwürfe (z. B. Booch, 1994; Coad, Yourdon, 1991). Außerdem kann einiges aus dem Bereich des strukturierten Entwurfs wiederverwendet werden (z. B. Card, Glass, 1990).

Anschließend werden die Kriterien mit messbaren Attributen des Entwurfs verknüpft. Dann werden Metriken für diese Attribute definiert. Um die üblichen Schwierigkeiten mit der oft unpräzisen Definition von Metriken zu vermeiden, beruhen die Definitionen auf einem formalen Modell des objektorientierten Entwurfs.

Das Modell kann eingesetzt werden, um Entwürfe zu bewerten, also deren Qualität zu ermitteln. Natürlich unterscheiden sich die Qualitätsanforderungen von Projekt zu Projekt, weshalb das Modell anpassbar sein sollte, zum Beispiel durch Parameter, die bei der Aggregation von Metriken als Gewichte dienen (nicht benötigte Aspekte können durch die Vergabe von Gewichten mit dem Wert 0 ausgeblendet werden). Die Wahl der Gewichte repräsentiert so einen konkreten Kompromiss zwischen den widerstreitenden Entwurfskriterien.

Das Messen für das Modell benötigt lediglich Artefakte wie zum Beispiel UML-Klassendiagramme, nicht den Code, so dass die Evaluation bereits in der Entwurfsphase möglich ist. Allerdings kann das Modell auch um code-basierte Entwurfsbewertungsmöglichkeiten (z. B. die zyklomatische Komplexität von Methoden) erweitert werden.

4 Das Qualitätsmodell

Die in der Literatur vorgefundenen Modelle sind überwiegend unbefriedigend, da sie sich meistens nur auf die Bewertung einer Klasse beschränken (z. B. Booch, 1994). Beziehungen zwischen Klassen (Vererbung, Assoziation, Benutzung) und größere Einheiten wie Pakete oder Subsysteme bekommen relativ wenig Aufmerksamkeit. Dabei bleiben diese Modelle meist auch auf der qualitativen Ebene stehen. Quantitative Ansätze gibt es fast nur auf Code-Ebene, wobei die Metriken oft zum Selbstzweck werden, d. h. die Anbindung an übergeordnete Kriterien fehlt oder bleibt unklar.

4.1 Grundannahmen

Bei der Bewertung wird von folgender Grundannahme ausgegangen: Der beste Entwurf ist derjenige, der über die gesamte Lebenszeit der Software minimale Kosten verursacht. Diese Definition hat den Vorteil, dass sich Kosten relativ leicht messen lassen. Allerdings kann es in Einzelfällen schwierig sein, anfallende Kosten eindeutig dem Entwurf als Verursacher zuzuordnen.

Die Ermittlung der Gesamtkosten ist während der Erstellung des Entwurfs allerdings nicht möglich, da die vom Entwurf bestimmten Kosten ja hauptsächlich erst in späteren Aktivitäten anfallen. Da man die Kosten nicht direkt messen kann, zieht man sich traditionell auf die Messung von Kriterien zurück, von denen man annimmt, dass sie mit den Kosten korreliert sind, also Indikatoren für die Kosten darstellen. Diese Zusammenhänge sind für den objektorientierten Entwurf allerdings empirisch kaum belegt. Manche sind immerhin für den strukturierten Entwurf belegt, von dem einige Kriterien wie zum Beispiel Kopplung und Zusammenhalt übernommen wurden.

4.2 Aufbau des Modells

Das Qualitätsmodell QOOD (Quality of Object-Oriented Design) zerfällt in drei Bereiche: Brauchbarkeit, Wartbarkeit und Wiederverwendbarkeit. Der Schwerpunkt liegt wegen des hohen Kostenanteils der Wartung

bei der Wartbarkeit. Die Brauchbarkeit deckt einen Bereich ab, der bei den meisten vorhandenen Modellen wohl als selbstverständlich vorausgesetzt und gar nicht erwähnt wird: Der Entwurf muss die Anforderungen vollständig und korrekt umsetzen sowie in sich konsistent sein, so dass er auch realisiert werden kann. Es lässt sich grob sagen, dass die Brauchbarkeit die Sicht des Kunden und des Anwenders reflektiert, die Wartbarkeit die Sicht der Entwickler und die Wiederverwendbarkeit die Sicht des Projektmanagers und -eigentümers. Das Qualitätsmodell besteht aus einer Hierarchie von Entwurfskriterien (vgl. Abb. 4) Die Kriterien wurden so gewählt, dass sich ein hoher Grad der Erfüllung dieser Eigenschaft positiv auf die Entwurfsqualität auswirkt. Aus Platzgründen können die Kriterien und ihre Zusammenhänge hier nicht näher erläutert werden. Da der Schwerpunkt des Modells auf der Wartbarkeit liegt, macht es Sinn, in diesem Bereich den wichtigsten Aspekt, die Änderbarkeit, zu verfeinern.

