Anhang: Problem- und Lösungsbereich

Bei den Anwendungsfunktionen eines Systems kann man zwischen **Verarbeitungsfunktionen** und den sogenannten **technischen Funktionen** unterscheiden.

Eine **Verarbeitungsfunktion** "tut" etwas und ist sozusagen ein Prozessor, der ein Problem löst, das bereits im Problembereich[92] der untersuchten Domäne existiert. Beim Studium des **Problembereichs** – also der Fachlichkeit – eines Programms stehen im Rahmen der **Analyse** die fachlichen Aufgaben, also die Verarbeitungsfunktionen, und ihre Verknüpfungen im Zentrum des Interesses. Die Beschränkung in der Analyse auf Verarbeitungsfunktionen verhindert, dass man sich in der Menge der zu analysierenden Funktionen verläuft. Man hat in der Analyse noch kein lauffähiges Programm, sondern befasst sich mit den für das Programm erforderlichen fachlichen Funktionen und deren Abläufen aus logischer Sicht.

Um eine untersuchte **Verarbeitungsfunktion** in Realität ordnungsgemäß auf dem Rechner zum Laufen zu bringen, braucht man beim **Entwurf** im **Lösungsbereich** in der Regel zusätzlich zu den betrachteten Verarbeitungsfunktionen noch die bereits genannten **technischen Funktionen des Lösungsbereichs**[93] wie Funktionen zur Datenhaltung.

Technische Funktionen gibt es ausschließlich im Lösungsbereich

Sogenannte **technische Funktionen** sind:

- Funktionen zur **Datenhaltung**,
- Funktionen zur **Ein- und Ausgabe**,
- Funktionen zur **Rechner-Rechner-Kommunikation,**
- Funktionen zur **Sicherheit**,
- Funktionen zur **Parallelität/Interprozesskommunikation**.

Natürlich muss es eine im Problembereich bzw. in der Analyse betrachtete Verarbeitungsfunktion in anderer Form auch im Lösungsbereich bzw. dem Entwurf geben, denn ansonsten würde ihre Funktionalität einfach unter den Tisch fallen.

Verarbeitungsfunktionen einschließlich der hier aufgeführten technischen Funktionen der Datenhaltung, Ein- und Ausgabe und Rechner-Rechner-Kommunikation werden zusammen auch **"Grundfunktionen der Informationstechnik"** genannt, da sie elementar für physische Systeme der Informationstechnik sind.

Die Funktionen zur Sicherheit und Parallelität/Interprozesskommunikation betreffen **Qualitäten des Systems**, die auf der Existenz bestimmter Funktionen beruhen, also auf **funktionalen Qualitäten**.

---

[92] Der Problembereich stellt die logische Sicht, also die Sicht der Fachlichkeit, einer Anwendung dar.
[93] Der Lösungsbereich umfasst alle Funktionen, die eine Anwendung zum Laufen braucht.

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*,
https://doi.org/10.1007/978-3-658-25975-4

Die folgende Tabelle analysiert die Bedeutung der verschiedenen Funktionalitäten eines Anwendungsprogramms bei der Analyse im Problembereich und dem Entwurf im Lösungsbereich:

Funktionalitäten	**Analyse/ Problembereich**	**Entwurf/ Lösungsbereich**
Verarbeitung	**detailliert betrachtet**	**detailliert betrachtet**
Datenhaltung	Es wird nur betrachtet, was gespeichert wird.	**detailliert betrachtet**
Ein- und Ausgabe	Es wird nur betrachtet, was ein- bzw. ausgegeben wird.	**detailliert betrachtet**
Rechner-Rechner-Kommunikation	Es wird nur betrachtet, was übertragen wird.	**detailliert betrachtet**
Sicherheit	nicht betrachtet	**detailliert betrachtet**
Parallelität/Interprozesskommunikation	nicht betrachtet	**detailliert betrachtet**

Tabelle Anhang-1 Funktionsklassen eines objektorientierten Anwendungsprogramms

Es ist nicht möglich, für technische Funktionen generell ein Schichtenmodell zu zeichnen.

Literaturverzeichnis

Abkürzungen erhalten bei Büchern und bei Internetquellen, die eine Jahreszahl aufweisen, 5 Zeichen. Die ersten drei Buchstaben werden aus dem ersten Namen der Autoren gebildet, wobei der erste Buchstabe groß geschrieben wird. Die Zeichen 4 und 5 speichern die letzten beiden Ziffern des Erscheinungsjahrs. Gibt es von einem Autor mehrere Veröffentlichungen im selben Jahr, so wird sein Name in der 3. Stelle eindeutig abgeändert.

Der Name von Internetquellen ohne Jahreszahl besteht aus 6 klein geschriebenen Zeichen.

App96 Appleton, B.: "(OTUG) Law of Demeter" 24 10 1996.
Online verfügbar:
http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/AppletonExplainsLoD.txt
[Zugriff am 08 02 2019].

appint Appleton, B.: "Introducing Demeter and its Laws".
Online verfügbar:
http://www.bradapp.com/docs/demeter-intro.html
[Zugriff am 08 02 2019].

Bar68 Barnett, T. O. and Constantine, L. L., eds: "Segmentation and Design Strategies for Modular Programming." In "Modular Programming: Proceedings of a National Symposium". Cambridge, Mass., Information & Systems Press, 1968.

Bec97 Beck, K.: "Make it Run, Make it Right: Design Through Refactoring. The Smalltalk Report", 6, pp. 19-24, 1997.

Bec99 Beck, K: "Extreme Programming Explained: Embrace Change". Addison-Wesley Longman, Amsterdam, 1999.

Ber11 Berens, D., "Ein Refaktorisierungswerkzeug zur Umsetzung des Law of Demeter". Masterarbeit, Okt. 2011.
Online verfügbar:
https://www.fernuni-hagen.de/imperia/md/content/ps/masterarbeit-berens.pdf
[Zugriff am 08 02 2019].

bockda Bock., D.: "The Paperboy, The Wallet, and The Law Of Demeter".
Online verfügbar:
http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf
[Zugriff am 08 02 2019].

Boo95 Booch, G.: "Object Solutions: Managing the Object-Oriented Project". Addison-Wesley, 1995.

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*,
https://doi.org/10.1007/978-3-658-25975-4

Dav95 Davis, A. M.: "201 Principles of Software Development". Mc Graw-Hill, 1995.

DeM79 DeMarco, T.: "Structured Analysis and System Specification". Prentice Hall, 1979.

Dij74 Dijkstra, E.: "On the role of scientific thought". In: Dijkstra, E.: "Selected Writings on Computing: A Personal Perspective". Springer-Verlag, 1982, pp. 60-66.
Das entsprechende Originalpapier stammt von 1974.
Auch online verfügbar:
https://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html
[Zugriff am 08 02 2019].

