Guide to the Software Engineering Body of Knowledge

2004 Version

SWEBOK®

A project of the IEEE Computer Society Professional Practices Committee

Guide to the Software Engineering Body of Knowledge

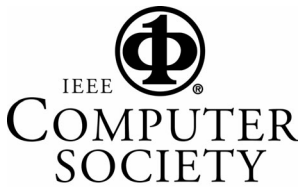
2004 Version

SWEBOK®

Executive Editors
Alain Abran, École de technologie supérieure
James W. Moore, The MITRE Corp.

Editors
Pierre Bourque, École de technologie supérieure
Robert Dupuis, Université du Québec à Montréal

Project Champion
Leonard L. Tripp, Chair, Professional Practices Committee,
IEEE Computer Society (2001-2003)



http://computer.org

Los Alamitos, California

Washington • Brussels • Tokyo

IEEE Computer Society Order Number C2330
ISBN 0-7695-2330-7
Library of Congress Number 2005921729

Additional copies may be ordered from:

IEEE Computer Society
Customer Service Center
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1314
Tel: + 1-714-821-8380
Fax: + 1-714-821-4641
E-mail: cs.books@computer.org

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
Tel: + 1-732-981-0060
Fax: + 1-732-981-9667
http://shop.ieee.org/store/
customer-service@ieee.org

IEEE Computer Society
Asia/Pacific Office
Watanabe Bldg., 1-4-2
Minami-Aoyama
Minato-ku, Tokyo 107-0062
JAPAN
Tel: + 81-3-3408-3118
Fax: + 81-3-3408-3553
tokyo.ofc@computer.org

Publisher: Angela Burgess
Group Managing Editor, CS Press: Deborah Plummer
Advertising/Promotions: Tom Fink
Production Editor: Bob Werner
Printed in the United States of America

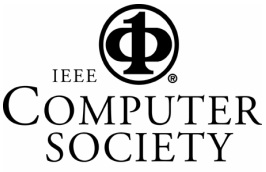


Table of Contents

FOREWORD

In this Guide, the IEEE Computer Society establishes for the first time a baseline for the body of knowledge for the field of software engineering, and the work partially fulfills the Society's responsibility to promote the advancement of both theory and practice in this field. In so doing, the Society has been guided by the experience of disciplines with longer histories but was not bound either by their problems or their solutions.

It should be noted that the Guide does not purport to define the body of knowledge but rather to serve as a compendium and guide to the body of knowledge that has been developing and evolving over the past four decades. Furthermore, this body of knowledge is not static. The *Guide* must, necessarily, develop and evolve as software engineering matures. It nevertheless constitutes a valuable element of the software engineering infrastructure.

In 1958, John Tukey, the world-renowned statistician, coined the term *software*. The term *software engineering* was used in the title of a NATO conference held in Germany in 1968. The IEEE Computer Society first published its *Transactions on Software Engineering* in 1972. The committee established within the IEEE Computer Society for developing software engineering standards was founded in 1976.

The first holistic view of software engineering to emerge from the IEEE Computer Society resulted from an effort led by Fletcher Buckley to develop IEEE standard 730 for software quality assurance, which was completed in 1979. The purpose of IEEE Std 730 was to provide uniform, minimum acceptable requirements for preparation and content of software quality assurance plans. This standard was influential in completing the developing standards in the following topics: configuration management, software testing, software requirements, software design, and software verification and validation.

During the period 1981-1985, the IEEE Computer Society held a series of workshops concerning the application of software engineering standards. These workshops involved practitioners sharing their experiences with existing standards. The workshops also held sessions on planning for future standards, including one involving measures and metrics for software engineering products and processes. The planning also resulted in IEEE Std 1002, Taxonomy of Software Engineering Standards (1986), which provided a new, holistic view of software engineering. The standard describes the form and content of a software engineering standards taxonomy. It explains the various types of software engineering standards, their functional and external relationships, and the role of various functions participating in the software life cycle.

In 1990, planning for an international standard with an overall view was begun. The planning focused on reconciling the software process views from IEEE Std 1074 and the revised US DoD standard 2167A. The revision was eventually published as DoD Std 498. The international standard was completed in 1995 with designation, ISO/IEC12207, and given the title of Standard for Software Life Cycle Processes. Std ISO/IEC 12207 provided a major point of departure for the body of knowledge captured in this book.

It was the IEEE Computer Society Board of Governors' approval of the motion put forward in May 1993 by Fletcher Buckley which resulted in the writing of this book. The Association for Computing Machinery (ACM) Council approved a related motion in August 1993. The two motions led to a joint committee under the leadership of Mario Barbacci and Stuart Zweben who served as cochairs. The mission statement of the joint committee was "To establish the appropriate sets(s) of criteria and norms for professional practice of software engineering upon which industrial decisions, professional certification, and educational curricula can be based." The steering committee organized task forces in the following areas:

1. Define Required Body of Knowledge and Recommended Practices.
2. Define Ethics and Professional Standards.
3. Define Educational Curricula for undergraduate, graduate, and continuing education.

This book supplies the first component: required body of knowledge and recommend practices.

The code of ethics and professional practice for software engineering was completed in 1998 and approved by both the ACM Council and the IEEE Computer Society Board of Governors. It has been adopted by numerous corporations and other organizations and is included in several recent textbooks.

The educational curriculum for undergraduates is being completed by a joint effort of the IEEE Computer Society and the ACM and is expected to be completed in 2004.

Every profession is based on a body of knowledge and recommended practices, although they are not always defined in a precise manner. In many cases, these are formally documented, usually in a form that permits them to be used for such purposes as accreditation of academic programs, development of education and training programs, certification of specialists, or professional licensing. Generally, a professional society or related body maintains custody of such a formal definition. In cases where no such formality exists, the body of knowledge and recommended practices are "generally recognized" by practitioners and may be codified in a variety of ways for different uses.

It is hoped that readers will find this book useful in guiding them toward the knowledge and resources they need in their lifelong career development as software engineering professionals.

The book is dedicated to Fletcher Buckley in recognition of his commitment to promoting software engineering as a professional discipline and his excellence as a software engineering practitioner in radar applications.

Leonard L. Tripp, IEEE Fellow 2003

Chair, Professional Practices Committee, IEEE Computer Society (2001-2003)
Chair, Joint IEEE Computer Society and ACM Steering Committee
for the Establishment of Software Engineering as a Profession (1998-1999)
Chair, Software Engineering Standards Committee, IEEE Computer Society (1992-1998)

Associate Editors

The following persons served as Associate Editors for either the Trial version published in 2001 or for the 2004 version.

Software Requirements

Peter Sawyer and Gerald Kotonya, Computing Department, Lancaster University, UK, {p.sawyer}{g.kotonya}@lancaster.ac.uk

Software Design

Guy Tremblay, Département d'informatique, UQAM, Canada, tremblay.guy@uqam.ca

Software Construction

Steve McConnell, Construx Software, USA, Steve.McConnell@construx.com
Terry Bollinger, the MITRE Corporation, USA, terry@mitre.org
Philippe Gabrini, Département d'informatique, UQAM, Canada, gabrini.philippe@uqam.ca
Louis Martin, Département d'informatique, UQAM, Canada, martin.louis@uqam.ca

Software Testing

Antonia Bertolino and Eda Marchetti, ISTI-CNR, Italy, {antonia.bertolino}{eda.marchetti}@isti.cnr.it

Software Maintenance

Thomas M. Pigoski, Techsoft Inc., USA, tmpigoski@techsoft.com
Alain April, École de technologie supérieure, Canada, aapril@ele.etsmtl.ca

Software Configuration Management

John A. Scott, Lawrence Livermore National Laboratory, USA, scott7@llnl.gov
David Nisse, USA, nissed@worldnet.att.net

Software Engineering Management

Dennis Frailey, Raytheon Company, USA, DJFrailey@Raytheon.com
Stephen G. MacDonell, Auckland University of technology, New Zealand, smacdone@aut.ac.nz
Andrew R. Gray, University of Otago, New Zealand

Software Engineering Process

Khaled El Emam, served while at the Canadian National Research Council, Canada, khaled.el-emam@nrc-cnrc.gc.ca

Software Engineering Tools and Methods

David Carrington, School of Information Technology and Electrical Engineering, The University of Queensland, Australia, davec@itee.uq.edu.au

Software Quality

Alain April, École de technologie supérieure, Canada, aapril@ele.etsmtl.ca
Dolores Wallace, retired from the National Institute of Standards and Technology, USA, Dolores.Wallace@nist.gov
Larry Reeker, NIST, USA, Larry.Reeker@nist.gov

References Editor

Marc Bouisset, Département d'informatique, UQAM, Bouisset.Marc@uqam.ca

Industrial Advisory Board

At the time of the publication, the following people formed the Industrial Advisory Board:

