WIKIPEDIA

Software development

Software development is the process of conceiving, specifying, designing, programming, documenting, testing, and bug fixing involved in creating and maintaining applications, frameworks, or other software components. Software development is a process of writing and maintaining the source code, but in a broader sense, it includes all that is involved between the conception of the desired software through to the final manifestation of the software, sometimes in a planned and structured process.[1] Therefore, software development may include research, new development, prototyping, modification, reuse, re-engineering, maintenance, or any other activities that result in software products.[2]

Software can be developed for a variety of purposes, the three most common being to meet specific needs of a specific client/business (the case with custom software), to meet a perceived need of some set of potential users (the case with commercial and open source software), or for personal use (e.g. a scientist may write software to automate a mundane task). **Embedded software development**, that is, the development of embedded software, such as used for controlling consumer products, requires the development process to be integrated with the development of the controlled physical product. System software underlies applications and the programming process itself, and is often developed separately.

The need for better quality control of the software development process has given rise to the discipline of software engineering, which aims to apply the systematic approach exemplified in the engineering paradigm to the process of software development.

There are many approaches to software project management, known as software development life cycle models, methodologies, processes, or models. The waterfall model is a traditional version, contrasted with the more recent innovation of agile software development.

Contents

Methodologies

Software development activities
- Identification of need
- Planning
- Designing
- Implementation, testing and documenting
- Deployment and maintenance

Subtopics
- View model
- Business process and data modelling
- Computer-aided software engineering
- Integrated development environment
- Modeling language
- Programming paradigm
- Reuse of solutions

See also
- Roles and industry
- Specific applications

References

Further reading

Methodologies

A software development process (also known as a software development methodology, model, or life cycle) is a framework that is used to structure, plan, and control the process of developing information systems. A wide variety of such frameworks has evolved over the years, each with its own recognized strengths and weaknesses. There are several different approaches to software development: some take a more structured, engineering-based approach to develop business solutions, whereas others may take a more incremental approach, where software evolves as it is developed piece-by-piece. One system development methodology is not necessarily suitable for use by all projects. Each of the available methodologies is best suited to specific kinds of projects, based on various technical, organizational, project and team considerations. [3]

Most methodologies share some combination of the following stages of software development:

- Analyzing the problem
- Market research
- Gathering requirements for the proposed business solution
- Devising a plan or design for the software-based solution
- Implementation (coding) of the software
- Testing the software
- Deployment
- Maintenance and bug fixing

These stages are often referred to collectively as the software development life-cycle, or SDLC. Different approaches to software development may carry out these stages in different orders, or devote more or less time to different stages. The level of detail of the documentation produced at each stage of software development may also vary. These stages may also be carried out in turn (a “waterfall” based approach), or they may be repeated over various cycles or iterations (a more "extreme" approach). The more extreme approach usually involves less time spent on planning and documentation, and more time spent on coding and development of automated tests. More “extreme” approaches also promote continuous testing throughout the development life-cycle, as well as having a working (or bug-free) product at all times. More structured or “waterfall” based approaches attempt to assess the majority of risks and develop a detailed plan for the software before implementation (coding) begins, and avoid significant design changes and re-coding in later stages of the software development life-cycle planning.

There are significant advantages and disadvantages to the various methodologies, and the best approach to solving a problem using software will often depend on the type of problem. If the problem is well understood and a solution can be effectively planned out ahead of time, the more "waterfall" based approach may work the best. If, on the other hand, the problem is unique (at least to the development team) and the structure of the software solution cannot be easily envisioned, then a more "extreme" incremental approach may work best.

Software development activities

Identification of need

The sources of ideas for software products are plentiful. These ideas can come from market research including the demographics of potential new customers, existing customers, sales prospects who rejected the product, other internal software development staff, or a creative third party. Ideas for software products are usually first evaluated by marketing personnel for economic feasibility, for fit with existing channels distribution, for possible effects on existing product lines, required features, and for fit with the company's marketing objectives. In a marketing evaluation phase, the cost and time assumptions become evaluated. A decision is reached early in the first phase as to whether, based on the more detailed information generated by the marketing and development staff, the project should be pursued further.[4]

In the book *"Great Software Debates"*, Alan M. Davis states in the chapter *"Requirements"*, sub-chapter *"The Missing Piece of Software Development"*

> Students of engineering learn engineering and are rarely exposed to finance or marketing. Students of marketing learn marketing and are rarely exposed to finance or engineering. Most of us become specialists in just one area. To complicate matters, few of us meet interdisciplinary people in the workforce, so there are few roles to mimic. Yet, software product planning is critical to the development success and absolutely requires knowledge of multiple disciplines.[5]

Because software development may involve compromising or going beyond what is required by the client, a software development project may stray into less technical concerns such as human resources, risk management, intellectual property, budgeting, crisis management, etc. These processes may also cause the role of business development to overlap with software development.