For08 Ford, N.: "The Productive Programmer". O'Reilly Media, 2008.

Fow04 Fowler, M.: "Inversion of Control Containers and the Dependency Injection Pattern", 23.01.2004.
Online verfügbar:
http://www.martinfowler.com/articles/injection.html
[Zugriff am 08 02 2019].

Fow05 Fowler, M.: "Inversion of Control", 26.6.2005.
Online verfügbar:
https://martinfowler.com/bliki/InversionOfControl.html
[Zugriff am 08 02 2019].

Fow06 Fowler, M: "RoleInterface", 22.12.2006.
Online verfügbar:
https://martinfowler.com/bliki/RoleInterface.html
[Zugriff am 08 02 2019].

Fow15 Fowler, M.: "Yagni", 26. 05. 2015.
Online verfügbar:
http://martinfowler.com/bliki/Yagni.html
[Zugriff am 08 02 2019].

Fri04 Frick, S. empros gmbh: "Testgetriebene Entwicklung – ein Leitfaden" 9.10.2004.
Online verfügbar:
http://www.empros.ch/downloads/testgetriebeneentwicklung041009.pdf
[Zugriff am 08 02 2019].

Gam09 Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software". Addison-Wesley, 2009

Gam94 Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1994.

Gau70 Gauthier, R. L., Ponto, S. D.: "Designing System Programs". Prentice Hall, 1970.

Gol14 Goll, J.: "Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java": Springer Vieweg, 2014.

Hol89 Liebherr, K. J. und Holland I. M.: "Assuring Good Style for Object-Oriented Programs", 1989.
Online verfügbar:
http://homepages.cwi.nl/~storm/teaching/reader/LieberherrHolland89.pdf
[Zugriff am 08 02 2019].

Jam86 Geoffrey, J.: "The Tao of Programming". Info Books, 1986.
Online verfügbar:
http://www.mit.edu/~xela/tao.html
[Zugriff am 08 02 2019].

Jef00 Jeffries, R., Anderson A., Hendrickson Ch.: "Extreme Programming Installed". Addison-Wesley, 2000.

Joh88 Johnson, R. E., Foote, B.: "Designing Reusable Classes". Journal of Object-Oriented Programming, June/July 1988.
Online verfügbar:
https://www.cse.msu.edu/~cse870/Input/SS2002/MiniProject/Sources/DRC.pdf
[Zugriff am 08 02 2019].

keenes Keene, S: "Reliability, Law of Least Astonishment and the Interoperability Imperative".
Online verfügbar:
https://www.hawaii.edu/csati/summit/SKeene_Reliability_Society_Interoperability.pdf
[Zugriff am 08 02 2019].

Lie88 Lieberherr, K., Holland, I., Riel, A. "Object-Oriented Programming: An Objective Sense of Style". OOPSLA '88 Proceedings, Sept. 88.
Online verfügbar:
http://www.ccs.neu.edu/research/demeter/papers/law-of-demeter/oopsla88-law-of-demeter.pdf
[Zugriff am 08 02 2019].

Lie89 Lieberherr, K. J., Holland, L. : "Assuring Good Style by Object-Oriented Programs".
Online verfügbar:
http://homepages.cwi.nl/~storm/teaching/reader/LieberherrHolland89.pdf
[Zugriff am 08 02 2019].

Lie95 K. J. Lieberherr, "Adaptive Object-Oriented Software – The Demeter Method With Propagation Patterns". PWS Publishing company, 1995.

lieber K.J. Lieberherr, "Law of Demeter: Principle of Least Knowledge".
Online verfügbar:
http://www.ccs.neu.edu/home/lieber/LoD.html
[Zugriff am 08 02 2019].

liebfo I. H. Karl J. Lieberherr, "Formulations and Benefits of the Law of Demeter".
Online verfügbar:
http://www.ccs.neu.edu/research/demeter/papers/law-of-demeter/law-formulations/revision1/ss.tex
[Zugriff am 08 02 2019].

liebla Lieberherr, K. "Law of Demeter (LoD) ".
Online verfügbar:
http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/general-formulation.html
[Zugriff am 08 02 2019].

Lie89 Lieberherr, K. J., Holland, L. : "Assuring Good Style by Object-Oriented Programs".
Online verfügbar:
http://homepages.cwi.nl/~storm/teaching/reader/LieberherrHolland89.pdf
[Zugriff am 08 02 2019].

Lis09 Liskov, B. (2009): “The power of abstraction,” Turing Award Lecture.
[Video] Online verfügbar:
http://amturing.acm.org/vp/liskov_1108679.cfm, Minute 45:45
[Zugriff am 08 02 2019].

Lis87 Liskov, B.: "Data Abstraction and Hierarchy". ACM SIGPLAN Notices, 23(5): pp. 17-34, May 1987.

Mar02 Martin, R. C.: "Agile Software Development: Principles, Patterns, and Practices". Prentice Hall, 2002.

Mar08 Martin, R. C.: "Clean Code: A Handbook of Agile Software Craftsmanship". Prentice Hall, 2008.

Mar12 Martin, R. C.: "Agile Software Development: Principles, Patterns and Practises". Pearson Education, 2012.

Mar13 Martin, R. C.: "Agile Software Development: Principles, Patterns and Practises". Pearson New International Edition, 2013.

Mar14 Martin, R. C." The Single Responsibility Principle"
Online verfügbar:
https://8thlight.com/blog/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html
[Zugriff am 08 02 2019].

Mar96 R. C. Martin: "The Dependency Inversion Principle", in C++ Report, 1996.
Online verfügbar:

Ma296 Martin, R. C.: "The Interface Segregation Principle"; in: C++ Report, 1996.
Online verfügbar:

Ma396 R. C. Martin: "The Open-Closed Principle". In C++ Report, 1996.
Online verfügbar:

marsrp Martin, Robert C.,: "SRP: The Single Responsibility Principle".
Online verfügbar:
http://www.guillemette.org/uqam/mgl7361/assets/documents/SOLID_principles.pdf
[Zugriff am 08 02 2019].

Mey14 Meyer, B.: "Agile!: The Good, the Hype and the Ugly". Springer, 2014.

Mey88 Meyer, B: Object-Oriented Software Construction. Prentice Hall, 1988.

Mey92 Meyer, B: Applying "Design by Contract". IEEE Computer 25 (1992), Nr. 10, p. 40–51

Pag88 Page-Jones, Meilir: "The Practical Guide to Structured Systems Design". 2d ed. Yourdon Press Computing Series, 1988.

Par71 Parnas, D. L.:"On the criteria to be used in decomposing systems into modules". Carnegie-Mellon University in Pittsburgh, Pennsylvania, 1971.
Online verfügbar:
https://prl.ccs.neu.edu/img/p-tr-1971.pdf
[Zugriff am 08 02 2019].