Mario R. Barbacci, Software Engineering Institute, representing the
IEEE Computer Society

Carl Chang, representing Computing Curricula 2001

François Coallier, École de technologie supérieure, speaking as ISO/IEC JTC 1 / SC7 Chairman

Charles Howell, The MITRE Corporation

Anatol Kark, National Research Council of Canada

Philippe Kruchten, University of British Columbia, served as representative of Rational Software

Laure Le Bars, SAP Labs (Canada)

Steve McConnell, Construx Software

Dan Nash, Raytheon Company

Fred Otto, Canadian Council of Professional Engineers (CCPE)

Richard Metz, The Boeing Company

Larry Reeker, National Institute of Standards and Technology, Department of Commerce, USA

The following persons served along with the IAB in the Executive Change Control Board for the 2004 edition:

Donald Bagert, Rose-Hulman Institute of Technology, representing the IEEE Computer Society Professional Practices Committee

Ann Sobel, Miami University, representing the Computing Curricula Software Engineering Steering Committee

Panel of Experts

The following persons served on the panel of experts for the preparation of the Trial version of the Guide:

Steve McConnell, Construx Software

Roger Pressman, R.S. Pressman and Associates

Ian Sommerville, Lancaster University, UK

Review Team

The following people participated in the review process of this Guide for the Trial version and/or for the 2004 version.

Abbas, Rasha, Australia
Abran, Alain, Canada
Accioly, Carlos, Brazil
Ackerman, Frank, USA
Akiyama, Yoshihiro, Japan
Al-Abdullah, Mohammad, USA
Alarcon, Miren Idoia, Spain
Alawy, Ahmed, USA
Alleman, Glen, USA
Allen, Bob, Canada
Allen, David, USA
Amorosa, Francesco, Italy
Amyot, Daniel, Canada
Andrade, Daniel, Brazil
April, Alain, Canada
Arroyo-Figueror, Javier, USA
Ashford, Sonny, USA
Atsushi, Sawada, Japan
Backitis Jr., Frank, USA
Bagert, Donald, USA
Baker, Jr., David, USA
Baker, Theodore, USA
Baldwin, Mark, USA
Bales, David, UK
Bamberger, Judy, USA
Banerjee, Bakul, USA
Barber, Scott, USA
Barker, Harry, UK
Barnes, Julie, USA
Barney, David, Australia
Barros, Rafael, Colombia
Bastarache, Louis, Canada
Bayer, Steven, USA
Beaulac, Adeline, Canada
Beck, William, USA
Beckman, Kathleen, USA
Below, Doreen, USA
Benediktsson, Oddur, Iceland
Ben-Menachem, Mordechai, Israel
Bergeron, Alain, Canada
Berler, Alexander, Greece
Bernet, Martin, USA
Bernstein, Larry, USA
Bertram, Martin, Germany
Bialik, Tracy, USA
Bielikova, Maria, Slovakia
Bierwolf, Robert, The Netherlands
Bisbal, Jesus, Ireland
Boivin, Michel, Canada
Bolton, Michael, Canada
Bomitali, Evelino, Italy
Bonderer, Reto, Switzerland
Bonk, Francis, USA
Booch, Grady, USA
Booker, Glenn, USA
Börstler, Jürgen, Sweden
Borzovs, Juris, Latvia
Botting, Richard, USA
Bourque, Pierre, Canada
Bowen, Thomas, USA
Boyd, Milt, USA
Boyer, Ken, USA
Brashear, Phil, USA
Briggs, Steve, USA
Bright, Daniela, USA
Brosseau, Jim, Canada
Brotbeck, George, USA
Brown, Normand, Canada
Bruhn, Anna, USA
Brune, Kevin, USA
Bryant, Jeanne, USA
Buglione, Luigi, Italy
Bullock, James, USA
Burns, Robert, USA
Burnstein, Ilene, USA
Byrne, Edward, USA
Calizaya, Percy, Peru
Carreon, Juan, USA
Carroll, Sue, USA
Carruthers, Kate, Australia
Caruso, Richard, USA
Carvalho, Paul, Canada
Case, Pam, USA
Cavanaugh, John, USA
Celia, John A., USA
Chalupa Sampaio, Alberto Antonio, Portugal
Chan, F.T., Hong Kong
Chan, Keith, Hong Kong
Chandra, A.K., India
Chang, Wen-Kui, Taiwan
Chapin, Ned, USA
Charette, Robert, USA
Chevrier, Marielle, Canada
Chi, Donald, USA
Chiew, Vincent, Canada
Chilenski, John, USA
Chow, Keith, Italy
Ciciliani, Ricardo, Argentina
Clark, Glenda, USA
Cleavenger, Darrell, USA
Cloos, Romain, Luxembourg
Coallier, François, Canada
Coblentz, Brenda, USA
Cohen, Phil, Australia
Collard, Ross, New Zealand
Collignon, Stephane, Australia
Connors, Kathy Jo, USA
Cooper, Daniel, USA
Councill, Bill, USA
Cox, Margery, USA
Cunin, Pierre-Yves, France
DaLuz, Joseph, USA
Dampier, David, USA
Daneva, Maya, Canada
Daneva, Maya, Canada
Daughtry, Taz, USA
Davis, Ruth, USA
De Cesare, Sergio, UK
Dekleva, Sasa, USA
Del Castillo, Federico, Peru
Del Dago, Gustavo, Argentina
DeWeese, Perry, USA
Di Nunno, Donn, USA
Diaz-Herrera, Jorge, USA
Dieste, Oscar, Spain
Dion, Francis, Canada
Dixon, Wes, USA
Dolado, Javier, Spain
Donaldson, John, UK
Dorantes, Marco, Mexico
Dorofee, Audrey, USA
Douglass, Keith, Canada
Du, Weichang, Canada
Duben, Anthony, USA
Dudash, Edward, USA
Duncan, Scott, USA
Duong, Vinh, Canada
Durham, George, USA

Dutil, Daniel, Canada
Dutton, Jeffrey, USA
Ebert, Christof, France
Edge, Gary, USA
Edwards, Helen Maria, UK
El-Kadi, Amr, Egypt
Endres, David, USA
Engelmann, Franz, Switzerland
Escue, Marilyn, USA
Espinoza, Marco, Peru
Fay, Istvan, Hungary
Fayad, Mohamed, USA
Fendrich, John, USA
Ferguson, Robert, USA
Fernandez, Eduardo, USA
Fernandez-Sanchez, Jose Luis, Spain
Filgueiras, Lucia, Brazil
Finkelstein, Anthony, UK
Flinchbaugh, Scott, USA
Forrey, Arden, USA
Fortenberry, Kirby, USA
Foster, Henrietta, USA
Fowler, Martin, USA
Fowler, John Jr., USA
Fox, Christopher, USA
Frankl, Phyllis, USA
Freibergs, Imants, Latvia
Frezza, Stephen, USA
Fruehauf, Karol, Switzerland
Fuggetta, Alphonso, Italy
Fujii, Roger, USA
FUSCHI, David Luigi, Italy
Fuschi, David Luigi, Italy
Gabrini, Philippe, Canada
Gagnon, Eric, Canada
Ganor, Eitan, Israel
Garbajosa, Juan, Spain
Garceau, Benoît, Canada
Garcia-Palencia, Omar, Colombia
Garner, Barry, USA
Gelperin, David, USA
Gersting, Judith, Hawaii
Giesler, Gregg, USA
Gil, Indalecio, Spain
Gilchrist, Thomas, USA
Giurescu, Nicolae, Canada
Glass, Robert, USA
Glynn, Garth, UK
Goers, Ron, USA
Gogates, Gregory, USA
Goldsmith, Robin, USA
Goodbrand, Alan, Canada
Gorski, Janusz, Poland
Graybill, Mark, USA
Gresse von Wangenheim, Christiane, Brazil
Grigonis, George, USA
Gupta, Arun, USA
Gustafson, David, USA
Gutcher, Frank, USA
Haas, Bob, USA
Hagar, Jon, USA
Hagstrom, Erick, USA
Hailey, Victoria, Canada
Hall, Duncan, New Zealand
Haller, John, USA
Halstead-Nussloch, Richard, USA
Hamm, Linda, USA
Hankewitz, Lutz, Germany
Harker, Rob, USA
Hart, Hal, USA
Hart, Ronald, USA
Hartner, Clinton, USA
Hayeck, Elie, USA
He, Zhonglin, UK
Hedger, Dick, USA
Hefner, Rick, USA
Heinrich, Mark, USA
Heinze, Sherry, Canada
Hensel, Alan, USA
Herrmann, Debra, USA
Hesse, Wolfgang, Germany
Hilburn, Thomas, USA
Hill, Michael, USA
Ho, Vinh, Canada
Hodgen, Bruce, Australia
Hodges, Brett, Canada
Hoffman, Douglas, Canada
Hoffman, Michael, USA
Hoganson, Tammy, USA
Hollocker, Chuck, USA
Horch, John, USA
Howard, Adrian, UK
Huang, Hui Min, USA
Hung, Chih-Cheng, USA
Hung, Peter, USA
Hunt, Theresa, USA
Hunter, John, USA
Hvannberg, Ebba Thora, Iceland
Hybertson, Duane, USA
Ikiz, Seckin, Turkey
Iyengar, Dwaraka, USA
Jackelen, George, USA
Jaeger, Dawn, USA
Jahnke, Jens, Canada
James, Jean, USA
Jino, Mario, Brazil
Johnson, Vandy, USA
Jones, Griffin, USA
Jones, James E., USA
Jones, Alan, UK
Jones, James, USA
Jones, Larry, Canada
Jones, Paul, USA
Ju, Dehua, China
Juan-Martinez, Manuel-Fernando, Spain
Juhasz, Zoltan, Hungary
Juristo, Natalia, Spain
Kaiser, Michael, Switzerland
Kambic, George, USA
Kamthan, Pankaj, Canada
Kaner, Cem, USA
Kark, Anatol, Canada
Kasser, Joe, USA
Kasser, Joseph, Australia
Katz, Alf, Australia
Kececi, Nihal, Canada
Kell, Penelope, USA
Kelly, Diane, Canada
Kelly, Frank, USA
Kenett, Ron, Israel
Kenney, Mary L., USA
Kerievsky, Joshua, USA
Kerr, John, USA
Kierzyk, Robert, USA
Kinsner, W., Canada
Kirkpatrick, Harry, USA
Kittiel, Linda, USA
Klappholz, David, USA
Klein, Joshua, Israel
Knight, Claire, UK
Knoke, Peter, USA
Ko, Roy, Hong Kong
Kolewe, Ralph, Canada
Komal, Surinder Singh, Canada
Kovalovsky, Stefan, Austria
Krauth, Péter, Hungary
Krishnan, Nirmala, USA
Kromholz, Alfred, Canada
Kruchten, Philippe, Canada
Kuehner, Nathanael, Canada
Kwok, Shui Hung, Canada
Lacroix, Dominique, Canada
LaMotte, Stephen W., USA
Land, Susan, USA
Lange, Douglas, USA
Laporte, Claude, Canada
Lawlis, Patricia, USA
Le, Thach, USA
Leavitt, Randal, Canada
LeBel, Réjean, Canada
Leciston, David, USA
Lee, Chanyoung, USA
Lehman, Meir (Manny), UK

