TECHNISCHE UNIVERSITÄT DRESDEN

DRESDEN concept

Softwaretechnologie II

Dr.-Ing. Sebastian Götz
Technische Universität Dresden
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie
http://st.inf.tu-dresden.de/teaching/swt2

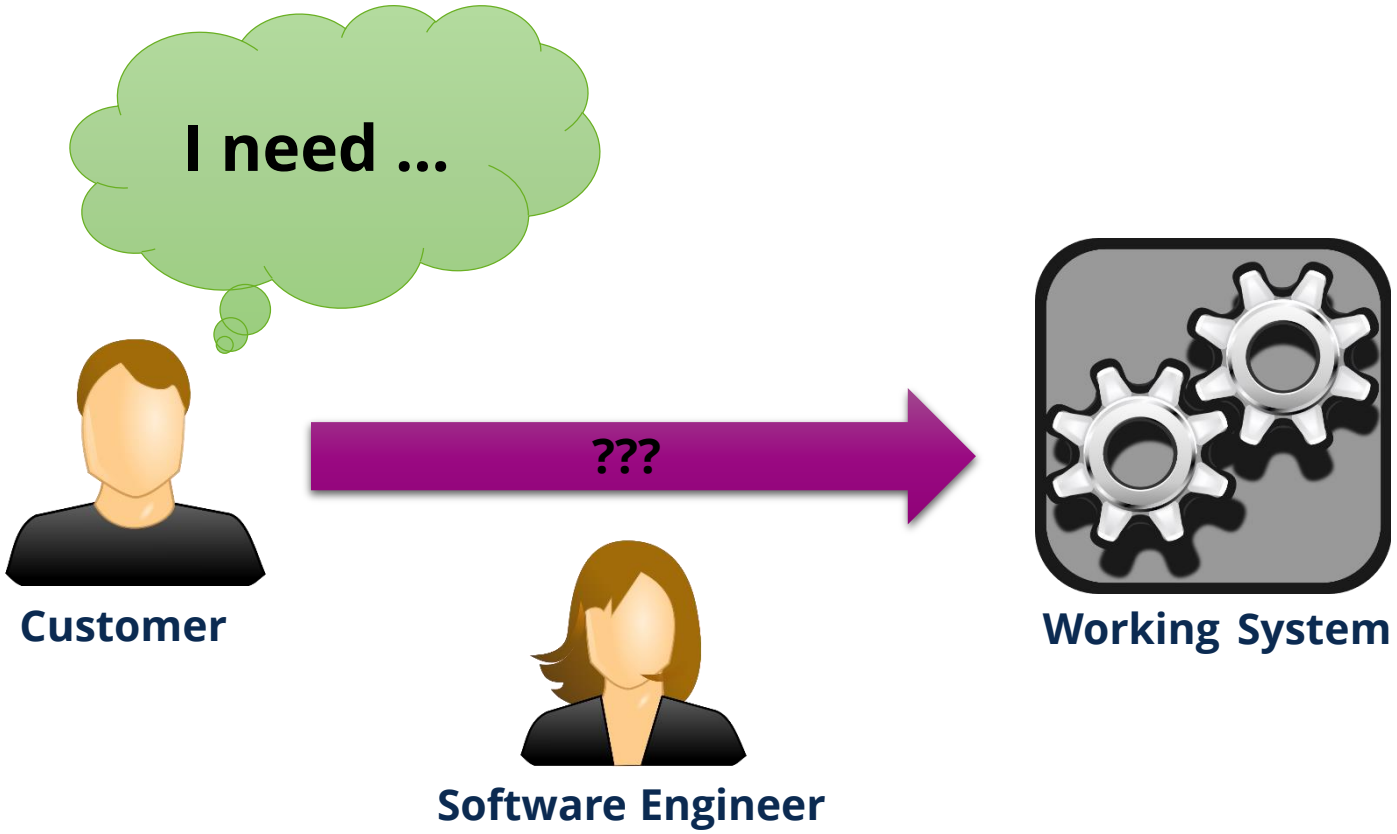
WS 19/20, 15.10.2019

COURSE OUTLINE

Course Outline

How to get from a set of informal and incomplete requirements (the customer‘s mind) to a working software system?