Ste74 Stevens, W. P., Myers, G. J., Constantine, L. L.: "Structured design". IBM Systems Journal 13 (2), pp. 115–139, 1974.

Swe85 Sweet, R. E.:"The Mesa Programming Environment". Xerox Palo Alto Research Center, 1985 veröffentlicht in SIGPLAN (ACM Special Interest Group on Programming Languages): SLIPE '85 Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments, pp. 216-229, Seattle, Washington, USA – June 25 - 28, 1985, ACM New York, 1985.
Online verfügbar:
http://www.digibarn.com/friends/curbow/star/XDEPaper.pdf
[Zugriff am 08 02 2019].

Tho03 Thomas, D., Hunt, A.: "Der Pragmatische Programmierer". Hanser, 2003.

Wes10 Westphal, R: "Abhängigkeiten bewusster wahrnehmen". Blog Ralph Westphal, 7.9.2010.
Online verfügbar:
http://blog.ralfw.de/2010/09/abhangigkeiten-bewusster-wahrnehmen.html
[Zugriff am 08 02 2019].

Wes13 Westphal, R.: "Messaging as a programming model: Doing OOP as if you meant it". CreateSpace Independent Publishing Platform, 2013.

Wir71 Wirth, N.: "Program development by stepwise refinement". Communications of the ACM, 14(4), 1971.

You79 Yourdon, Edward; Constantine, Larry L.: "Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design". Prentice Hall, 1979.

Abkürzungsverzeichnis

API	Application Programming Interface
AWT	Abstract Window Toolkit
DbC	Design by Contract
DI	Dependency Injection
DIP	Dependency Inversion Principle
DRY	Don’t repeat yourself
FCOI	Favour Composition over Inheritance
GUI	Graphical User Interface
IEC	International Electrotechnical Commission
IoC	Inversion of Control
ISO	International Organization for Standardization
ISP	Interface Segregation Principle
KISS	Keep it simple, stupid
LoD	Law of Demeter
LSP	Liskovsches Substitutionsprinzip
PLA	Principle of Least Astonishment
OCP	Open-Closed Principle
SLA	Single Level of Abstraction
SOA	Service-Oriented Architecture
SoC	Separation of Concerns
SOLID	Akronym für SRP, OCP, LSP, ISP, DIP
SRP	Single Responsibility Principle

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*,
https://doi.org/10.1007/978-3-658-25975-4

UI	User Interface
UML	Unified Modeling Language
XP	Extreme Programming
YAGNI	You aren't gonna need it

Begriffsverzeichnis

- **Abgeleitete Klasse**
  Eine abgeleitete Klasse (Unterklasse, untergeordnete Klasse, Subklasse) wird von einer anderen Klasse, der sogenannten Basisklasse (Oberklasse, übergeordnete Klasse, Superklasse), abgeleitet. Eine abgeleitete Klasse erbt die Struktur (Attribute mit Namen und Typ) und das Verhalten (Methoden) ihrer Basisklasse in einer eigenständigen Kopie.

- **Abhängigkeit**
  Eine Abhängigkeit ist eine spezielle Beziehung zwischen zwei Modellelementen oder Codestücken, die zum Ausdruck bringt, dass sich eine Änderung des unabhängigen Elementes auf das abhängige Element auswirken kann.

- **Abstract Window Toolkit**
  Die Klassenbibliothek AWT ist die Vorgängerin der Klassenbibliothek Swing. AWT-GUI-Komponenten sind in Aussehen und Verhalten abhängig vom Betriebssystem. Aufgrund ihrer Abhängigkeit vom Betriebssystem werden sie als "schwergewichtig" bezeichnet. "Leichtgewichtige" Swing-GUI-Komponenten werden hingegen mit Hilfe der Java 2D-Klassenbibliothek durch die Java Virtual Machine selbst auf den Bildschirm gezeichnet und sind damit in Aussehen und Verhalten unabhängig vom Betriebssystem.

- **Abstrakte Klasse**
  Von einer abstrakten Klasse können keine Instanzen gebildet werden. Ein Grund dafür ist oftmals, dass die Klasse abstrakte (nicht implementierte) Methoden enthält.

- **Abstraktion**
  Unter Abstraktion versteht man das Weglassen irrelevanter Einzelheiten und die Konzentration auf das Wesentliche.

- **Aggregation**
  Eine Aggregation ist eine spezielle Assoziation, die eine Beziehung zwischen einer referenzierenden Komponente und einer referenzierten Komponente ausdrückt. Bei einer Aggregation ist die Lebensdauer eines referenzierten Objekts nicht mit der Lebensdauer des referenzierenden Objekts gekoppelt. Bei einer Komposition hingegen gehört ein Teil genau zu einem zusammengesetzten Ganzen, wobei ein Teil genauso lange wie das enthaltende Ganze lebt.

- **Architektur eines Systems**
  Die Beschreibung der Architektur eines Systems (Systemarchitektur) umfasst:

  – die Beschreibung der Zerlegung des Systems (Statik) in seine physischen Komponenten,
  – eine Beschreibung, wie durch das Zusammenwirken der Komponenten (Dynamik) die verlangten Funktionen erbracht werden sowie

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*,
https://doi.org/10.1007/978-3-658-25975-4

– eine Beschreibung der Strategie für die Architektur, d. h. für die Zerlegung (Statik) und für das Verhalten (Dynamik), damit im Team ein "shared understanding" der Architektur gegeben ist.

- **Assoziation**
  Eine Beschreibung eines Satzes von Verknüpfungen (Links) zwischen Objekten. Dabei verbindet eine Verknüpfung bzw. ein Link zwei oder mehr[94] Objekte als Peers (Gleichberechtigte). Eine Assoziation ist prinzipiell eine symmetrische Strukturbeziehung zwischen Klassen. Man kann aber die Navigation auf eine einzige Richtung einschränken.

- **Attribut**
  Den Begriff eines Attributs gibt es bei Klassen/Objekten/Assoziationen und bei Datenbanken. Die Objektorientierung hat diesen Begriff von dem datenorientierten Paradigma übernommen. Er bedeutet:

  1. Ein Attribut ist eine Eigenschaft einer Klasse oder eines Objekts.
  2. Ein Assoziationsattribut charakterisiert eine Assoziation.
  3. Eine Spalte innerhalb einer Relation (Tabelle) einer Datenbank wird auch als Attribut bezeichnet. Hierbei handelt es sich jedoch nicht um den Inhalt der Spalte selber, sondern um die Spaltenüberschrift. Ein Attributwert ist der konkrete Inhalt eines Spaltenelements in einer Zeile.