Leigh, William, USA
Lembo, Jim, USA
Lenss, John, USA
Leonard, Eugene, USA
Lethbridge, Timothy, Canada
Leung, Hareton, Hong Kong
Lever, Ronald, The Netherlands
Levesque, Ghislain, Canada
Ley, Earl, USA
Linders, Ben, Netherlands
Linscomb, Dennis, USA
Little, Joyce Currie, USA
Logan, Jim, USA
Long, Carol, UK
Lounis, Hakim, Canada
Low, Graham, Australia
Lutz, Michael, USA
Lynch, Gary, USA
Machado, Cristina, Brazil
MacKay, Stephen, Canada
MacKenzie, Garth, USA
MacNeil, Paul, USA
Magel, Kenneth, USA
Mains, Harold, USA
Malak, Renee, USA
Maldonado, José Carlos, Brazil
Marcos, Esperanza, Spain
Marinescu, Radu, Romania
Marm, Waldo, Peru
Marusca, Ioan, Canada
Matlen, Duane, USA
Matsumoto, Yoshihiro, Japan
McBride, Tom, Australia
McCarthy, Glenn, USA
McChesney, Ian, UK
McCormick, Thomas, Canada
McCown, Christian, USA
McDonald, Jim, USA
McGrath Carroll, Sue, USA
McHutchison, Diane, USA
McKinnell, Brian, Canada
McMichael, Robert, USA
McMillan, William, USA
McQuaid, Patricia, USA
Mead, Nancy, USA
Meeuse, Jaap, The Netherlands
Meier, Michael, USA
Meisenzahl, Christopher, USA
Melhart, Bonnie, USA
Mengel, Susan, USA
Meredith, Denis, USA
Meyerhoff, Dirk, Germany
Mili, Hafedh, Canada
Miller, Chris, Netherlands
Miller, Keith, USA
Miller, Mark, USA
Miranda, Eduardo, Canada
Mistrik, Ivan, Germany
Mitasiunas, Antanas, Lithuania
Modell, Howard, USA
Modell, Staiger,USA
Modesitt, Kenneth, USA
Moland, Kathryn, USA
Moldavsky, Symon, Ukraine
Montequín, Vicente R., Spain
Moreno, Ana Maria, Spain
Mosiuoa, Tseliso, Lesotho
Moudry, James, USA
Msheik, Hamdan, Canada
Mularz, Diane, USA
Mullens, David, USA
Müllerburg, Monika, Germany
Murali, Nagarajan, Australia
Murphy, Mike, USA
Napier, John, USA
Narasimhadevara, Sudha, Canada
Narawane, Ranjana, India
Narayanan, Ramanathan, India
Navarro Ramirez, Daniel, Mexico
Navas Plano, Francisco, Spain
Navrat, Pavol, Slovakia
Neumann, Dolly, USA
Nguyen-Kim, Hong, Canada
Nikandros, George, Australia
Nishiyama, Tetsuto, Japan
Nunn, David, USA
O'Donoghue, David, Ireland
Oliver, David John, Australia
Olson, Keith, USA
Oskarsson, Östen, Sweden
Ostrom, Donald, USA
Oudshoorn, Michael, Australia
Owen, Cherry, USA
Pai, Hsueh-Ieng, Canada
Parrish, Lee, USA
Parsons, Samuel, USA
Patel, Dilip, UK
Paulk, Mark, USA
Pavelka, Jan, Czech Republic
Pavlov, Vladimir, Ukraine
Pawlyszyn, Blanche, USA
Pecceu, Didier, France
Perisic, Branko, Yugoslavia
Perry, Dale, USA
Peters, Dennis, Canada
Petersen, Erik, Australia
Pfahl, Dietmar, Germany
Pfeiffer, Martin, Germany
Phillips, Dwayne, USA
Phipps, Robert, USA
Phister, Paul, USA
Phister, Jr., Paul, USA
Piattini, Mario, Spain
Piersall, Jeff, USA
Pillai, S.K., India
Pinder, Alan, UK
Pinheiro, Francisco A., Brazil
Plekhanova, Valentina, UK
Poon, Peter, USA
Poppendieck, Mary, USA
Powell, Mace, USA
Predenkoski, Mary, USA
Prescott, Allen, USA
Pressman, Roger, USA
Price, Art, USA
Price, Margaretha, USA
Pullum, Laura, USA
Purser, Keith, USA
Purssey, John, Australia
Pustaver, John, USA
Quinn, Anne, USA
Radnell, David, Australia
Rae, Andrew, UK
Rafea, Ahmed, Egypt
Ramsden, Patrick, Australia
Rao, N.Vyaghrewara, India
Rawsthorne, Dan, USA
Reader, Katherine, USA
Reddy, Vijay,USA
Redwine, Samuel, USA
Reed, Karl, Australia
Reedy, Ann, USA
Reeker, Larry, USA
Rethard, Tom, USA
Reussner, Ralf, Germany
Rios, Joaquin, Spain
Risbec, Philippe, France
Roach, Steve, USA
Robillard, Pierre, Canada
Rocha, Zalkind, Brazil
Rodeiro Iglesias, Javier, Spain
Rodriguez-Dapena, Patricia, Spain
Rogoway, Paul, Israel
Rontondi, Guido, Italy
Roose, Philippe, France
Rosca, Daniela, USA
Rosenberg, Linda, USA
Rourke, Michael, Australia
Rout, Terry, Australia
Rufer, Russ, USA
Ruiz, Francisco, Spain
Ruocco, Anthony, USA
Rutherfoord, Rebecca, USA
Ryan, Michael, Ireland
Salustri, Filippo, Canada