Here: non-object-oriented development methods



Main Goals

- Learn about “forward engineering” of software
  - Technology, process, experiences, human conditions
  - What a software engineer may sell (services, products, product lines…)
  - Modeling with big models
  - The influence of logic and graph rewriting (Because almost all requirements and design notations are graph-based)
  - Requirements engineering, testing, and software quality
  - Other design methods than Object-orientation
  - Software Product Lines
- Learn about the behavioral language Petri Nets, and its derivatives
  - Earning money with software (introduction to business models)
- Get as many ideas as possible (broad overview)
  - NOT: technical in-depth teaching (this is left to other courses)

The top level of the V-model: Requirements, Validation, Software Quality

- Know about requirement specification
- Software Quality:
  - Contract-based development
  - Know what inspections are
  - Know about maintenance problems
  - Know about basic testing concepts
- Model quality
  - Model analysis
  - Model structuring

Design

- Know different forms of design methods
  - functional, object-oriented, data-oriented
- Know behavioral methods to generate code for verifiable specifications
  - Petri nets
- Get overview of software processes
  - MDA, XP, V-model, ...
- Know about “software architecture” and architectural styles

Course Parts

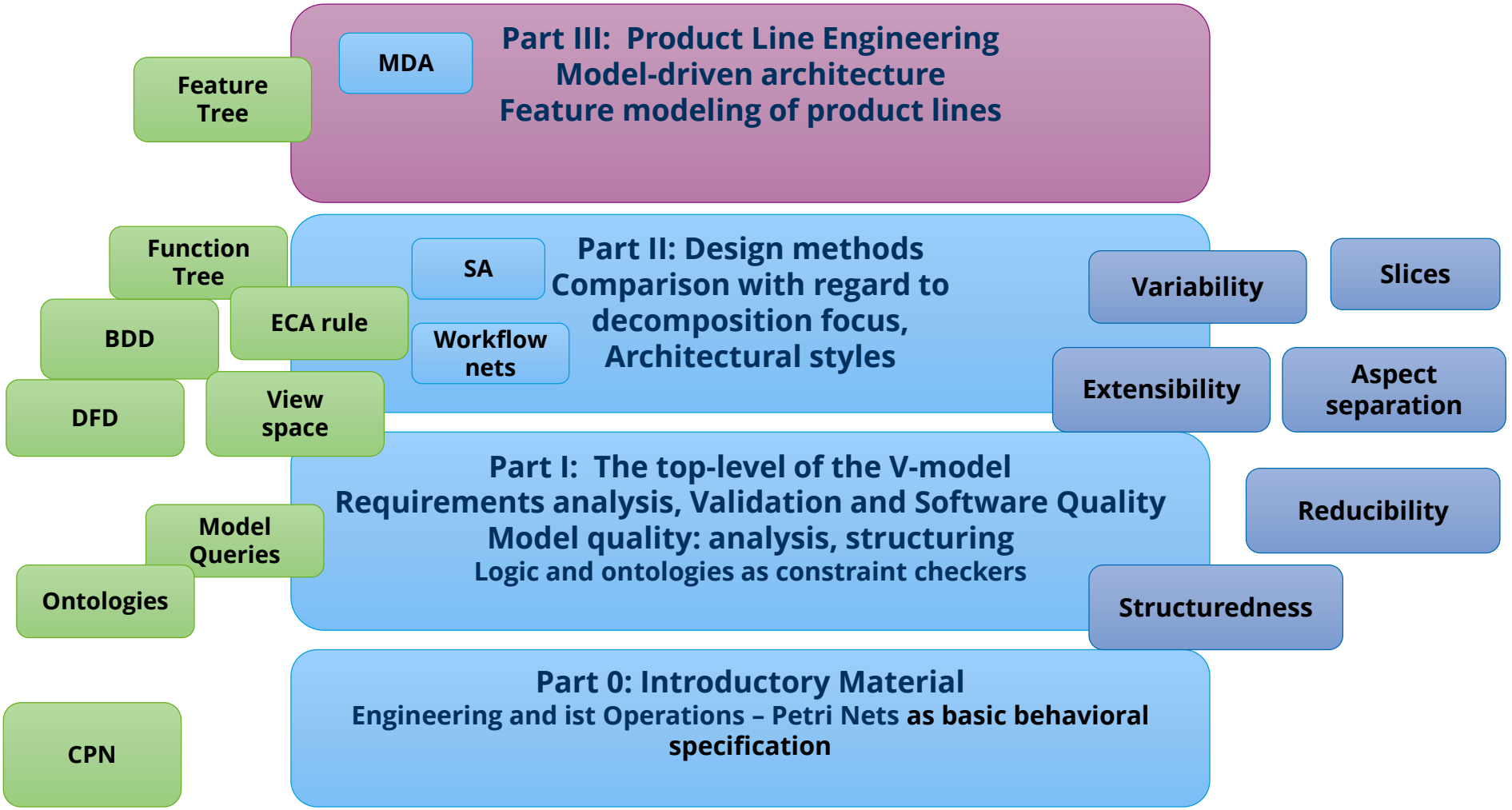
Part III: Product Line Engineering
Model-driven architecture
Feature modeling of product lines