- **Äußere Qualität**
  Die äußere Qualität ist die Sicht des Kunden auf die Qualität einer Software.

- **Basisklasse**
  Eine Basisklasse (Superklasse, Oberklasse, übergeordnete Klasse) steht in einer Vererbungshierarchie über einer aktuell betrachteten Klasse.

- **Belang**
  Zusammenhängende Funktionen bilden in der Sprechweise von "Separation of Concerns" einen Belang (engl. concern). Verschiedene Belange sollen nach dem Prinzip "Separation of Concerns" sauber getrennt voneinander geführt werden.

- **Benutzungsabstraktion**
  Der Zugang zu einem Modul bzw. einer Komponente erfolgt über eine definierte, schmale Schnittstelle, welche die Leistung eines Moduls abstrahiert und dessen Services anbietet. Benutzer müssen nur wissen, welche Leistung durch ein Modul erbracht wird, nicht aber, wie dessen Implementierung erfolgt. Bei Vorliegen einer Benutzungsabstraktion kann die Verwendung eines Moduls erfolgen, ohne dass man dessen Aufbau kennt.

- **Black-Box-Wiederverwendung**
  Da keine internen Details der Objekte sichtbar sind und die Objekte als Black-Boxes erscheinen, wird "Objektkomposition" auch Black-Box-Wiederverwendung genannt [Gam09, p. 26].

[94] Man kann Multiplizitäten für Objekte einführen.

- **Classifier**
  Classifier ist ein Begriff aus dem Metamodell von UML. Die Metaklasse `Classifier` ist eine abstrakte Metaklasse.

  Es gibt eine Hierarchie von Classifiern. Die Wurzel ist die Metaklasse `Classifier` des Pakets `Classes::Kernel`. Von dieser Wurzel wird abgeleitet und ein ganzer Baum von Classifiern aufgebaut. Eine der Spezialisierungen der Metaklasse `Classifier` ist die Metaklasse `Class`, die Abstraktion einer Klasse. Instanzen der Metaklasse `Class` sind die Klassen.

  Jedes Modellelement von UML, von dem eine Instanz gebildet werden kann, ist ein Classifier. Ein Classifier bis auf die abstrakte Metaklasse `Classifier` kann instanziiert werden.

  Ein Classifier besitzt in der Regel eine Struktur und ein Verhalten. Schnittstellen besitzen als einzige Ausnahme meist keine Attribute, d. h., sie haben keine Struktur.

- **Client**
  Siehe Kunde

- **Concern**
  Das engl. Wort "concern" kann mit Belang, Anliegen, Interesse oder Verantwortung übersetzt werden.

- **Dependency Injection**
  Die Erzeugung von Objekten wird an eine dafür vorgesehene Instanz delegiert. Diese Instanz, die auch Injektor genannt wird, erzeugt zur Laufzeit auch die Verknüpfungen zwischen den Objekten. Damit wird die Abhängigkeit zwischen nutzendem und benötigtem Objekt stark abgeschwächt, aber nicht vollständig aufgelöst.

  Zur Kompilierzeit kennt ein Objekt einer nutzenden Klasse statt der konkreten Klasse nur eine Abstraktion der Klasse des vom nutzenden Objekt benötigten Objekts. Der Vertrag dieser Abstraktion muss vom Objekt der benutzten Klasse, welches der Injektor an die Stelle der Abstraktion setzt, eingehalten werden.

- **Dependency Lookup**
  Bei dem Konzept "Dependency Lookup" sucht ein Objekt, das ein anderes Objekt braucht, nach diesem anderen Objekt z. B. in einem Register, um die Verknüpfung mit dem benötigten Objekt herzustellen. Zur Suche braucht das suchende Objekt nur einen Schlüssel wie den Namen des gesuchten Objekts zu kennen und ist beim Suchen von dem anderen Objekt weitgehend entkoppelt.

  Ein suchendes Objekt muss nicht mehr die konkrete Klasse des von ihm benötigten Objekts, aber die Abstraktion der Klasse des benötigten Objekts kennen und einhalten, um die Methoden des gesuchten Objekts aufrufen zu können.

  Das suchende Objekt ist aber ferner auch vom Register abhängig, da es von diesem seine benötigten Objekte bezieht.

- **Design**
  Wird im Sinne von Entwurf verwendet.

- **Design Pattern**
  Siehe Entwurfsmuster

- **Domäne**
  Eine Domäne umfasst die Aufgaben einer bestimmten Anwendung, auch Fachkonzept genannt.

- **Entwurfsmuster** (engl. **design patterns**)
  Klassen oder Objekte in Rollen, die in einem bewährten Lösungsansatz zusammenarbeiten, um gemeinsam die Lösung eines wiederkehrenden Problems zu erbringen.

- **Extreme Programming**
  Extreme Programming (XP) von Kent Beck [Bec99] ist eine agile Methode, die im Gegensatz zu spezifikationsorientierten Methoden das Programmieren in den Vordergrund eines Projekts stellt sowie die Planung und die Erstellung von Dokumenten auf das Allernötigste beschränkt.

- **Feature**
  Ein Feature ist eine kleine, nützliche Funktion für den Kunden.

- **Framework**
  Ein Framework offeriert dem nutzenden System Klassen, von welchen das System abgeleitet werden kann und somit deren Funktionslogik erben kann. Ein Framework bestimmt die Architektur der Anwendung, also die Struktur im Großen. Es definiert weiter die Unterteilung in Klassen und Objekte, die jeweiligen zentralen Zuständigkeiten, die Zusammenarbeit der Klassen und Objekte sowie den Kontrollfluss [Gam09, p. 37].

- **Geheimnisprinzip**
  Siehe Information Hiding

- **Generalisierung**
  Eine Generalisierung ist die Umkehrung der Spezialisierung. Wenn man generalisieren möchte, ordnet man oben in der Vererbungshierarchie die allgemeineren Eigenschaften ein und nach unten die spezielleren, da man durch die Vererbung die generalisierten Eigenschaften wieder erbt. In der Vererbungshierarchie geht also die Generalisierung nach oben und die Spezialisierung nach unten.

- **Geschäftsprozess**
  Ein Geschäftsprozess ist ein Prozess der Arbeitswelt mit fachlichem Bezug. Er stellt eine Zusammenfassung verwandter Einzelaktivitäten, um ein geschäftliches Ziel zu erreichen, dar.

- **Information Hiding**
  Wird eine Einheit wie ein Modul oder eine Klasse gekapselt, werden nach außen nur wenige Informationen über die Services dieser Einheit freigegeben (schmale Schnittstelle). Die Implementierung wird gekapselt oder verborgen ("Information Hiding").