Salustri, Filippo, Canada
Salwin, Arthur, USA
Sanden, Bo, USA
Sandmayr, Helmut, Switzerland
Santana Filho, Ozeas Vieira, Brazil
Sato, Tomonobu, Japan
satyadas, antony, USA
Satyadas, Antony, USA
Schaaf, Robert, USA
Scheper, Charlotte, USA
Schiffel, Jeffrey, USA
Schlicht, Bill, USA
Schrott, William, USA
Schwarm, Stephen, USA
Schweppe, Edmund, USA
Sebern, Mark, USA
Seffah, Ahmed, Canada
Selby, Nancy, USA
Selph, William, USA
Sen, Dhruba, USA
Senechal, Raymond, USA
Sepulveda, Christian, USA
Setlur, Atul, USA
Sharp, David, USA
Shepard, Terry, Canada
Shepherd, Alan, Germany
Shillato, Rrobert W, USA
Shintani, Katsutoshi, Japan
Silva, Andres, Spain
Silva, Andres, Spain
Singer, Carl, USA
Sinnett, Paul, UK
Sintzoff, André, France
Sitte, Renate, Australia
Sky, Richard, USA
Smilie, Kevin, USA
Smith, David, USA
Sophatsathit, Peraphon, Thailand
Sorensen, Reed, USA
Soundarajan, Neelam, USA
Sousa Santos, Frederico, Portugal
Spillers, Mark, USA
Spinellis, Diomidis, Greece
Splaine, Steve, USA
Springer, Donald, USA
Staiger, John, USA
Starai, Thomas, USA
Steurs, Stefan, Belgium
St-Pierre, Denis, Canada
Stroulia, Eleni, Canada
Subramanian, K.S., India
Sundaram, Sai, UK
Swanek, James, USA
Swearingen, Sandra, USA
Szymkowiak, Paul, Canada
Tamai, Tetsuo, Japan
Tasker, Dan, New Zealand
Taylor, Stanford, USA
Terekhov, Andrey A., Russian Federation
Terski, Matt, USA
Thayer, Richard, USA
Thomas, Michael, USA
Thompson, A. Allan, Australia
Thompson, John Barrie, UK
Titus, Jason, USA
Tockey, Steve, USA
Tovar, Edmundo, Spain
Towhidnejad, Massood, USA
Trellue, Patricia, USA
Trèves, Nicolas, France
Troy, Elliot, USA
Tsui, Frank, USA
Tsuneo, Furuyama, Japan
Tuohy, Kenney, USA
Tuohy, Marsha P., USA
Turczyn, Stephen, USA
Upchurch, Richard, USA
Urbanowicz, Theodore, USA
Van Duine, Dan, USA
Van Ekris, Jaap, Netherlands
Van Oosterhout, Bram, Australia
Vander Plaats, Jim, USA
Vegas, Sira, Spain
Verner, June, USA
Villas-Boas, André, Brazil
Vollman, Thomas, USA
Walker, Richard, Australia
Walsh, Bucky, USA
Wang, Yingxu, Sweden
Wear, Larry, USA
Weigel, richard, USA
Weinstock, Charles, USA
Wenyin, Liu, China
Werner, Linda, USA
Wheeler, David, USA
White, Nathan, USA
White, Stephanie, USA
Whitmire, Scott, USA
Wijbrans, Klaas, The Netherlands
Wijbrans-Roodbergen, Margot, The Netherlands
Wilkie, Frederick, UK
Wille, Cornelius, Germany
Wilson, Charles, USA
Wilson, Leon, USA
Wilson, Russell, USA
Woechan, Kenneth, USA
Woit, Denise, Canada
Yadin, Aharon, Israel
Yih, Swu, Taiwan
Young, Michal, USA
Yrivarren, Jorge, Peru
Znotka, Juergen, Germany
Zuser, Wolfgang, Austria
Zvegintzov, Nicholas, USA
Zweben, Stu, USA

The following motion was unanimously adopted by the Industrial Advisory Board on 6 February 2004.

The Industrial Advisory Board finds that the Software Engineering Body of Knowledge project initiated in 1998 has been successfully completed; and endorses the 2004 Version of the Guide to the SWEBOK and commends it to the IEEE Computer Society Board of Governors for their approval.

The following motion was adopted by the IEEE Computer Society Board of Governors in February 2004.

MOVED, that the Board of Governors of the IEEE Computer Society approves the 2004 Edition of the Guide to the Software Engineering Body of Knowledge and authorizes the Chair of the Professional Practices Committee to proceed with printing.

PREFACE

Software engineering is an emerging discipline and there are unmistakable trends indicating an increasing level of maturity:

- Several universities throughout the world offer undergraduate degrees in software engineering. For example, such degrees are offered at the University of New South Wales (Australia), McMaster University (Canada), the Rochester Institute of Technology (US), the University of Sheffield (UK), and other universities.
- In the US, the Engineering Accreditation Commission of the Accreditation Board for Engineering and Technology (ABET) is responsible for the accreditation of undergraduate software engineering programs.
- The Canadian Information Processing Society has published criteria to accredit software engineering undergraduate university programs.
- The Software Engineering Institute's Capability Maturity Model for Software (SW CMM) and the new Capability Maturity Model Integration (CMMI) are used to assess organizational capability for software engineering. The famous ISO 9000 quality management standards have been applied to software engineering by the new ISO/IEC 90003.
- The Texas Board of Professional Engineers has begun to license professional software engineers.
- The Association of Professional Engineers and Geoscientists of British Columbia (APEGBC) has begun registering software professional engineers, and the Professional Engineers of Ontario (PEO) has also announced requirements for licensing.
- The Association for Computing Machinery (ACM) and the Computer Society of the Institute of Electrical and Electronics Engineers (IEEE) have jointly developed and adopted a Code of Ethics and Professional Practice for software engineering professionals.[1]
- The IEEE Computer Society offers the Certified Software Development Professional certification for software engineering. The Institute for Certification of Computing Professionals (ICCP) has long offered a certification for computing professionals.

All of these efforts are based upon the presumption that there is a Body of Knowledge that should be mastered by practicing software engineers. The Body of Knowledge exists in the literature that has accumulated over the past thirty years. This book provides a Guide to that Body of Knowledge.

PURPOSE

The purpose of the Guide to the Software Engineering Body of Knowledge is to provide a consensually validated characterization of the bounds of the software engineering discipline and to provide a topical access to the Body of Knowledge supporting that discipline. The Body of Knowledge is subdivided into ten software engineering Knowledge Areas (KA) plus an additional chapter providing an overview of the KAs of strongly related disciplines. The descriptions of the KAs are designed to discriminate among the various important concepts, permitting readers to find their way quickly to subjects of interest. Upon finding a subject, readers are referred to key papers or book chapters selected because they succinctly present the knowledge.

In browsing the Guide, readers will note that the content is markedly different from computer science. Just as electrical engineering is based upon the science of physics, software engineering should be based, among other things, upon computer science. In these two cases, though, the emphasis is necessarily different. Scientists extend our knowledge of the laws of nature while engineers apply those laws of nature to build useful artifacts, under a number of constraints. Therefore, the emphasis of the Guide is placed on the construction of useful software artifacts.

Readers will also notice that many important aspects of information technology that may constitute important software engineering knowledge are not covered in the Guide, including specific programming languages, relational databases, and networks. This is a consequence of an engineering-based approach. In all fields—not only computing—the designers of engineering curricula have realized that specific technologies are replaced much more rapidly than the engineering work force. An engineer must be equipped with the essential knowledge that supports the selection of the appropriate technology at the appropriate time in the appropriate circumstance. For

[1] The ACM/CS Software Engineering Code of Ethics and Professional Practice can be found at http://www.computer.org/certification/ethics.htm.

example, software might be built in Fortran using functional decomposition or in C++ using object-oriented techniques. The techniques for software configuring instances of those systems would be quite different. But, the principles and objectives of configuration management remain the same. The Guide therefore does not focus on the rapidly changing technologies, although their general principles are described in relevant KAs.

These exclusions demonstrate that this Guide is necessarily incomplete. The Guide covers software engineering knowledge that is necessary but not sufficient for a software engineer. Practicing software engineers will need to know many things about computer science, project management, and systems engineering—to name a few—that fall outside the Body of Knowledge characterized by this Guide. However, stating that this information should be known by software engineers is not the same as stating that this knowledge falls within the bounds of the software engineering discipline. Instead, it should be stated that software engineers need to know some things taken from other disciplines—and that is the approach adopted in this Guide. So, this Guide characterizes the Body of Knowledge falling within the scope of software engineering and provides references to relevant information from other disciplines. A chapter of the Guide provides a taxonomical overview of the related disciplines derived from authoritative sources.

The emphasis on engineering practice leads the Guide toward a strong relationship with the normative literature. Most of the computer science, information technology, and software engineering literature provides information useful to software engineers, but a relatively small portion is normative. A normative document prescribes what an engineer should do in a specified situation rather than providing information that might be helpful. The normative literature is validated by consensus formed among practitioners and is concentrated in standards and related documents. From the beginning, the SWEBOK project was conceived as having a strong relationship to the normative literature of software engineering. The two major standards bodies for software engineering (IEEE Computer Society Software Engineering Standards Committee and ISO/IEC JTC1/SC7) are represented in the project. Ultimately, we hope that software engineering practice standards will contain principles directly traceable to the Guide.