Planning

Planning is an objective of each and every activity, where we want to discover things that belong to the project. An important task in creating a software program is extracting the requirements or requirements analysis.[6] Customers typically have an abstract idea of what they want as an end result but do not know what *software* should do. Skilled and experienced software engineers recognize incomplete, ambiguous, or even contradictory requirements at this point. Frequently demonstrating live code may help reduce the risk that the requirements are incorrect.

"Although much effort is put in the requirements phase to ensure that requirements are complete and consistent, rarely that is the case; leaving the software design phase as the most influential one when it comes to minimizing the effects of new or changing requirements. Requirements volatility is challenging because they impact future or already going development efforts."[7]

Once the general requirements are gathered from the client, an analysis of the scope of the development should be determined and clearly stated. This is often called a scope document.

Designing

Once the requirements are established, the design of the software can be established in a software design document. This involves a preliminary or high-level design of the main modules with an overall picture (such as a block diagram) of how the parts fit together. The language, operating system, and hardware components should all be known at this time. Then a detailed or low-level design is created, perhaps with prototyping as proof-of-concept or to firm up requirements.

Implementation, testing and documenting

Implementation is the part of the process where software engineers actually program the code for the project.

Software testing is an integral and important phase of the software development process. This part of the process ensures that defects are recognized as soon as possible. In some processes, generally known as test-driven development, tests may be developed just before implementation and serve as a guide for the implementation's correctness.

Documenting the internal design of software for the purpose of future maintenance and enhancement is done throughout development. This may also include the writing of an API, be it external or internal. The software engineering process chosen by the developing team will determine how much internal documentation (if any) is necessary. Plan-driven models (e.g., Waterfall) generally produce more documentation than Agile models.

Deployment and maintenance

Deployment starts directly after the code is appropriately tested, approved for release, and sold or otherwise distributed into a production environment. This may involve installation, customization (such as by setting parameters to the customer's values), testing, and possibly an extended period of evaluation.

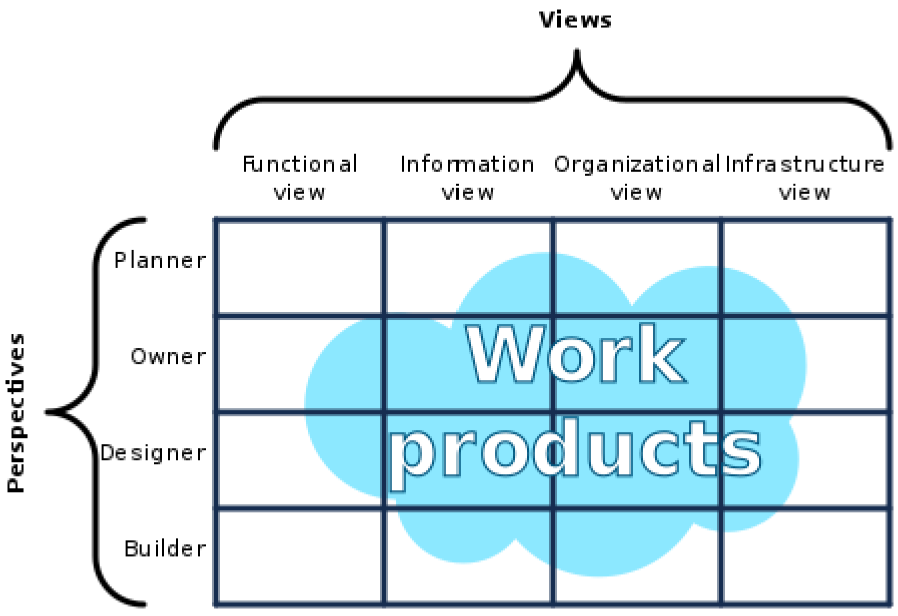
Software training and support is important, as software is only effective if it is used correctly.