  "Information Hiding" sorgt beispielsweise dafür, dass die internen, privaten Strukturen eines Objekts einer Klasse nach außen unzugänglich sind. Nur der Implementierer einer Klasse kennt normalerweise die internen Strukturen, Methodenrümpfe und Servicemethoden eines Objekts. Implementierung und Schnittstellen werden getrennt. Die Daten eines Objekts sind nur über die Methodenköpfe einer Schnittstelle erreichbar.

- **Innere Qualität**
  Der Entwickler sieht die innere Qualität einer Software. Software muss eine korrekte, stabile und verlässliche Konstruktion darstellen, wobei der Code leicht änderbar und erweiterbar sein muss.

- **Instanz**
  Eine Instanz ist ein Objekt einer Klasse.

- **Instanziierung**
  Das Erzeugen einer Instanz einer Klasse.

- **Inversion of Control**
  Bei der Umkehr des Kontrollflusses gibt ein selbst geschriebenes Modul die Steuerung des Kontrollflusses an ein anderes Modul ab, das oft wiederverwendbar ist und in Gestalt eines Framework vorliegt.

- **Kapselung**
  Die Kapselung eines Moduls umfasst "Information Hiding" und den Zugriff auf ein Modul über eine Schnittstelle als Abstraktion der Leistung eines Moduls. Die Implementierung der Module ist verborgen und ist nur über deren Schnittstellen zugänglich (Benutzungsabstraktion).

- **Klasse**
  Eine Klasse stellt im Paradigma der Objektorientierung einen Datentyp dar, von dem Objekte erzeugt werden können. Eine Klasse hat eine Struktur und ein Verhalten. Die Struktur umfasst die Attribute. Die Methoden und ggf. der Zustandsautomat der Klasse bestimmen das Verhalten der Objekte.

- **Klassendiagramm**
  Ein Klassendiagramm zeigt insbesondere Klassen und ihre wechselseitigen statischen Beziehungen (Assoziationen, Generalisierungen, Realisierungen, Abhängigkeiten).

- **Komponente**
  Siehe Modul

- **Komposition**
  Eine Komposition ist ein Spezialfall einer Assoziation. Bei einer Komposition gehört ein Teil genau zu einem zusammengesetzten Ganzen. Ein Teil lebt genauso lange wie das enthaltende Ganze.

- **konkrete Klasse**
  Eine konkrete Klasse kann im Gegensatz zu einer abstrakten Klasse instanziiert werden, d. h. es können Objekte von dieser Klasse gebildet werden.

- **Konstruktor**
  Ein Konstruktor ist eine spezielle Methode. Ein Konstruktor trägt den Namen der Klasse, wird beim Erzeugen eines Objekts aufgerufen und dient zu dessen Initialisierung. Ein Konstruktor hat keinen Rückgabetyp und kann nicht vererbt werden.

- **Kunde**
  Hier: Teil eines Computerprogramms, welches gewisse Objekte oder Funktionalitäten benutzt.

- **logisch zusammenhängend**
  Aus Sicht des Problembereichs stark zusammenhängende Module werden als "logisch zusammenhängend" bezeichnet.

- **Lösungsbereich**
  Im Problembereich ist man in einer idealen Welt mit unendlicher Performance und keinem technischen Fehler, im Lösungsbereich ist man in der physischen Welt mit ihren Einschränkungen und technischen Fehlern.

  Im Problembereich ist man in der Welt der Logik der Anwendung. Im Lösungsbereich ist man in der Welt der Konstruktion.

  Im Problembereich gibt es kein technisches System, im Lösungsbereich gibt es ein technisches System.

  Im Problembereich kümmert man sich um die Verarbeitungsprozesse in einer idealen Gedankenwelt (Essenz des Systems), im Lösungsbereich um alle Funktionsklassen.

- **Mock-Objekt**
  Ein Mock-Objekt ist ein Objekt, das als Platzhalter für echte Objekte innerhalb eines Komponententests verwendet wird. Der Begriff kommt aus dem Englischen und kann mit "etwas vortäuschen" übersetzt werden. Ein Mock-Objekt hat eine Logik implementiert. Ein Mock-Objekt kann beim Testen wie das tatsächliche Objekt aufgerufen werden und liefert vorher festgelegte, sinnvolle Werte zurück.

- **Modul**
  Es gibt keine einheitliche Definition. In diesem Buch wird ein Modul folgendermaßen verwendet:

Ein Modul kapselt seine Implementierung. Solange die Schnittstelle eines Moduls nicht verändert wird, kann man im Inneren eines Moduls beliebige Änderungen durchführen. Die Leistung eines Moduls steht über schmale Schnittstellen nach außen zur Verfügung. Die Verwendung eines Moduls kann also erfolgen, ohne dass man den Aufbau eines Moduls kennt (Benutzungsabstraktion).

Wenn die Schnittstellen der Module feststehen, können die Module unabhängig voneinander entwickelt werden (Parallelität der Entwicklung der Module). Module sind in sich logisch sehr stark zusammenhängend und tragen nach dem "Single Responsibility Principle" von Robert C. Martin nur eine einzige Verantwortlichkeit. Module sollen getrennt getestet werden können.

- **Oberklasse**
  Siehe Basisklasse

- **Objekt**
  Siehe Klasse

- **öffentliche Methode**
  Siehe Schnittstellenmethode

- **Over-Engineering**
  Man spricht von Over-Engineering, wenn ein Produkt in höherer Qualität oder mit mehr Aufwand erstellt wird, als es der Kunde wünscht. Dabei wird oft die Zahlungsbereitschaft des Kunden überschritten.

- **Paket in Java**
  Ein Paket in Java dient zur Gruppierung von Klassen und Schnittstellen sowie von Paketen in einem Paket. Ein Paket stellt einen Namensraum dar, ist eine Einheit für den Zugriffsschutz und erleichtert die Übersicht.

- **Paradigma**
  Ein Paradigma ist ein Denkkonzept.

- **Polymorphie von Objekten**
  Polymorphie von Objekten bedeutet Vielgestaltigkeit. Ein Objekt eines Subtyps kann auch in Gestalt der entsprechenden Basisklasse auftreten.

- **Problembereich**
  Der sogenannte Problembereich oder Problem Domain ist der Bereich, der automatisiert werden soll. Es ist derjenige Teil der realen Welt, der später durch die zu realisierende Software abgedeckt werden soll.

- **Realisierung**
  Eine Realisierung ist eine statische Beziehung zwischen zwei Elementen, in der das eine Element einen Vertrag spezifiziert und das andere Element sich verpflichtet, diesen Vertrag bei der Realisierung einzuhalten. Realisierungsbeziehungen gibt es beispielsweise bei Klassen, die Schnittstellen implementieren.