INTENDED AUDIENCE

The Guide is oriented toward a variety of audiences, all over the world. It aims to serve public and private organizations in need of a consistent view of software engineering for defining education and training requirements, classifying jobs, developing performance evaluation policies, or specifying software development tasks. It also addresses practicing, or managing, software engineers and the officials responsible for making public policy regarding licensing and professional guidelines. In addition, professional societies and educators defining the certification rules, accreditation policies for university curricula, and guidelines for professional practice will benefit from SWEBOK, as well as the students learning the software engineering profession and educators and trainers engaged in defining curricula and course content.

EVOLUTION OF THE GUIDE

From 1993 to 2000, the IEEE Computer Society and the Association for Computing Machinery (ACM) cooperated in promoting the professionalization of software engineering through their joint Software Engineering Coordinating Committee (SWECC). The Code of Ethics was completed under stewardship of the SWECC primarily through volunteer efforts. The SWEBOK project was initiated by the SWECC in 1998.

The SWEBOK project's scope, the variety of communities involved, and the need for broad participation suggested a need for full-time rather than volunteer management. For this purpose, the IEEE Computer Society contracted the Software Engineering Management Research Laboratory at the Université du Québec à Montréal (UQAM) to manage the effort. In recent years, UQAM has been joined by the École de technologie supérieure, Montréal, Québec.

The project plan comprised three successive phases: Strawman, Stoneman, and Ironman. An early prototype, Strawman, demonstrated how the project might be organized. The publication of the widely circulated Trial Version of the Guide in 2001 marked the end of the Stoneman phase of the project and initiated a period of trial usage. The current Guide marks the end of the Ironman period by providing a Guide that has achieved consensus through broad review and trial application.

The project team developed two important principles for guiding the project: *transparency* and *consensus*. By transparency, we mean that the development process is itself documented, published, and publicized so that important decisions and status are visible to all concerned parties. By consensus, we mean that the only practical method for legitimizing a statement of this kind is through broad participation and agreement by all significant sectors of the relevant community.

Literally hundreds of contributors, reviewers, and trial users have played a part in producing the current document.

Like any software project, the SWEBOK project has many stakeholders—some of which are formally represented. An Industrial Advisory Board, composed of representatives from industry (Boeing, Construx Software, the MITRE Corporation, Rational Software, Raytheon Systems, and SAP Labs-Canada), research agencies (National Institute of Standards and Technology, National Research Council of Canada), the Canadian Council of Professional Engineers, and the IEEE Computer Society, has provided financial support for the project. The IAB's generous support permits us to make the products of the SWEBOK project publicly available without any charge (see http://www.swebok.org). IAB membership is supplemented with the chairs of ISO/IEC JTC1/SC7 and the related Computing Curricula 2001 initiative. The IAB reviews and approves the project plans, oversees consensus building and review processes, promotes the project, and lends credibility to the effort. In general, it ensures the relevance of the effort to real-world needs.

The Trial Version of the Guide was the product of extensive review and comment. In three public review cycles, a total of roughly 500 reviewers from 42 countries provided roughly 9,000 comments, all of which are available at www.swebok.org. To produce the current version, we released the Trial Version for extensive trial usage. Trial application in specialized studies resulted in 17 papers describing good aspects of the Guide, as well as aspects needing improvement. A Web-based survey captured additional experience: 573 individuals from 55 countries registered for the survey; 124 reviewers from 21 countries actually provided comments—1,020 of them. Additional suggestions for improvement resulted from liaison with related organizations and efforts: IEEE-CS/ACM Computing Curricula Software Engineering; the IEEE CS Certified Software Development Professional project; ISO/IEC JTC1/SC7 (software and systems engineering standards); the IEEE Software Engineering Standards Committee; the American Society for Quality, Software Division; and an engineering professional society, the Canadian Council of Professional Engineers.

CHANGES SINCE THE TRIAL VERSION

The overall goal of the current revision was to improve the readability, consistency, and usability of the Guide. This implied a general rewrite of the entire text to make the style consistent throughout the document. In several cases, the topical breakdown of the KA was rearranged to make it more usable, but we were careful to include the same information that was approved by the earlier consensus process. We updated the reference list so that it would be easier to obtain the references.

Trial usage resulted in the recommendation that three KAs should be rewritten. Practitioners remarked that the Construction KA was difficult to apply in a practical context. The Management KA was perceived as being too close to general management and not sufficiently specific to software engineering concerns. The Quality KA was viewed as an uncomfortable mix of process quality and product quality; it was revised to emphasize the latter.

Finally, some KAs were revised to remove material duplicating that of other KAs.

LIMITATIONS

Even though the Guide has gone through an elaborate development and review process, the following limitations of this process must be recognized and stated:

- Software engineering continues to be infused with new technology and new practices. Acceptance of new techniques grows and older techniques are discarded. The topics listed as "generally accepted" in this Guide are carefully selected at this time. Inevitably, though, the selection will need to evolve.
- The amount of literature that has been published on software engineering is considerable and the reference material included in this Guide should not be seen as a definitive selection but rather as a reasonable selection. Obviously, there are other excellent authors and excellent references than those included in the Guide. In the case of the Guide, references were selected because they are written in English, readily available, recent, and easily readable, and—taken as a group—they provide coverage of the topics within the KA.
- Important and highly relevant reference material written in languages other than English have been omitted from the selected reference material.

Additionally, one must consider that

- Software engineering is an emerging discipline. This is especially true if you compare it to certain more established engineering disciplines. This means notably that the boundaries between the KAs of software engineering and between software engineering and its related disciplines remain a matter for continued evolution.

The contents of this Guide must therefore be viewed as an "informed and reasonable" characterization of the software engineering Body of Knowledge and as baseline for future evolution. Additionally, please note that the Guide is not attempting nor does it claim to replace or amend in any way laws, rules, and procedures that have been defined by official public policy makers around the world regarding the practice and definition of engineering and software engineering in particular.

Alain Abran
École de technologie supérieure

Pierre Bourque
École de Technologie Supérieure

Leonard Tripp
1999 President
IEEE Computer Society

Executive Editors of the Guide to the Software Engineering Body of Knowledge

Editors of the Guide to the Software Engineering Body of Knowledge

Chair of the Professional Practices Committee, IEEE Computer Society

(2001-2003)

James W. Moore
The MITRE Corporation

Robert Dupuis
Université du Québec à Montréal

December 2004

The SWEBOK project Web site is http://www.swebok.org/

ACKNOWLEDGMENTS

The SWEBOK editorial team gratefully acknowledges the support provided by the members of the Industrial Advisory Board. Funding for this project has been provided by the ACM, Boeing, the Canadian Council of Professional Engineers, Construx Software, the IEEE Computer Society, the MITRE Corporation, the National Institute of Standards and Technology, the National Research Council of Canada, Rational Software, Raytheon Company, and SAP Labs (Canada). The team is thankful for the counsel provided by the Panel of Experts. The team also appreciates the important work performed by the Associate Editors. We would also like to express our gratitude for initial work on the Knowledge Area Descriptions completed by Imants Freibergs, Stephen Frezza, Andrew Gray, Vinh T. Ho, Michael Lutz, Larry Reeker, Guy Tremblay, Chris Verhoef, and Sybille Wolff. The editorial team must also acknowledge the indispensable contribution of the hundreds of reviewers.

The editorial team also wishes to thank the following people who contributed to the project in various ways: Mark Ardis, Yussef Belkebir, Michel Boivin, Julie Bonneau, Simon Bouchard, François Cossette, Vinh Duong, Gilles Gauthier, Michèle Hébert, Paula Hawthorn, Richard W. Heiman, Julie Hudon, Idrissa Konkobo, Rene Köppel, Lucette Lapointe, Claude Laporte, Luis Molinié, Hamdan Msheik, Iphigénie N'Diyae, Serge Oligny, Suzanne Paquette, Keith Paton, Dave Rayford, Normand Séguin, Paul Sinnett, Denis St-Pierre, Dale Strok, Pascale Tardif, Louise Thibaudeau, Dolores Wallace, Évariste Valery Bevo Wandji, and Michal Young.

Finally, there are surely other people who have contributed to this Guide, either directly or indirectly, whose names we have inadvertently omitted. To those people, we offer our tacit appreciation and apologize for having omitted explicit recognition here.

CHAPTER 1

INTRODUCTION TO THE GUIDE

In spite of the millions of software professionals worldwide and the ubiquitous presence of software in our society, software engineering has only recently reached the status of a legitimate engineering discipline and a recognized profession.

Achieving consensus by the profession on a core body of knowledge is a key milestone in all disciplines and had been identified by the IEEE Computer Society as crucial for the evolution of software engineering towards professional status. This Guide, written under the auspices of the Professional Practices Committee, is part of a multi-year project designed to reach such a consensus.

WHAT IS SOFTWARE ENGINEERING?