Maintaining and enhancing software to cope with newly discovered faults or requirements can take substantial time and effort, as missed requirements may force redesign of the software.

Subtopics

View model

A view model is a framework that provides the viewpoints on the system and its environment, to be used in the software development process. It is a graphical representation of the underlying semantics of a view.



The TEAF Matrix of Views and Perspectives.

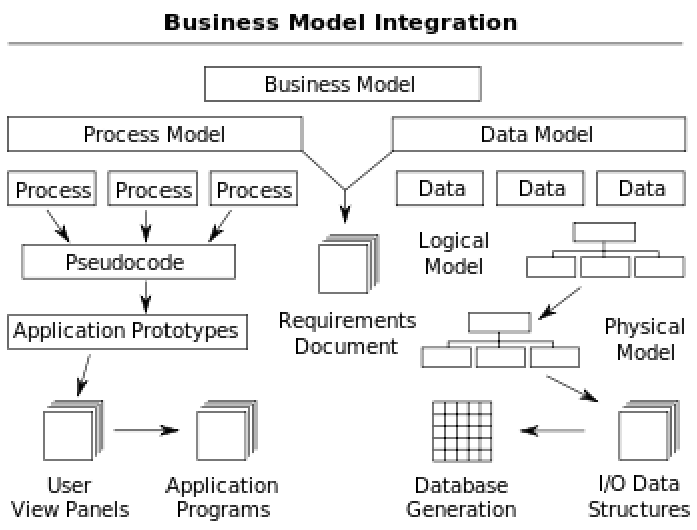
The purpose of viewpoints and views is to enable human engineers to comprehend very complex systems and to organize the elements of the problem and the solution around domains of expertise. In the engineering of physically intensive systems, viewpoints often correspond to capabilities and responsibilities within the engineering organization.[8]

Most complex system specifications are so extensive that no one individual can fully comprehend all aspects of the specifications. Furthermore, we all have different interests in a given system and different reasons for examining the system's specifications. A business executive will ask different questions of a system make-up than would a system implementer. The concept of viewpoints framework, therefore, is to provide separate viewpoints into the specification of a given complex system. These viewpoints each satisfy an audience with interest in some set of aspects of the system. Associated with each viewpoint is a viewpoint language that optimizes the vocabulary and presentation for the audience of that viewpoint.

Business process and data modelling

Graphical representation of the current state of information provides a very effective means for presenting information to both users and system developers.

- A business model illustrates the functions associated with the business process being modeled and the organizations that perform these functions. By depicting activities and information flows, a foundation is created to visualize, define, understand, and validate the nature of a process.
- A data model provides the details of information to be stored and is of primary use when the final product is the generation of computer software code for an application or the preparation of a functional specification to aid a computer software make-or-buy decision. See the figure on the right for an example of the interaction between business process and data models.[9]



example of the interaction between business process and data models.[9]

Usually, a model is created after conducting an interview, referred to as business analysis. The interview consists of a facilitator asking a series of questions designed to extract required information that describes a process. The interviewer is called a facilitator to emphasize that it is the participants who provide the information. The facilitator should have some knowledge of the process of interest, but this is not as important as having a structured methodology by which the questions are asked of the process expert. The methodology is important because usually a team of facilitators is collecting information across the facility and the results of the information from all the interviewers must fit together once completed.[9]

The models are developed as defining either the current state of the process, in which case the final product is called the "as-is" snapshot model, or a collection of ideas of what the process should contain, resulting in a "what-can-be" model. Generation of process and data models can be used to determine if the existing processes and information systems are sound and only need minor modifications or enhancements, or if re-engineering is required as a corrective action. The creation of business models is more than a way to view or automate your information process. Analysis can be used to fundamentally reshape the way your business or organization conducts its operations.[9]

Computer-aided software engineering

Computer-aided software engineering (CASE), in the field software engineering, is the scientific application of a set of software tools and methods to the development of software which results in high-quality, defect-free, and maintainable software products.[10] It also refers to methods for the development of information systems together with automated tools that can be used in the software development process.[11] The term "computer-aided software engineering" (CASE) can refer to the software used for the automated development of systems software, i.e., computer code. The CASE functions include analysis, design, and programming. CASE tools automate methods for designing, documenting, and producing structured computer code in the desired programming language.[12]

Two key ideas of Computer-aided Software System Engineering (CASE) are:[13]

- Foster computer assistance in software development and software maintenance processes, and
- An engineering approach to software development and maintenance.