- **Schnittstelle**
  Eine Schnittstelle (engl. interface) stellt das Bindeglied zwischen den Nutzern eines Moduls und der Implementierung dieses Moduls dar. Nur die Schnittstelle eines Moduls ist nach außen sichtbar. Eine Schnittstelle erlaubt es, auf die Implementierung eines Moduls zuzugreifen, ohne die Implementierung dieses Moduls zu kennen. Sind Module nicht gekoppelt, können sie nicht zusammenarbeiten.

  Eine Schnittstelle stellt eine Abstraktion des entsprechenden Moduls und dessen angebotenen Leistungen dar. Auf der Ebene von Klassen beinhaltet eine Schnittstelle die Methodenköpfe – also Methodennamen, Rückgabetypen und Übergabeparameter – von öffentlichen Methoden der Klasse. Eine bestimmte Schnittstelle kann alle Operationen oder aber auch nur einen Teil der Operationen einer Klasse repräsentieren. Eine Schnittstelle spezifiziert einen Vertrag, den das realisierende Element erfüllen muss.

  Schnittstellen können in objektorientierten Programmiersprachen oftmals mit einer formalen Syntax beschrieben werden wie beispielsweise in Form eines Interface in Java.

  In der Vergangenheit sah ein Interface in Java keine syntaktische Möglichkeit vor, Methoden zu definieren. Dies ist erst seit Java 8 möglich. Schnittstellen in Java können jetzt auch den Charakter einer abstrakten Klasse[95] haben und dedizierte Methoden vorgeben. Die Einführung von Implementierungen in einer Schnittstelle untergräbt jedoch die eigentliche Bedeutung einer Schnittstelle.

  Eine Schnittstelle im Sinne dieses Buches enthält grundsätzlich keine Implementierungen.

- **Schnittstellenmethode**
  Über den Methodenkopf einer Schnittstellenmethode (öffentlichen Methode) kann man auf die Daten eines Objektes zugreifen.

- **schüchterner Code**
  Bei schüchternem Code nach dem "Law of Demeter" darf ein Objekt nicht über ein zweites Objekt auf ein drittes Objekt zugreifen.

- **shared understanding**
  "Shared understanding" ist ein Begriff aus der Literatur, siehe beispielsweise [Pat14]. Er bedeutet, dass das wesentliche Wissen über das System und das Projekt zwischen allen Projektbeteiligten geteilt ist.

- **Spezialisierung**
  Siehe Generalisierung

- **Struktur**
  Die Struktur eines Systems ist sein statischer Aufbau.

[95] Abstrakte Klassen dürfen im Allgemeinen neben abstrakten Methoden auch für einige Methoden Implementierungen enthalten – wie neuerdings auch die Interfaces in Java.

- **Stub**
  Stub bedeutet Platzhalter. Ein Stub ist ein vorläufiger, einfacher Ersatz für eine andere Komponente, die noch nicht erstellt ist, aber benötigt wird, damit ein Programm ablauffähig wird und getestet werden kann. Ein Stub bildet die noch fehlende Komponente in einfachster Weise nach. Er hat keine Logik. Die Aufgabe des Stubs ist es dabei nur, die Durchführung eines Aufrufs zu gewährleisten. Damit ist die Lauffähigkeit eines Programmes hergestellt. Ein Stub gibt beim Aufruf einer seiner Methoden einen festen Wert zurück.

- **Subklasse**
  Siehe abgeleitete Klasse

- **Subtyp**
  Eine abgeleitete Klasse stellt einen eigenen Typ dar, der dann als Subtyp bezeichnet wird, wenn das liskovsche Substitutionsprinzip (siehe Kapitel 6.2) und damit der Vertrag (siehe Kapitel 6.1) der Basisklasse eingehalten wird. Dann hat die Interpretation "is a" ihre Berechtigung, da ein Subtyp alles hat (Methoden und Datenfelder), was eine Basisklasse auch hat, und sich die Objekte des Subtyps genau so verhalten wie die Objekte der Basisklasse, wenn die Objekte des Subtyps über Methoden der Basisklasse angesprochen werden.

- **Superklasse**
  Siehe Basisklasse

- **Unterklasse**
  Siehe abgeleitete Klasse

- **Verantwortlichkeit**
  "A reason to change" (Robert C. Martin) charakterisiert eine sogenannte Verantwortlichkeit (engl. responsibility) einer Klasse.

- **Vererbung**
  In der Objektorientierung kann eine Klasse von einer anderen Klasse statisch zur Kompilierzeit erben bzw. von ihr abgeleitet werden. Durch die Vererbung besitzt die abgeleitete Klasse automatisch alle Attribute und Methoden der Klasse, von der sie abgeleitet wird.

- **Vererbungshierarchie**
  Durch die Vererbungsbeziehung zwischen Klassen entsteht eine Hierarchie: Abgeleitete Klassen werden ihren Basisklassen untergeordnet bzw. Basisklassen sind ihren abgeleiteten Klassen übergeordnet.

- **Verhalten**
  Im Verhalten eines Systems kommt seine Funktionalität zum Ausdruck.

- **Vertrag**
  Ein Vertrag regelt insbesondere die Beziehungen zwischen Aufrufer und Aufgerufenem.

- **White-Box-Wiederverwendung**
  Wiederverwendung durch Unterklassenbildung wird oft auch White-Box-Wiederverwendung genannt, da die öffentlichen und geschützten Elemente der Basisklasse in der abgeleiteten Klasse direkt sichtbar sind, wenn Sie nicht verdeckt bzw. überschrieben werden.