The IEEE Computer Society defines software engineering as

"(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

(2) The study of approaches as in (1)."[1]

WHAT IS A RECOGNIZED PROFESSION?

For software engineering to be fully known as a legitimate engineering discipline and a recognized profession, consensus on a core body of knowledge is imperative. This fact is well illustrated by Starr when he defines what can be considered a legitimate discipline and a recognized profession. In his Pulitzer Prize-winning book on the history of the medical profession in the USA, he states,

"The legitimization of professional authority involves three distinctive claims: first, that the knowledge and competence of the professional have been validated by a community of his or her peers; second, that this consensually validated knowledge rests on rational, scientific grounds; and third, that the professional's judgment and advice are oriented toward a set of substantive values, such as health. These aspects of legitimacy correspond to the kinds of attributes—collegial, cognitive, and moral—usually embodied in the term "profession."[2]

WHAT ARE THE CHARACTERISTICS OF A PROFESSION?

Gary Ford and Norman Gibbs studied several recognized professions, including medicine, law, engineering, and accounting.[3] They concluded that an engineering profession is characterized by several components:

- An initial *professional education* in a curriculum validated by society through *accreditation*
- Registration of fitness to practice via voluntary *certification* or mandatory *licensing*
- Specialized *skill development* and *continuing professional education*
- Communal support via a *professional society*
- A commitment to norms of conduct often prescribed in a *code of ethics*

This Guide contributes to the first three of these components. Articulating a Body of Knowledge is an essential step toward developing a profession because it represents a broad consensus regarding what a software engineering professional should know. Without such a consensus, no licensing examination can be validated, no curriculum can prepare an individual for an examination, and no criteria can be formulated for accrediting a curriculum. The development of consensus is also a prerequisite to the adoption of coherent skills development and continuing professional education programs in organizations.

WHAT ARE THE OBJECTIVES OF THE SWEBOK PROJECT?

The Guide should not be confused with the Body of Knowledge itself, which already exists in the published literature. The purpose of the Guide is to describe what portion of the Body of Knowledge is generally accepted, to organize that portion, and to provide a topical access to it. Additional information on the meaning given to "generally accepted" can be found below and in Appendix A.

The Guide to the Software Engineering Body of Knowledge (SWEBOK) was established with the following five objectives:

1. To promote a consistent view of software engineering worldwide
2. To clarify the place—and set the boundary—of software engineering with respect to other disciplines such as computer science, project management, computer engineering, and mathematics
3. To characterize the contents of the software engineering discipline

---

[1] "IEEE Standard Glossary of Software Engineering Terminology," IEEE std 610.12-1990, 1990.

[2] P. Starr, *The Social Transformation of American Medicine*, Basic Books, 1982, p. 15.

[3] G. Ford and N.E. Gibbs, *A Mature Profession of Software Engineering*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., tech. report CMU/SEI-96-TR-004, Jan. 1996.

4. To provide a topical access to the Software Engineering Body of Knowledge
5. To provide a foundation for curriculum development and for individual certification and licensing material

The first of these objectives, a consistent worldwide view of software engineering, was supported by a development process which engaged approximately 500 reviewers from 42 countries in the Stoneman phase (1998–2001) leading to the Trial version, and over 120 reviewers from 21 countries in the Ironman phase (2003) leading to the 2004 version. More information regarding the development process can be found in the Preface and on the Web site (www.swebok.org). Professional and learned societies and public agencies involved in software engineering were officially contacted, made aware of this project, and invited to participate in the review process. Associate editors were recruited from North America, the Pacific Rim, and Europe. Presentations on the project were made at various international venues and more are scheduled for the upcoming year.

The second of the objectives, the desire to set a boundary for software engineering, motivates the fundamental organization of the Guide. The material that is recognized as being within this discipline is organized into the first ten Knowledge Areas (KAs) listed in Table 1. Each of these KAs is treated as a chapter in this Guide.

Table 1 The SWEBOK Knowledge Areas (KAs)

Software requirements
Software design
Software construction
Software testing
Software maintenance
Software configuration management
Software engineering management
Software engineering process
Software engineering tools and methods
Software quality

In establishing a boundary, it is also important to identify what disciplines share that boundary, and often a common intersection, with software engineering. To this end, the Guide also recognizes eight related disciplines, listed in Table 2 (see Chapter 12, "Related Disciplines of Software Engineering"). Software engineers should, of course, have knowledge of material from these fields (and the KA descriptions may make reference to them). It is not, however, an objective of the SWEBOK Guide to characterize the knowledge of the related disciplines, but rather what knowledge is viewed as specific to software engineering.

Table 2 Related disciplines

◆ Computer engineering	◆ Project management
◆ Computer science	◆ Quality management
◆ Management	◆ Software ergonomics
◆ Mathematics	◆ Systems engineering

HIERARCHICAL ORGANIZATION

The organization of the KA descriptions or chapters supports the third of the project's objectives—a characterization of the contents of software engineering. The detailed specifications provided by the project's editorial team to the associate editors regarding the contents of the KA descriptions can be found in Appendix A.

The Guide uses a hierarchical organization to decompose each KA into a set of topics with recognizable labels. A two- or three-level breakdown provides a reasonable way to find topics of interest. The Guide treats the selected topics in a manner compatible with major schools of thought and with breakdowns generally found in industry and in software engineering literature and standards. The breakdowns of topics do not presume particular application domains, business uses, management philosophies, development methods, and so forth. The extent of each topic's description is only that needed to understand the generally accepted nature of the topics and for the reader to successfully find reference material. After all, the Body of Knowledge is found in the reference material themselves, not in the Guide.

REFERENCE MATERIAL AND MATRIX

To provide a topical access to the knowledge—the fourth of the project's objectives—the Guide identifies reference material for each KA, including book chapters, refereed papers, or other recognized sources of authoritative information. Each KA description also includes a matrix relating the reference material to the listed topics. The total volume of cited literature is intended to be suitable for mastery through the completion of an undergraduate education plus four years of experience.

In this edition of the Guide, all KAs were allocated around 500 pages of reference material, and this was the specification the associate editors were invited to apply. It may be argued that some KAs, such as software design for instance, deserve more pages of reference material than others. Such modulation may be applied in future editions of the Guide.

It should be noted that the Guide does not attempt to be comprehensive in its citations. Much material that is both suitable and excellent is not referenced. Material was selected in part because—taken as a collection—it provides coverage of the topics described.

DEPTH OF TREATMENT

From the outset, the question arose as to the depth of treatment

the Guide should provide. The project team adopted an approach which supports the fifth of the project's objectives—providing a foundation for curriculum development, certification, and licensing. The editorial team applied the criterion of *generally accepted* knowledge, to be distinguished from advanced and research knowledge (on the grounds of maturity) and from specialized knowledge (on the grounds of generality of application). The definition comes from the Project Management Institute: "The generally accepted knowledge applies to most projects most of the time, and widespread consensus validates its value and effectiveness."[4]

Specialized Practices used only for certain types of software	Generally Accepted Established traditional practices recommended by many organizations
	Advanced and Research Innovative practices tested and used only by some organizations and concepts still being developed and tested in research organizations

Figure 1 Categories of knowledge

However, the term "generally accepted" does not imply that the designated knowledge should be uniformly applied to all software engineering endeavors—each project's needs determine that—but it does imply that competent, capable software engineers should be equipped with this knowledge for potential application. More precisely, generally accepted knowledge should be included in the study material for the software engineering licensing examination that graduates would take after gaining four years of work experience. Although this criterion is specific to the US style of education and does not necessarily apply to other countries, we deem it useful. However, the two definitions of generally accepted knowledge should be seen as complementary.

LIMITATIONS RELATED TO THE BOOK FORMAT

The book format for which this edition was conceived has its limitations. The nature of the contents would be better served using a hypertext structure, where a topic would be linked to topics other than the ones immediately preceding and following it in a list.

Some boundaries between KAs, subareas, and so on are also sometimes relatively arbitrary. These boundaries are not to be given too much importance. As much as possible, pointers and links have been given in the text where relevant and useful.

Links between KAs are not of the input-output type. The KAs are meant to be views on the knowledge one should possess in software engineering with respect to the KA in question. The decomposition of the discipline within KAs and the order in which the KAs are presented are not to be assimilated with any particular method or model. The methods are described in the appropriate KA in the Guide, and the Guide itself is not one of them.

THE KNOWLEDGE AREAS

Figure 1 maps out the eleven chapters and the important topics incorporated within them. The first five KAs are presented in traditional waterfall life-cycle sequence. However, this does not imply that the Guide adopts or encourages the waterfall model, or any other model. The subsequent KAs are presented in alphabetical order, and those of the related disciplines are presented in the last chapter. This is identical to the sequence in which they are presented in this Guide.