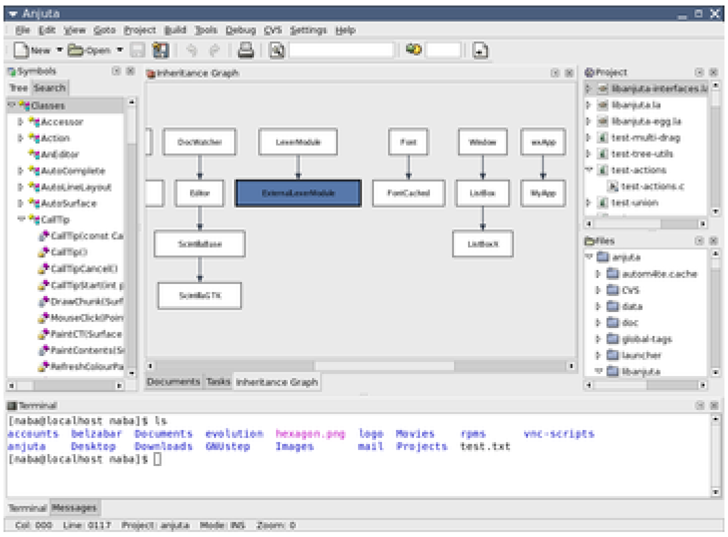
Typical CASE tools exist for configuration management, data modeling, model transformation, refactoring, source code generation.

Integrated development environment

An integrated development environment (IDE) also known as *integrated design environment* or *integrated debugging environment* is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a:

- Source code editor,
- Compiler or interpreter,
- Build automation tools, and
- Debugger (usually).

IDEs are designed to maximize programmer productivity by providing tight-knit components with similar user interfaces. Typically an IDE is dedicated to a specific programming language, so as to provide a feature set which most closely matches the programming paradigms of the language.


```
[naba@localhost naba]$ ls
accounts  belzabar  Documents  evolution  hexagon.png  logo  Movies    rpms      vnc-scripts
anjuta    Desktop   Downloads  GNUstep    Images       mail  Projects  test.txt
[naba@localhost naba]$ 
```
Anjuta
File Edit View Goto Project Build Tools Debug CVS Settings Help
Symbols
Inheritance Graph
Project
Files
Terminal
Messages
Col: 000 Line: 0117 Project: anjuta Mode: INS Zoom: 0
Anjuta, a C and C++ IDE for the GNOME environment

Modeling language

A modeling language is any artificial language that can be used to express information or knowledge or systems in a structure that is defined by a consistent set of rules. The rules are used for interpretation of the meaning of components in the structure. A modeling language can be graphical or textual.[14] Graphical modeling languages use a diagram techniques with named symbols that represent concepts and lines that connect the symbols and that represent relationships and various other graphical annotation to represent constraints. Textual modeling languages typically use standardised keywords accompanied by parameters to make computer-interpretable expressions.

Examples of graphical modelling languages in the field of software engineering are:

- Business Process Modeling Notation (BPMN, and the XML form BPML) is an example of a process modeling language.
- EXPRESS and EXPRESS-G (ISO 10303-11) is an international standard general-purpose data modeling language.
- Extended Enterprise Modeling Language (EEML) is commonly used for business process modeling across layers.
- Flowchart is a schematic representation of an algorithm or a stepwise process,
- Fundamental Modeling Concepts (FMC) modeling language for software-intensive systems.
- IDEF is a family of modeling languages, the most notable of which include IDEF0 for functional modeling, IDEF1X for information modeling, and IDEF5 for modeling ontologies.

- LePUS3 is an object-oriented visual Design Description Language and a formal specification language that is suitable primarily for modelling large object-oriented (Java, C++, C#) programs and design patterns.
- Specification and Description Language(SDL) is a specification language targeted at the unambiguous specification and description of the behaviour of reactive and distributed systems.
- Unified Modeling Language (UML) is a general-purpose modeling language that is an industry standard for specifying software-intensive systems. UML 2.0, the current version, supports thirteen different diagram techniques and has widespread tool support.

Not all modeling languages are executable, and for those that are, using them doesn't necessarily mean that programmers are no longer needed. On the contrary, executable modeling languages are intended to amplify the productivity of skilled programmers, so that they can address more difficult problems, such as parallel computing and distributed systems.