Index

Abhängigkeit 17
  Erklärung 12
  logische 29
  statische 13
  zur Kompilierzeit 13
  zur Laufzeit 28
Abhängigkeiten
  Abschwächung 31
  Entstehung 13
  Kopplung 31
Abstraktion 5
  Identifikation Teilsysteme 5
  Klassen 6
  schmale Schnittstellen Module 5
Abstraktion Services
  durch höhere Ebene vorgegeben ... 75
Aggregation 14, 133
  Programmbeispiel 22
Analyse 163
Änderungsgrund
  einziger 85
Assoziation 14
  bidirektional 17
  Programmbeispiel 17
Aufrufhierarchie
  klassischer Entwurf 74
Belang ..... *siehe* Concern
Benutzungsabhängigkeit ..... *siehe* use-Beziehung
Benutzungsabstraktion 52, 59
Benutzungsbeziehung ..... *siehe* use-Beziehung
Beobachtermuster 146
  Callback-Schnittstelle 147
  Pull-Prinzip 147
  Push-Prinzip 147
Beziehung
  statische nach UML 13
Black-Box-Verhalten 132
Black-Box-Wiederverwendung 135
Callback-Schnittstelle 146, 150
Cohesion 56, 57
  strong 56, 85
Constructor Injection 96
  Abstraktion Klasse benötigtes Objekt 99
  Programmbeispiel 96
Concern 61
Coupling 55, 57
Dependency Injection 33, 88, 89, 95, 96, 102
  Abstraktion Klasse benötigtes Objekt 33, 89, 96, 102
  Constructor Injection 96
  Interface Injection 96
  Setter Injection 96
  Varianten 96
Dependency Inversion 73, 77
  Abhängigkeit invertiert 75
Dependency Inversion Principle 73
  Abstraktion 75
  Bewertung 77
  Nachteile 78
  Vertrag der Abstraktion 76
  Vorteile 78
  Wiederverwendbarkeit Elemente der höheren Ebene 74
Dependency Lookup 33, 88, 89, 101
  Abstraktion Klasse benötigtes Objekt 33, 89, 90, 102
  Programmbeispiel 91
  Schlüssel 33, 90
  Singleton 92
  Verknüpfung zur Laufzeit 90
Design by Contract 107, 160
  Bewertung 113
  konzeptionelles Beispiel 113
  Nachteile 113
  Vertrag 107
  Vorteile 113
Design to Test 36, 37
  Bewertung 38
  komponentenweise Strategie 37
  Nachteile 38
  Vorteile 38
divide and conquer ..... *siehe* Teile und Herrsche
divide et impera ..... *siehe* Teile und Herrsche
Don’t repeat yourself 44
Don't call us, we'll call you ..... *siehe* Hollywood Principle

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*,
https://doi.org/10.1007/978-3-658-25975-4

DRY 44
Aktualisierungsprobleme 46
Anwendbarkeit 45
Bewertung 46
Nachteile 47
Replikate 45
Vorteile 46
Du wirst es nicht brauchen .... *siehe* You aren't gonna need it
Entwurf....163
Entwurf durch Verträge .... *siehe* Design by Contract
Entwurfsprinzipien
Bedeutung für die Qualität....160
generelle Einfachheit....156
modulare Struktur von Systemen .156
Reduktion der Komplexität 40
evolvierbar .... *siehe* wandelbar
Frameworks
Callback-Schnittstellen 144
Funktion
technisch 163
Verarbeitungsfunktion 163
Geheimnisprinzip .... *siehe* Information Hiding
Generalisierung 15
Gesetz von Demeter .... *siehe* Law of Demeter
Grundfunktionen der Informationstechnik 163
Hilfsmethoden . *siehe* Service-Methoden
Implementierung 26
Abstraktion 29
Information Hiding....6, 59
Bewertung 60
Get- und set-Methoden 66
Klassen 6
Vorteile 61
Injektion 95
Injektor....33, 90, 95
Interface Injection 96
Abstraktion Klasse benötigtes Objekt 101
Programmbeispiel 100
Interface Segregation Principle....81
Bewertung 84
Konzeptionelles Beispiel 82
Nachteile 84
Rollen-Schnittstelle....83
Vorteile 84
Invariante 108, 109, 110
beim Überschreiben 113
nicht aufweichen 113
nur verschärfen 113
Schnittstellenmethode....111
Service-Methoden 111
Verletzung 111
Inversion
Dependency Injection....148
Inversion of Control....144
Anmeldeverfahren....148
Bewertung 148
Callback-Schnittstelle....146
Dependency Lookup 148
ereignisorientierte Programmierung 145
Framework 148
Hollywood Principle....144, 145
Listener 146
Nachteile 149
Programmbeispiel 149
Vorteile 148
'is a'-Beziehung....24, 128
Kapselung....6
Benutzungsabstraktion....6
Information Hiding 6
Keep it simple, stupid....40
KISS 40
Bewertung 41
Einfachheit 41
Nachteile 41
Vorteile 41
Klasse
Aufrufschnittstellen
Schnittstellenmethoden 8
Daten 7
Rümpfe Schnittstellenmethoden 7
Servicemethoden 7
Klassenform....69
Klasseninvariante 109
Kohäsion.... *siehe* Cohesion
starke 56
Komplexität....36
Komposition....14, 133
Programmbeispiel 20
Konzepte
Bedeutung für die Qualität....160
konzeptionelles Beispiel
Design by Contract....113
Interface Segregation Principle 82
liskovsches Substitutionsprinzip ...118
logische Abhängigkeit 30

Korrektheit 106
äußere Qualität....106
Design by Contract....106
Erfüllungsgrad Kundenforderungen 106
innere Qualität....106
Konstruktion 106
Vermeiden von Überraschungen..160
Korrektheit Konstruktion
Einfachheit 106
Fehlervermeidung 106
LSP 106
shared understanding....106
Überprüfungen 106
Law of Demeter 63
Argument classes....68
Ausprägungen....63, 66
Bewertung 72
Current Class 68
'Don't talk to strangers' 64
Fremde....64
Freunde....64, 68
Geheimnisprinzip....65
Immediate Part Classes 68
Klassenform 66, 69
Kommunikation zwischen zwei nicht direkt benachbarten Objekten....66
Kopplungen 72
minimierte Klassenform....69
Modularität 64
Nachteile 72
Objektform....66, 68
Objektorientierung....63
'Schüchterner' Code 64
strikte Klassenform....69
Strong ~....66, 67
Testbarkeit 72
Übersichtlichkeit 72
Verkettung objektorientierter Methodenaufrufe....63
Vorteile 72
Wandelbarkeit 72
Wartbarkeit....72
Weak ~....66
Law of Demeter for Functions/Methods ....*siehe* Law of Demeter
Law of Goodstyle *siehe* Law of Demeter
liskovsches Substitutionsprinzip 114, 116, 128, 160
Bewertung....119
konzeptionelles Beispiel....118
Nachteile 119
reines Erweitern 117
Überschreiben....117
Vorteile....119
Listener
pollend 146
logische Abhängigkeit....29
konzeptionelles Beispiel....30
Laufzeit 29
konzeptionelles Beispiel
Open-Closed Principle 127
Loose Coupling....55, 57
Änderbarkeit....55
Benutzungsabstraktion....57
Erweiterbarkeit 55
Schnittstellen....57
Stabilität 55
Testbarkeit 54, 55
Wartbarkeit....55
Wiederverwendbarkeit 55
Loose Coupling and Strong Cohesion53
Änderbarkeit....54
Bewertung....58
Erweiterbarkeit 54
Stabilität 54
Vorteile....58
Wartbarkeit....54
Lose Kopplung.... *siehe* Loose Coupling
Lösungsbereich 163
Mock-Objekt....77, 102
Modul....2, 4
Abstraktion 3, 5
Änderungswahrscheinlichkeit....4, 60
Benutzungsabstraktion....3, 5, 6
Cohesion....53
einzige Verantwortlichkeit 4, 5, 55, 60
einziger Grund für Änderung....60
Information Hiding....3, 5, 6
innerer Zusammenhalt 53
Kapselung 5
Kapselung Designentscheidungen....4, 60
'schmale' Schnittstelle 3, 6
Schnittstelle....57
Single Responsibility Principle 5
Stabilität 4
Strong Cohesion 53
Verantwortlichkeit....4
Vorteile....8
modular....2
Modularisierung 2
Vorteile....8
Module