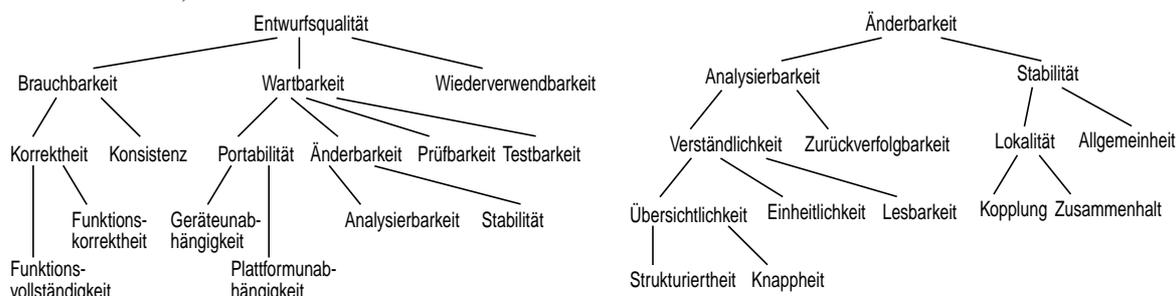


Abb. 4: Das Qualitätsmodell und die Verfeinerung der Änderbarkeit.

Betrachtet man die Kriterien genauer, stellt man fest, dass manche sich direkt auf den Entwurf beziehen (z. B. Korrektheit), andere aber auf die aus dem Entwurf abgeleitete Implementierung (z. B. Portabilität). Eine klare Trennung zwischen direkten und indirekten Entwurfskriterien ist allerdings schwierig, da der Entwurf ja struktureller Bestandteil der Implementierung wird.

5 Ausblick

Die Arbeiten am Qualitätsmodell sind noch nicht abgeschlossen. Nach der Stabilisierung der Kriterien muss noch die Quantifizierung durch Metriken erfolgen.

Das Modell kann nach seiner Fertigstellung für die Verbesserung des Entwurfs und die Qualitätssicherung eingesetzt werden. Qualitätssicherung zerfällt in drei Bereiche:

- organisatorisch: Ein Rahmen für das Entstehen hoher Qualität wird geschaffen. Den Entwerfern müssen die Eigenschaften eines guten Entwurfs bewusst gemacht werden. Diese werden dem Modell entnommen.
- konstruktiv: Richtlinien und Hinweise werden gegeben, wie Qualität in den Entwurf eingebaut werden kann. Diese werden aus dem Modell abgeleitet.
- analytisch: Der Entwurf wird evaluiert und auf Schwächen und potentielle Probleme untersucht, zum Beispiel durch Entwurfsinspektionen. Die Evaluation basiert auf dem Modell.

Um die Qualitätssicherung zu unterstützen, sollte das Modell um Checklisten angereichert werden. Diese helfen dabei, den Entwurf zu evaluieren, ohne alle Metriken berechnen zu müssen. Die Checklisten können auch Bereiche abdecken, die schwierig zu messen sind (z. B. Verständlichkeit).

Darüber hinaus kann das Modell um Anmerkungen zur Entwurfsverbesserung erweitert werden. Zum Beispiel kann für jedes Kriterium eine Liste typischer Probleme, die zu einer schlechten Bewertung führen, angegeben werden. Zu jedem typischen Problem gibt es dann eine Liste möglicher Lösungen, zum Beispiel Entwurfsmuster (Gamma et al., 1995, Buschmann et al., 1996) oder Refaktorisierungen (Fowler, 1999). Eine Sammlung von Prinzipien und Heuristiken (Riel, 1996) sowie Beispiele für guten und schlechten Entwurf wären auch hilfreich. Auf diese Weise wird das Qualitätsmodell zur Basis eines Handbuchs für den objektorientierten Entwurf.

Literatur

- Boehm, B. (1976): Software Engineering. IEEE Transactions on Computers, 25(12), 1226-1241.
- Booch, G. (1994): Object-Oriented Analysis and Design with Applications (2. Aufl.). Benjamin/Cummings, Redwood City.
- Brooks, F. (1987): No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer, 20(4), 10-19.
- Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. (1996): Pattern-Oriented Software Architecture: A System of Patterns. Wiley, Chichester.
- Card, D.; Glass, R. (1990): Measuring Software Design Quality. Prentice-Hall, Englewood Cliffs.
- Coad, P.; Yourdon, E. (1991): Object Oriented Design. Prentice Hall, Englewood Cliffs.
- Fowler, M.; (1999): Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading.
- Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1995): Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading.
- Lieberherr, K.; Holland, I. (1989): Assuring Good Style for Object-Oriented Programs. IEEE Software, 6(5), 38-48.
- Parnas, D. (1972): On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM, 15(12), 1053-1058.
- Riel, A. (1996): Object-Oriented Design Heuristics. Addison-Wesley, Reading.
- Stevens, W.; Myers, G.; Constantine, L. (1974): Structured Design. IBM Systems Journal, 13(2), 115-139.