Programming paradigm

A programming paradigm is a fundamental style of computer programming, which is not generally dictated by the project management methodology (such as waterfall or agile). Paradigms differ in the concepts and abstractions used to represent the elements of a program (such as objects, functions, variables, constraints) and the steps that comprise a computation (such as assignations, evaluation, continuations, data flows). Sometimes the concepts asserted by the paradigm are utilized cooperatively in high-level system architecture design; in other cases, the programming paradigm's scope is limited to the internal structure of a particular program or module.

A programming language can support multiple paradigms. For example, programs written in C++ or Object Pascal can be purely procedural, or purely object-oriented, or contain elements of both paradigms. Software designers and programmers decide how to use those paradigm elements. In object-oriented programming, programmers can think of a program as a collection of interacting objects, while in functional programming a program can be thought of as a sequence of stateless function evaluations. When programming computers or systems with many processors, process-oriented programming allows programmers to think about applications as sets of concurrent processes acting upon logically shared data structures.

Just as different groups in software engineering advocate different *methodologies*, different programming languages advocate different *programming paradigms*. Some languages are designed to support one paradigm (Smalltalk supports object-oriented programming, Haskell supports functional programming), while other programming languages support multiple paradigms (such as Object Pascal, C++, C#, Visual Basic, Common Lisp, Scheme, Python, Ruby, and Oz).

Many programming paradigms are as well known for what methods they *forbid* as for what they enable. For instance, pure functional programming forbids using side-effects; structured programming forbids using goto statements. Partly for this reason, new paradigms are often regarded as doctrinaire or overly rigid by those accustomed to earlier styles. Avoiding certain methods can make it easier to prove theorems about a program's correctness, or simply to understand its behavior.

Examples of high-level paradigms include:

- Aspect-oriented software development
- Domain-specific modeling
- Model-driven engineering
- Object-oriented programming methodologies
  - Grady Booch's object-oriented design (OOD), also known as object-oriented analysis and design (OOAD). The Booch model includes six diagrams: class, object, state transition, interaction, module, and process.[15]
- Search-based software engineering
- Service-oriented modeling
- Structured programming
- Top-down and bottom-up design
  - Top-down programming: evolved in the 1970s by IBM researcher Harlan Mills (and Niklaus Wirth) in developed structured programming.

Reuse of solutions

- A software framework is a re-usable design or implementation for a software system or subsystem.
- Existing components (Component-based software engineering) can be reused, assembled together to create a larger application.

- API (Application programming interface, Web service) establish a set of "subroutine definitions, protocols, and tools for building application software" which can be utilized in future builds.
- Open Source documentations, via libraries such as GitHub, provide free code for software developers to re-use and implement into new applications or designs.

See also

- Continuous integration
- Custom software
- DevOps
- Functional specification
- Programming productivity
- Software blueprint
- Software design
- Software development effort estimation
- Software development process
- Software project management
- Specification and Description Language
- User experience
- Software industry

Roles and industry

- Bachelor of Science in Information Technology
- Computer programmer
- Consulting software engineer
- Offshore software development
- Software developer
- Software engineer
- Software publisher

Specific applications

- Video game development
- Web application development
- Web engineering
- Mobile Application Development

References

1. "Application Development (AppDev) Defined and Explained" (http://www.bestpricecomputers.co.uk/glossary/application-development.htm). Bestpricecomputers.co.uk. 2007-08-13. Retrieved 2012-08-05.
2. DRM Associates (2002). "New Product Development Glossary" (http://www.npd-solutions.com/glossary.html). Retrieved 2006-10-29.
3. System Development Methodologies for Web-Enabled E-Business: A Customization Framework Linda V. Knight (DePaul University, USA), Theresa A. Steinbach (DePaul University, USA) and Vince Kellen (Blue Wolf, USA)
4. Joseph M. Morris (2001). *Software Industry Accounting*. p.1.10
5. Alan M. Davis. Great Software Debates (October 8, 2004), pp:125-128 Wiley-IEEE Computer Society Press
6. Ralph, P., and Wand, Y. A Proposal for a Formal Definition of the Design Concept (http://www.kathrynpieplow.pwrfaculty.org/wp-content/uploads/2010/01/Ralph-Wand-definition-design.pdf). In, Lyytinen, K., Loucopoulos, P., Mylopoulos, J., and Robinson, W., (eds.), Design Requirements Engineering: A Ten-Year Perspective: Springer-Verlag, 2009, pp. 103-136
7. Otero, Carlos. "Software Design Challenges" (http://www.ittoday.info/ITPerformanceImprovement/Articles/2012-06Otero.html). *IT Performance Improvement*. Taylor & Francis LLC. Retrieved 19 October 2017.
8. Edward J. Barkmeyer ea (2003). *Concepts for Automating Systems Integration* (http://www.mel.nist.gov/msidlibrary/doc/AMIS-Concepts.pdf) NIST 2003.