Part II: Design methods
Overview, Comparison of Design Methods with regard to Decomposition focus, Extensibility

Part I: The top-level of the V-model
Requirements analysis
Validation and Software Quality
Model quality: analysis, structuring

Part 0: Introductory Material
Engineering – Petri Nets

Course Structure: Learning Everything for Engineering Product Lines



MOTIVATION

Software Bugs

Peter G. Neumann http://www.risks.org
The *Risk Digest* collects all possible software bugs and news related to software-related issues

Mercedes console display with conflicting information
<Henry Baker <hbaker1@pipeline.com>>
Fri, 14 Dec 2007 10:48:39 -0800

The console display says "check engine" & "no malfunction" at the same time!
Dueling messages!

It is supposed to say "check engine" & "1 malfunction", if "check engine" is the only malfunction being reported.

What is “Big”?

Class	Lines of Code	Person years
Very small	<1000	<0.2
Small	1000 - 10000	0.2 - 2
Medium	10000 - 100000	2 - 20
Large	100000 – 1 Mio	20 - 200
Very large	>1. Mio	>200

Quelle: Gumm/Sommer, Einführung in die Informatik, 4. Auflage, 2000

Big Systems

- Telefone switching software Siemens EWSD (Version 8.1):
  - 12,5 Mio. lines of code
  - ca. 6000 person years
- ERP-Software SAP R/3 (Version 4.0)
  - ca. 50 Mio. lines of code
- Total amount of lines of code in software (around 2000):
  - Credit Suisse 25 Mio. Code-Zeilen
  - Chase Manhattan Bank: 200 Mio. Code-Zeilen
  - Citicorp Bank: 400 Mio. Code-Zeilen
  - AT&T: 500 Mio. Code-Zeilen
  - General Motors: 2 Mrd. Code-Zeilen

EWSD = Elektronisches Wählsystem Digital (Siemens)
ERP = Enterprise Resource Planning
SAP: Deutscher Software-Konzern

Permanent Software Crisis?

- "Software Crisis":
  - Errors in computer systems are mostly software faults
  - Software projects are often too late
  - Software development costs too much
- Software Crisis started in 1968, but exists still today
- Modularity was discovered
- Much larger software systems
- Massive testing necessary
- Today: distributed compute infrastructures

> To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.
>
> E. W. Dijkstra, ICSE 1968

The Industry

- Top Players: IBM, Google, Microsoft, HP, Google, Oracle, SAP, Amazon
- 2/3 standard software : 1/3 individual software (with growing rate)
- Life Cycle of Classic Software
  - Average: 5 – 15y
  - max > 35 y (control software, certified systems, data bases)
    - Programmers die out
  - Development time: 1 – 3y
  - More an more software is developed, which is not long-lasting
    - Used 1y or less (e.g., many apps for mobile devices)

Costs

- Hardware speed doubles every 2 years, but software productivity increases only about < 5%/y
- Costs: Commercialization is rather difficult
  - acad. Prototype / acad. Product / Product = 1 : 3 : 3
- Relation of development and maintenance 40:60 up to 20:80
  - Development and maintenance are done by different departments
- Costs: Extreme Requirements
  - **Certification**: show the software and its development process to a certification agency (TÜV, etc.)
  - **Example**: **Pension Insurance**: certified software must be executable after 40 years
  - German pension rules of the 50s must be processed today
- Nobody knows the details anymore
- Solution: write an interpreter for the old assembler
- This has happened twice..