STRUCTURE OF THE KA DESCRIPTIONS

The KA descriptions are structured as follows.

In the introduction, a brief definition of the KA and an overview of its scope and of its relationship with other KAs are presented.

The breakdown of topics constitutes the core of each KA description, describing the decomposition of the KA into subareas, topics, and sub-topics. For each topic or sub-topic, a short description is given, along with one or more references.

The reference material was chosen because it is considered to constitute the best presentation of the knowledge relative to the topic, taking into account the limitations imposed on the choice of references (see above). A matrix links the topics to the reference material.

The last part of the KA description is the list of recommended references. Appendix A of each KA includes suggestions for further reading for those users who wish to learn more about the KA topics. Appendix B presents the list of standards most relevant to the KA. Note that citations enclosed in square brackets "[ ]" in the text identify recommended references, while those enclosed in parentheses "( )" identify the usual references used to write or justify the text. The former are to be found in the corresponding section of the KA and the latter in Appendix A of the KA.

Brief summaries of the KA descriptions and appendices are given next.

SOFTWARE REQUIREMENTS KA (SEE FIGURE 2, COLUMN A)

A requirement is defined as a property that must be exhibited in order to solve some real-world problem.

The first knowledge subarea is *Software Requirements Fundamentals*. It includes definitions of software requirements

[4] *A Guide to the Project Management Body of Knowledge*, 2000 ed., Project Management Institute, www.pmi.org.

themselves, but also of the major types of requirements: product vs. process, functional vs. nonfunctional, emergent properties. The subarea also describes the importance of quantifiable requirements and distinguishes between systems and software requirements.

The second knowledge subarea is the *Requirements Process*, which introduces the process itself, orienting the remaining five subareas and showing how requirements engineering dovetails with the other software engineering processes. It describes process models, process actors, process support and management, and process quality and improvement.

The third subarea is *Requirements Elicitation*, which is concerned with where software requirements come from and how the software engineer can collect them. It includes requirement sources and elicitation techniques.

The fourth subarea, *Requirements Analysis*, is concerned with the process of analyzing requirements to

- Detect and resolve conflicts between requirements
- Discover the bounds of the software and how it must interact with its environment
- Elaborate system requirements to software requirements

Requirements analysis includes requirements classification, conceptual modeling, architectural design and requirements allocation, and requirements negotiation.

The fifth subarea is *Requirements Specification*. Requirements specification typically refers to the production of a document, or its electronic equivalent, that can be systematically reviewed, evaluated, and approved. For complex systems, particularly those involving substantial non-software components, as many as three different types of documents are produced: system definition, system requirements specification, and software requirements specification. The subarea describes all three documents and the underlying activities.

The sixth subarea is *Requirements Validation*, the aim of which is to pick up any problems before resources are committed to addressing the requirements. Requirements validation is concerned with the process of examining the requirements documents to ensure that they are defining the right system (that is, the system that the user expects). It is subdivided into descriptions of the conduct of requirements reviews, prototyping, and model validation and acceptance tests.

The seventh and last subarea is *Practical Considerations*. It describes topics which need to be understood in practice. The first topic is the iterative nature of the requirements process. The next three topics are fundamentally about change management and the maintenance of requirements in a state which accurately mirrors the software to be built, or that has already been built. It includes change management, requirements attributes, and requirements tracing. The final topic is requirements measurement.

SOFTWARE DESIGN KA (SEE FIGURE 2, COLUMN B)

According to the IEEE definition [IEEE 610.12-90], design is both "the process of defining the architecture, components, interfaces, and other characteristics of a system or component" and "the result of [that] process." The KA is divided into six subareas.

The first subarea presents *Software Design Fundamentals*, which form an underlying basis to the understanding of the role and scope of software design. These are general software concepts, the context of software design, the software design process, and the enabling techniques for software design.

The second subarea groups together the *Key Issues in Software Design*. They include concurrency, control and handling of events, distribution of components, error and exception handling and fault tolerance, interaction and presentation, and data persistence.

The third subarea is *Software Structure and Architecture*, the topics of which are architectural structures and viewpoints, architectural styles, design patterns, and, finally, families of programs and frameworks.

The fourth subarea describes *software Design Quality Analysis and Evaluation*. While there is a entire KA devoted to software quality, this subarea presents the topics specifically related to software design. These aspects are quality attributes, quality analysis, and evaluation techniques and measures.

The fifth subarea is *Software Design Notations*, which are divided into structural and behavioral descriptions.

The last subarea describes *Software Design Strategies and Methods*. First, general strategies are described, followed by function-oriented design methods, object-oriented design methods, data-structure-centered design, component- based design, and others.

SOFTWARE CONSTRUCTION KA (SEE FIGURE 2, COLUMN C)

Software construction refers to the detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging. The KA includes three subareas.

The first subarea is *Software Construction Fundamentals*. The first three topics are basic principles of construction: minimizing complexity, anticipating change, and constructing for verification. The last topic discusses standards for construction.

The second subarea describes *Managing Construction*. The topics are construction models, construction planning, and construction measurement.

The third subarea covers *Practical Considerations*. The topics are construction design, construction languages, coding, construction testing, reuse, construction quality, and integration.

SOFTWARE TESTING (SEE FIGURE 2, COLUMN D)

Software Testing consists of the dynamic verification of the

behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior. It includes five subareas.

It begins with a description of *Software Testing Fundamentals*. First, the testing-related terminology is presented, then key issues of testing are described, and finally the relationship of testing to other activities is covered.

The second subarea is *Test Levels*. These are divided between the targets of the tests and the objectives of the tests.

The third subarea is *Test Techniques*. The first category includes the tests based on the tester's intuition and experience. A second group comprises specification-based techniques, followed by code-based techniques, fault-based techniques, usage-based techniques, and techniques relative to the nature of the application. A discussion of how to select and combine the appropriate techniques is also presented.

The fourth subarea covers *Test-Related Measures*. The measures are grouped into those related to the evaluation of the program under test and the evaluation of the tests performed.

The last subarea describes the *Test Process* and includes practical considerations and the test activities.

SOFTWARE MAINTENANCE (SEE FIGURE 2, COLUMN E)

Once in operation, anomalies are uncovered, operating environments change, and new user requirements surface. The maintenance phase of the life cycle commences upon delivery, but maintenance activities occur much earlier. The Software Maintenance KA is divided into four subareas.

The first one presents *Software Maintenance Fundamentals*: definitions and terminology, the nature of maintenance, the need for maintenance, the majority of maintenance costs, the evolution of software, and the categories of maintenance.

The second subarea groups together the *Key Issues in Software Maintenance*. These are the technical issues, the management issues, maintenance cost estimation, and software maintenance measurement.

The third subarea describes the *Maintenance Process*. The topics here are the maintenance processes and maintenance activities.

Techniques for Maintenance constitute the fourth subarea. These include program comprehension, re-engineering, and reverse engineering.

SOFTWARE CONFIGURATION MANAGEMENT (SEE FIGURE 3, COLUMN F)

Software Configuration Management (SCM) is the discipline of identifying the configuration of software at distinct points in time for the purpose of systematically controlling changes to the configuration and of maintaining the integrity and traceability of the configuration throughout the system life cycle. This KA includes six subareas.

The first subarea is *Management of the SCM Process*. It covers the topics of the organizational context for SCM, constraints and guidance for SCM, planning for SCM, the SCM plan itself, and surveillance of SCM.

The second subarea is *Software Configuration Identification*, which identifies items to be controlled, establishes identification schemes for the items and their versions, and establishes the tools and techniques to be used in acquiring and managing controlled items. The first topics in this subarea are identification of the items to be controlled and the software library.

The third subarea is *Software Configuration Control*, which is the management of changes during the software life cycle. The topics are: first, requesting, evaluating, and approving software changes; second, implementing software changes; and third, deviations and waivers.

The fourth subarea is *Software Configuration Status Accounting*. Its topics are software configuration status information and software configuration status reporting.

The fifth subarea is *Software Configuration Auditing*. It consists of software functional configuration auditing, software physical configuration auditing, and in-process audits of a software baseline.

The last subarea is *Software Release Management and Delivery*, covering software building and software release management.

SOFTWARE ENGINEERING MANAGEMENT (SEE FIGURE 3, COLUMN G)

The Software Engineering Management KA addresses the management and measurement of software engineering. While measurement is an important aspect of all KAs, it is here that the topic of measurement programs is presented. There are six subareas for software engineering management. The first five cover software project management and the sixth describes software measurement programs.

The first subarea is *Initiation and Scope Definition*, which comprises determination and negotiation of requirements, feasibility analysis, and process for the review and revision of requirements.

The second subarea is *Software Project Planning* and includes process planning, determining deliverables, effort, schedule and cost estimation, resource allocation, risk management, quality management, and plan management.