Abschwächung wechselseitiger Abhängigkeiten 53
Nachbedingung....108, 110, 112
nicht aufweichen....112
nur verschärfen 112
Objektform 68
Objektkomposition 135
Abhängigkeit von Abstraktion....136
Black-Box-Sicht der beteiligten Klassen....136
Kapselung nicht aufgebrochen....136
Objektorientiertes System
Aufrufhierarchie....73
Observer Pattern.... ....*siehe* Beobachtermuster
öffentliche Methoden.... ....*siehe* Schnittstellenmethoden
Once and only once ....*siehe* DRY
Open-Closed Principle
Bewertung....126
Geschlossenheit....125
Geschlossenheit lauffähiger Code126
Geschlossenheit Quellcode....126
Geschlossenheit Spezifikation 126
konzeptionelles Beispiel....127
Nachteile 126
Objektkomposition....124
Offenheit....125
Offenheit lauffähiger Code 126
Offenheit Quellcode....126
Offenheit Spezifikation 126
Programmbeispiel 128
Vererbung 124
Vorteile 126
Open-Closed Principle 125
Polling 145
Polymorphie
Methoden 114
Objekte....115
Überschreiben....115
Vererbungshierarchie 115
Postcondition ..... *siehe* Nachbedingung
Precondition....*siehe* Vorbedingung
Principle of Least Astonishment....119, 120
Bewertung....121
Fehlervermeidung 120
Nachteile 121
Verständlichkeit....120
Vorteile 121
Principle of Least Knowledge.... .... *siehe* Law of Demeter
Problembereich....163
Programmbeispiel
Aggregation....22
Assoziation....17
Constructor Injection 96
Dependency Lookup 91
Interface Injection....100
Inversion of Control 149
Komposition 20
Realisierung 26
Setter Injection 99
statische Abhängigkeit 27
Strong Law of Demeter 69, 71
Vererbung 24
Weak Law of Demeter....69, 70
Programmbeispiel
Open-Closed Principle 128
Programmiere gegen Schnittstellen, nicht gegen Implementierungen ...138
abstrakte Klasse....138
Bewertung....140
Nachteile 140
Schnittstelle....138, 139
Vorteile....140
Wiederverwendbarkeit 140
Qualität
Evolvability .... *siehe* Qualität:Wandelbarkeit
Korrektheit....106
Wandelbarkeit 157
Qualitätsmerkmal
Cohesion....57
Coupling....57
Qualtäten eines Systems
funktional....163
Realisierung....26
Programmbeispiel 26
Realisierungsabhängigkeit....76
Responsibility... *siehe* Verantwortlichkeit
Rollen-Schnittstelle....81
Schnittstelle
Aufrufhierarchie....59
'fett'....81
funktionale Eigenschaften 59
Kohäsion 81
nicht funktionale Eigenschaften....59
Peers....59
'polluted' 81
Semantik der Funktionen 58

schnittstellenbasierte Programmierung 125
Schnittstellenmethoden.....6
Separation of Concerns 61, 87
Bewertung 62
elementares Ordnungsprinzip 87
Prozess der Zerlegung 87
Vorteile 62
Service-Methoden.....7
Services
vorgegeben durch höhere Klasse...76
Setter Injection 96
Abstraktion Klasse benötigtes Objekt 100
Programmbeispiel 99
Single Level of Abstraction Principle..47
Bewertung 48
Nachteile 49
Vorteile 48
Single Responsibility Principle 85, 87
Bewertung 87
Designprinzip Modularisierung 87
einzige Kraft 86
einzige Verantwortlichkeit.....86
einziger Änderungsgrund 85
Nachteile 88
Strong Cohesion.....85
Verantwortlichkeit 86
Vorteile 87
Single-Source-Prinzip .....*siehe* DRY
Stabilität 124
bei Programmänderungen.....124
statische Abhängigkeit
Programmbeispiel 27
statische Abhängigkeit im Allgemeinen
Abhängigkeit nach UML 17
Aggregation nach UML.....14
Assoziation nach UML.....14
Generalisierung nach UML.....15
Komposition nach UML 14
Vererbung .....
.....*siehe* statische Abhängigkeit im Allgemeinen:Generalisierung nach UML
Strong Law of Demeter 67
Information Hiding 67
Programmbeispiel 69, 71
Systemarchitektur
testbar 36
technische Funktion 163
Datenhaltung.....163
Ein- und Ausgabe.....163
Parallelität/IPC 163
Rechner-Rechner-Kommunikation 163
Sicherheit 163
Teile und Herrsche 3, 5, 36
Bewertung 37
Nachteile 37
Teilprobleme 36
Vorteile 37
Teilsystem
Komplexität 3
Umkehr Abhängigkeit .....75, ...*siehe* Dependency Inversion
Umkehr Besitz Abstraktion 75
Umkehr der Abhängigkeiten ....*siehe* Dependency Inversion
Umkehr der Kontrolle..... ..... *siehe* Inversion of Control
use-Beziehung.....28, 76
Verantwortlichkeit 86
Verarbeitungsfunktion.....163
Vererbung.....135
Programmbeispiel 24
Verkettung Methodenaufrufe 64
Vertrag.....59, 107, 108, 116
Invariante 108
Klasse 109
Nachbedingung 108
überschreibende Methode 111
Vorbedingung.....108
Vorbedingung 108, 109, 112
nicht verschärfen.....112
nur aufweichen.....112
wandelbar 31, 52
Weak Law of Demeter 66
Programmbeispiel 69, 70
White-Box-Verhalten.....132
White-Box-Wiederverwendung 135
Wiederverwendung
Elemente der höheren Ebene 77
Wrapper-Methode
Kommunikation mit Fremden 66
Wrapper-Methode Strong LoD für
Attribute Basisklasse.....67
Kapselung 67
YAGNI 42
Bewertung 44
klare Forderungen.....43
Nachteile verkehrter Anwendung ...44
Over-Engineering 42
Premature Generalization 42
Spekulative Funktionen.....42

spekulative Generalisierung 42
Vorteile 44
You aren't gonna need it 42
Ziehe Objektkomposition der Klassenvererbung vor 132
Bewertung 136
Nachteile 137
Vorteile 136
Zusicherung
Invariante 108, 110
Nachbedingung 108, 110
Vorbedingung 108, 109