Cost Example: The Year 2000 Problem

- COBOL programmers saved space and stored only the last two digits of the year
  - In the 70s, programs should only live 20 years
- In 2000, catastrophes were prophesied
  - Power plants?
  - Pension insurances (birth dates)
- From 1996 on, the industry panicked
  - Spent enormous amounts to update software
- New systems got installed
  - SAP R/3 with date data type
- Rewriting didn’t work
  - Programmers didn’t trust the rewrites
  - Solution: sliding window technique

Cost Example: The Euro Introduction

- End of 2001, many countries introduced the Euro
- Too bad: on paper, the Euro was introduced 2 years before
  - Some companies had to maintain double booking for 2 years
  - At least for some months in 2002
  - Double booking was very costly: accounts had to be printed in two currencies
- How to test the transition?
  - In May 2001, the Dresdner bank ran a test
  - Which failed,.. And produced many wrong money transfers!
- Many people worked day and night...

- Same issues may come up for Japan: https://www.theguardian.com/technology/2018/jul/25/big-tech-warns-japan-millennium-bug-y2k-emperor-akihito-abdication

Cost Example: Telecommunication

- Telecommunication needs 24/7 systems with permanent uptime: Failure < 1 h./40 y., working rate 99.999%
- One second failure may cost $5Mio
- Telecommunication software product line
  - 20-30 000 Module of 1000 loc (lines of code)
  - Single product has 2-8000 modules
- Necessary: 5000 persons/7years.
- Costs ca. 7 billion €.

Human Problems

- Programmers are not educated well
  - To develop
  - To communicate
- Software construction is a social process
  - It's hard to organize people
- Software stays, the people go
  - Software evolves, many versions
- Projects run out of time
  - How to control?
- Programmer Productivity – Rules of Thumb
  - System software: 1000-2000 loc/y
  - System like Software: 5000 loc/y
  - Application software: 5-10000 loc/y
- Individual differences up to factor 5
  - Has not changed in the last 30 years
- Differences by programming language and reuse mechanisms

History of the Term "Software-Engineering"

Softwaretechnologie (Software-Engineering)
Softwareingenieurswesen
Softwaretechnik: Einzeltechnik aus der Lehre der Softwaretechnologie

software engineering: Die Entdeckung und Anwendung solider Ingenieur-Prinzipien mit dem Ziel, auf wirtschaftliche Weise Software zu bekommen, die zuverlässig ist und auf realen Rechnern läuft.

(F.L. Bauer, NATO-Konferenz Software-Engineering 1968)

A Little History

- NATO Conference on Software Engineering in Garmisch-Patenkirchen. Oct 7-10, 1968
- Hence the conference was called "on Software Engineering“
  [in Thayer&McGettrick IEEE Press]
- → "Software Crisis“

- "Component Software"

"The whole trouble comes from the fact that there is so much tinkering with software. It is not made in a clean fabrication process. What we need is Software Engineering." Friedrich L. Bauer, 1968

http://homepages.cs.ncl.ac.uk/brian.randell/NATO/N1968/index.html

Photos of Famous People at ICSE I

E. Dijkstra
"Strukturierte Programmierung"

K. Samuelson
"Stack"

D. McIlroy
"Mass-produced Software Components"

W. van der Poel
"ZEBRA"

B. Randell
"Fehlertoleranz"

D. Gries
"The Science of Programming"

T. Hoare
"Hoare Kalkül CSP"

G. Goos (3rd on the right)
"Compiler Construction"

Photos of Famous People at ICSE I

A. Perlis
"Algol",
"Computer Science"

C. Strachey
"denotational semantics"

N. Wirth
"Pascal", "EBNF"

P. Brinch Hansen
"Operating System Principles"

http://homepages.cs.ncl.ac.uk/brian.randell/NATO/N1968/index.html
http://homepages.cs.ncl.ac.uk/brian.randell/NATO/N1969/index.html