9. Paul R. Smith & Richard Sarfaty (1993). Creating a strategic plan for configuration management using Computer Aided Software Engineering (CASE) tools. (http://www.osti.gov/energycitations/servlets/purl/10160331-YhIRrY/) Paper For 1993 National DOE/Contractors and Facilities CAD/CAE User's Group.
10. Kuhn, D.L (1989). "Selecting and effectively using a computer-aided software engineering tool". Annual Westinghouse computer symposium; 6-7 Nov 1989; Pittsburgh, PA (USA); DOE Project.
11. P. Loucopoulos and V. Karakostas (1995). *System Requirements Engineering*. McGraw-Hill.
12. CASE (http://www.its.bldrdoc.gov/projects/devglossary/_case.html) Archived (https://web.archive.org/web/20120218104409/http://www.its.bldrdoc.gov/projects/devglossary/_case.html) 2012-02-18 at the Wayback Machine definition In: *Telecom Glossary 2000 (http://www.its.bldrdoc.gov/projects/devglossary/) Archived (https://web.archive.org/web/20051122063850/http://www.its.bldrdoc.gov/projects/devglossary/) 2005-11-22 at the Wayback Machine. Retrieved 26 Oct 2008.*
13. K. Robinson (1992). *Putting the Software Engineering into CASE*. New York : John Wiley and Sons Inc.
14. Xiao He (2007). "A metamodel for the notation of graphical modeling languages (ftp://antvision.cn/pub/%E5%8C%97%E4%BA%AC%E7%A7%91%E6%8A%80%E5%A4%A7%E5%AD%A6/%E8%AE%BA%E6%96%87/%E4%BD%95%E5%95%B8%E8%80%81%E5%B8%88%E8%AE%BA%E6%96%87/A%20metamodel%20for%20the%20notation%20of%20graphical%20modeling%20languages.pdf)". In: *Computer Software and Applications Conference, 2007. COMPSAC 2007 – Vol. 1. 31st Annual International*, Volume 1, Issue, 24–27 July 2007, pp 219-224.
15. Merx, Georges G.; Norman, Ronald J. (2006). *Unified Software Engineering with Java*. Prentice-Hall, Inc. p. 201. ISBN 0130473766.

Further reading

- Kit, Edward (1992). *Software Testing in The Real World*. Addison-Wesley Professional. ISBN 0201877562.
- McCarthy, Jim (1995). *Dynamics of Software Development*. Microsoft Press. ISBN 1556158238.
- Conde, Dan (2002). *Software Product Management: Managing Software Development from Idea to Product to Marketing to Sales*. Aspatore Books. ISBN 1587622025.
- Davis, A. M. (2005). *Just enough requirements management: Where software development meets marketing*. Dorset House Publishing Company, Incorporated. ISBN 0932633641.
- Hasted, Edward (2005). *Software That Sells: A Practical Guide to Developing and Marketing Your Software Project*. Wiley Publishing. ISBN 0764597833.
- Hohmann, Luke (2003). *Beyond Software Architecture: Creating and Sustaining Winning Solutions*. Addison-Wesley Professional. ISBN 0201775948.
- John W. Horch (2005). "Two Orientations On How To Work With Objects." In: *IEEE Software*. vol. 12, no. 2, pp. 117–118, Mar., 1995.
- Rittinghouse, John (2003). *Managing Software Deliverables: A Software Development Management Methodology*. Digital Press. ISBN 155558313X.
- Wiegers, Karl E. (2005). *More About Software Requirements: Thorny Issues and Practical Advice*. Microsoft Press. ISBN 0735622671.
- Wysocki, Robert K. (2006). *Effective Software Project Management*. Wiley. ISBN 0764596365.