Different Forms of Software – A Classification

- Artefact**:** (lat. artificially made) Code or text or graphics that is made for software (documentation, specification, code, models, etc.)
- Program: Sources with object files, libraries
- Model: Partial program, abstracting from many details, typically used during development
- Software: Program with user and developer documentation, requirement specification, design descriptions, implementation description, **well-elaborated test suite**
- Product: Mature software. Good, simple, and pedagogic documentation. Simple Installation. Support guaranteed
  - Companies like products
- Product line **(product family)**: A group of products, having a common framework and product-specific extensions.
  - Note: every product is sold independently
- Framework**:** A software skeleton for many or all products in a product line
- **Software Ecosystem:** A software platform on which many third parties (plugin providers) can create their own products

Your Career

- First, you will be a designer and programmer in a team
  - You will need design skills most urgently for your own and small-size projects
  - In the software process, design flaws are most costly
- Afterwards, you will be project leader
  - Without good knowledge in design, you will not be a good developer nor project leader
- And then manager
  - But neither a good manager
  - Basic Microsoft strategy: every manager must be able to program

Your Career (II)

- Some become entrepreneurs
- What is an entrepreneur?
  - [Prof. R. Würth: Lecture notes on entrepreneurship http://www.iep.uni-karlsruhe.de/seite_260.php]

> Ein Unternehmer ist ein Problemlöser.
> Insbesondere sind ein Unternehmer und ein Kapitalist zweierlei. Der Kapitalist sieht den Gewinn im Mittelpunkt, aber der Unternehmer findet seine Befriedigung nur im Lösen von Problemen seiner Kunden und seiner Mitarbeiter. Damit kann er zwar auch Geld verdienen, im Wesentlichen lebt er aber nur einen grundlegenden Zug des Menschen aus: für Probleme befriedigende Lösungen zu finden.

ORGANISATION

Vorlesungen und Übungen

Vorlesung: Mi 11:10 E023

Wichtigste Informationsquelle:

- http://st.inf.tu-dresden.de/teaching/swt2
- http://st.inf.tu-dresden.de/ -> Teaching -> Softwaretechnologie II
- News auf dieser Seite regelmäßig lesen!

Übungsleiter: Martin Morgenstern

- Übungen können nur einen kleinen Teil der Vorlesung abdecken
- Ab Woche 2
- Semester ist in Komplexe aufgeteilt:
  - Petrinetze
  - Ontologien
  - Anforderungsanalyse: ZOPP, Lasten- und Pflichtenheft
  - Graph-/Logikprogrammierung
  - Regressiontesten

Übungsgruppen

Teilung der Übungsgruppen in kleine Gruppen á 4-5 Personen

Zumeist 2-3 Wochen Zeit zur Bearbeitung eines Komplexes

Alle Übungskomplexe sollten bearbeitet werden
→ Prüfung auch als 2/0/0 möglich

Achtung, obwohl die Vorlesung in Deutsch stattfindet, sind einige englische Folien eingestreut.
— Lehre auf dem Master-Niveau findet oft in Englisch statt
— Gewöhnen Sie sich bitte ein

Warning: Remarks on the Nature of the Course

- A University is unlike a high school
  - You should not expect to get a single book
- Software Engineering is too broad for that, unfortunately
- The lectures have to focus on the most important things
  - You should not expect to be an expert just by attending the course
- Find your way from the lecture slides into the books
  - Follow the reading instructions
  - Learn the additional material and read the additional readings
  - Follow the exercises in the groups
- Expect to learn min. 3-4 weeks for the oral exam
  - Don't wait until 1 week before the exam! That's too late...
- **Be aware: you have not yet seen larger systems**
  - Middle-size systems start over 100KLOC

Remarks on the Nature of the Course

- The purpose of lecturing is
  - To give you a condensed insight on the most important topics, such that you do not waste too much time during reading
  - To give you pointers for future work, once you left the course
- If you haven't got the pointer, you can waste years in darkness

Exams

- There will be two weeks for oral exams
- Somewhen in Feburary/March 2020
- The concrete weeks will be announced early January

- To register for your exam, write an email to st-exam@mailbox.tu-dresden.de
  - Tell the module you want to be examined (e.g., INF-B-530 or INF-BAS3)

LITERATURE

Recommended Literature: Overview Books

- We recommend one of (reading instructions can be followed in one of them):
  - Helmut Balzert, Lehrbuch der Softwaretechnik, 2. Auflage. Heidelberg, 2000, ISBN 3-8274-0042-2 (deutsch)
  - Bernd Brügge, Allen H. Dutoit, Objektorientierte Softwaretechnik, Pearson Studium
  - L. A. Maciaszek, B. L. Liong. Practical Software Engineering. A Case Study Approach. Addision-Wesley. Modern book on SE, UML in action in several case studies.
- Other good books, priority from top to bottom:
  - Ghezzi, Jazayeri, Mandrioli. Fundamentals of Software Engineering. Prentice Hall. Nice fundamented book. No fuzz, concrete.
  - S. Pfleeger: Software Engineering – Theory and Practice. Prentice-Hall. Good book, not too deep, but broad.
  - Van Vliet: Software Engineering. Wiley.
  - R. Pressman. Software Engineering – A Practitioner's Approach. McGrawHill

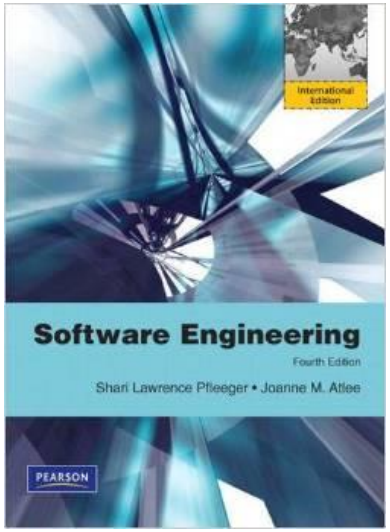
Some Chapters Have Obligatory Readings

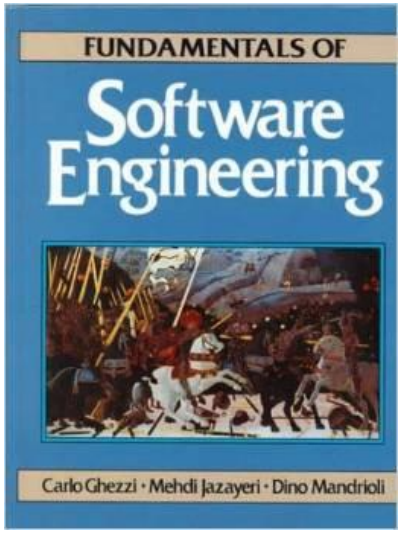
S. L. Pfleeger and J. Atlee:
Software Engineering: Theory and Practice.
Pearson. 2009.

— Chapter 5 (Designing the Architecture)

C. Ghezzi, M. Jazayeri and D. Mandrioli:
Fundamentals of Software Engineering.
Prentice Hall. 1992.

— Chapter 4 (Design and Software Architecture)

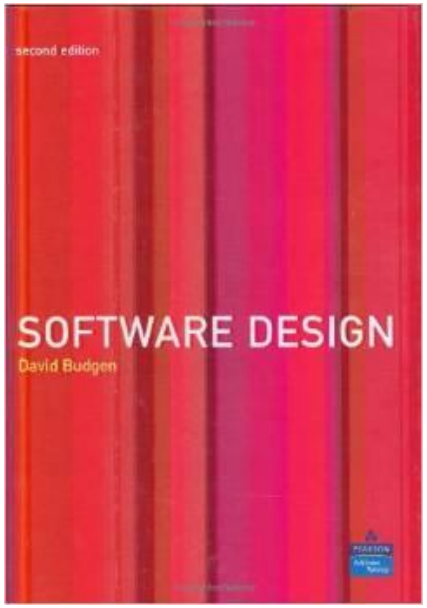


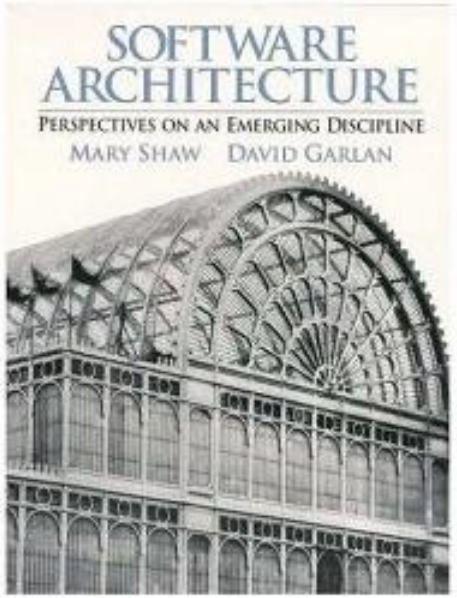


Secondary Reading

D. Budgen:
Software Design (2nd Edition).
Addison-Wesley. 2003.

M. Shaw and D. Garlan:
Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.





Recommended Books on UML – Unified Modeling Language

- UML is required. It is expected that you learn UML yourself from a good book.
- We recommend one of:
  - Online documentation on www.omg.org/uml
  - H. Störrle. UML für Studenten. Addison-Wesley (cheap, good!).
  - Leszek A. Maciaszek. Requirements Analysis and System Design – Developing Information Systems with UML. Addison-Wesley. Excellent concept book.
  - Object Management Group (OMG). UML - Unified Modeling Language. 2.0.

- Other excellent books:
  - Ken Lunn. Software development with UML. Palgrave-Macmillan. Many case realistic studies.

Reference Books

- R. Thayer, A. McGettrick. Software Engineering: A European Perspective. IEEE Press. Good collection of papers.
- M. Dorfman, R. Thayer. Software Engineering. IEEE Press. Good collection of papers.
- John McDermid. Software engineer's reference book. Butterworth-Heinemann. ISBN 0-7506-0813-7.
- A. Endres, D. Rombach. A Handbook of software and systems engineering - Empirical observations, laws and theories. Addison-Wesley. Very good collection of software laws. Nice!

Analysis and Design

- E. Gamma et al., Design Patterns, Addison-Wesley, ISBN 0-201-63361-2.
  - This standard reference book belongs to the bookshelf of every software engineer!
- Others
  - Rumbaugh et al. Object-oriented modelling and design. Prentice-Hall.
  - Booch. Object-oriented Analysis and Design. Addison-Wesley.
  - In German: Heide Balzert. Objektorientierten Systemanalyse. Spektrum der Wissenschaft.
  - Prieto-Diaz/Arango, Domain Analysis and Software Systems Modelling, IEEE Computer Society Press tutorial, ISBN 0-8186-8996-X, 1991

Component-Based Design

- C. Szyperski: Component Software. Addison-Wesley
- K. Czarnecki, U. Eisenecker: Generative Programming. Addison-Wesley
- U. Aßmann. Invasive Software Composition. Springer.

Project Management

- B. W. Boehm, Software Risk Management, 1989
- F. Brooks, The Mythical Man-Month, Addison-Wesley, 1975
- G. Weinberg, The Psychology of Computer Programming, Computer Science Series, 1971.
- E. Yourdan: The Death March.
- P. Neumann: Computer Risks, Addison-Wesley 1995.
- David Thielen. The 12 simple secrets of Microsoft McGraw-Hill.
- Dana Sobel. Longitude. About John Harrison. Just a good book about an excellent engineer.
- Simon Singh. Fermat's last theorem. Just an excellent book about an excellent mathematician (Wiles) thinking excellently hard.

Implementation

- J.L. Bentley, Programming Pearls, Addison-Wesley, 2. Auflage 1989, ISBN 0-201-10331-1
- J.L. Bentley, More Programming Pearls, Addison-Wesley, 1988, ISBN 0-201-11889-0
- J.L. Bentley, Writing Efficient Programs, Prentice-Hall, ISBN 0-13-970244-X, 1982

Testing and Quality

- Uwe Viggenschow. Objektorientiertes Testen und Testautomatisierung in der Praxis. Konzepte, Techniken und Verfahren. Dpunkt-Verlag, Heidelberg. www.oo-testen.de. Nice practical book on testing.
- P. Liggesmeyer. Software-Qualitätsmanagement. Verlag Spektrum der Wissenschaften, Heidelberg.
- Boris Beizer: System Testing and Quality Assurance,Van Nostrand Reinhold, New York, 1984, ISBN 0-442-21306-9
- Glenford J. Myers, The Art of Software Testing, 1979
- Nesi (ed.), Objective Software Quality, 1995, Springer LNCS 926, ISBN 3-540-59449-3
- N. Fenton, S.L. Pfleeger. Software Metrics – a rigorous and practical approach. PWS Publishing.