The third subarea is *Software Project Enactment*. The topics here are implementation of plans, supplier contract management, implementation of measurement process, monitor process, control process, and reporting.

The fourth subarea is *Review and Evaluation*, which includes the topics of determining satisfaction of requirements and reviewing and evaluating performance.

The fifth subarea describes *Closure*: determining closure and closure activities.

Finally, the sixth subarea describes *Software Engineering*

Measurement, more specifically, measurement programs. Product and process measures are described in the Software Engineering Process KA. Many of the other KAs also describe measures specific to their KA. The topics of this subarea include establishing and sustaining measurement commitment, planning the measurement process, performing the measurement process, and evaluating measurement.

SOFTWARE ENGINEERING PROCESS (SEE FIGURE 3, COLUMN H)

The Software Engineering Process KA is concerned with the definition, implementation, assessment, measurement, management, change, and improvement of the software engineering process itself. It is divided into four subareas.

The first subarea presents *Process Implementation and Change*. The topics here are process infrastructure, the software process management cycle, models for process implementation and change, and practical considerations.

The second subarea deals with *Process Definition*. It includes the topics of software life cycle models, software life cycle processes, notations for process definitions, process adaptation, and automation.

The third subarea is *Process Assessment*. The topics here include process assessment models and process assessment methods.

The fourth subarea describes *Process and Product Measurements*. The software engineering process covers general product measurement, as well as process measurement in general. Measurements specific to KAs are described in the relevant KA. The topics are process measurement, software product measurement, quality of measurement results, software information models, and process measurement techniques.

SOFTWARE ENGINEERING TOOLS AND METHODS (SEE FIGURE 3, COLUMN I)

The Software Engineering Tools and Methods KA includes both software engineering tools and software engineering methods.

The *Software Engineering Tools* subarea uses the same structure as the Guide itself, with one topic for each of the other nine software engineering KAs. An additional topic is provided: miscellaneous tools issues, such as tool integration techniques, which are potentially applicable to all classes of tools.

The *Software Engineering Methods* subarea is divided into four subsections: heuristic methods dealing with informal approaches, formal methods dealing with mathematically based approaches, and prototyping methods dealing with software development approaches based on various forms of prototyping.

SOFTWARE QUALITY (SEE FIGURE 3, COLUMN J)

The Software Quality KA deals with software quality considerations which transcend the software life cycle processes. Since software quality is a ubiquitous concern in software engineering, it is also considered in many of the other KAs, and the reader will notice pointers to those KAs throughout this KA. The description of this KA covers three subareas.

The first subarea describes the *Software Quality Fundamentals* such as software engineering culture and ethics, the value and costs of quality, models and quality characteristics, and quality improvement.

The second subarea covers *Software Quality Management Processes*. The topics here are software quality assurance, verification and validation, and reviews and audits.

The third and final subarea describes *Practical Considerations* related to software quality. The topics are software quality requirements, defect characterization, software quality management techniques, and software quality measurement.

RELATED DISCIPLINES OF SOFTWARE ENGINEERING (SEE FIGURE 3, COLUMN K)

The last chapter is entitled *Related Disciplines of Software Engineering*. In order to circumscribe software engineering, it is necessary to identify the disciplines with which software engineering shares a common boundary. This chapter identifies, in alphabetical order, these related disciplines. For each related discipline, and using a consensus-based recognized source as found, are identified:

- an informative definition (when feasible);
- a list of KAs.

The related disciplines are:

- Computer engineering
- Computer science
- Management
- Mathematics
- Project management
- Quality management
- Software ergonomics
- Systems engineering

APPENDICES

APPENDIX A. KA DESCRIPTION SPECIFICATIONS

The appendix describes the specifications provided by the editorial team to the associate editors for the content, recommended references, format, and style of the KA descriptions.

APPENDIX B. EVOLUTION OF THE GUIDE

The second appendix describes the project's proposal for the evolution of the Guide. The 2004 Guide is simply the current edition of a guide which will continue evolving to meet the needs of the software engineering community. Planning for evolution is not yet complete, but a tentative outline of the process is provided in this appendix. As of this writing, this process has been endorsed by the project's Industrial Advisory Board and briefed to the Board of Governors of the IEEE Computer Society but is not yet either funded or implemented.

APPENDIX C. ALLOCATION OF STANDARDS TO KAS

The third appendix is an annotated table of the most relevant standards, mostly from the IEEE and the ISO, allocated to the KAs of the SWEBOK Guide.

APPENDIX D. BLOOM RATINGS

As an aid, notably to curriculum developers (and other users), in support of the project's fifth objective, the fourth appendix rates each topic with one of a set of pedagogical categories commonly attributed to Benjamin Bloom. The concept is that educational objectives can be classified into six categories representing increasing depth: knowledge, comprehension, application, analysis, synthesis, and evaluation. Results of this exercise for all KAs can be found in Appendix D. This Appendix must not, however, be viewed as a definitive classification, but much more as a starting point.

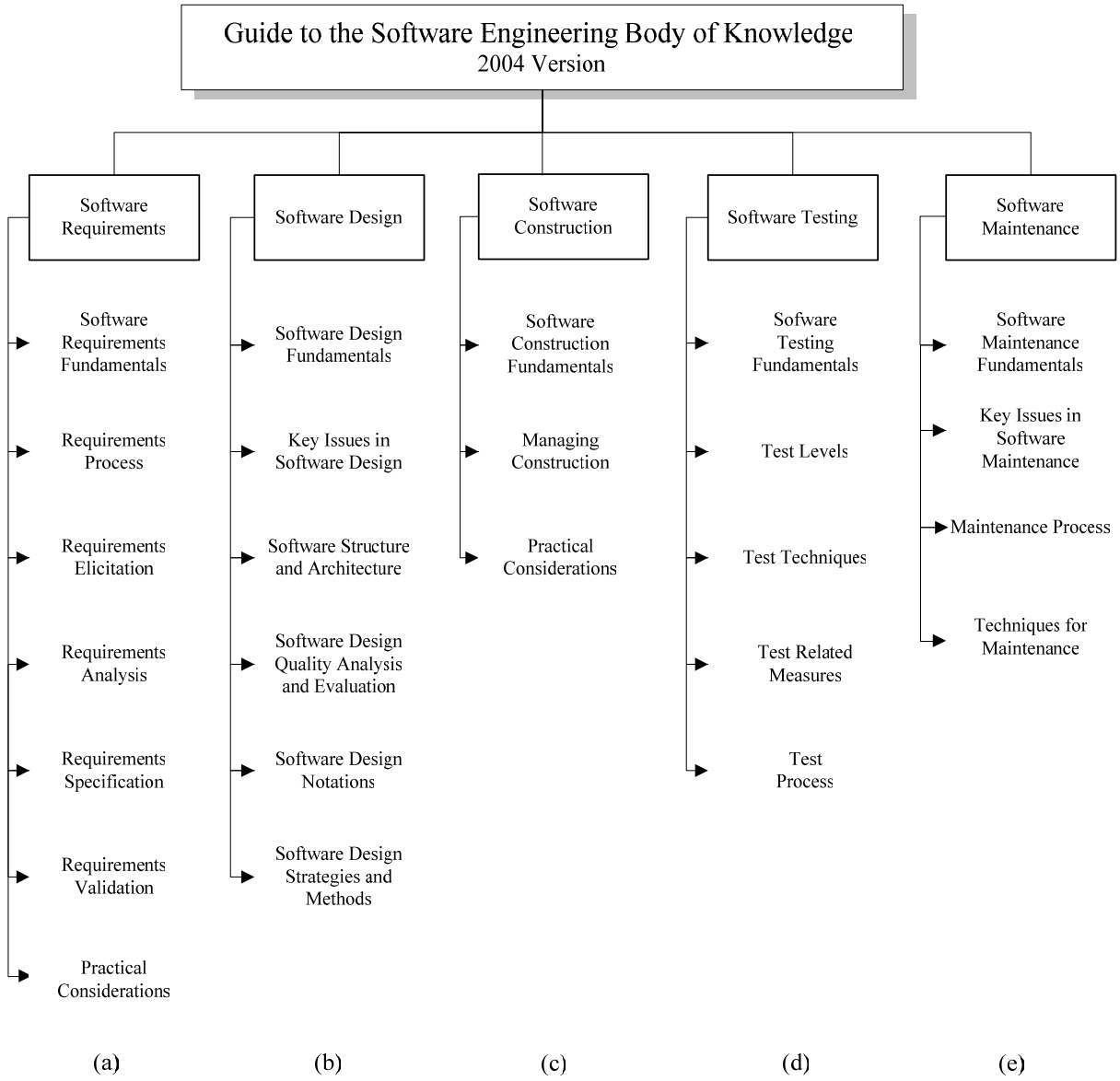


Figure 2 First five KAs

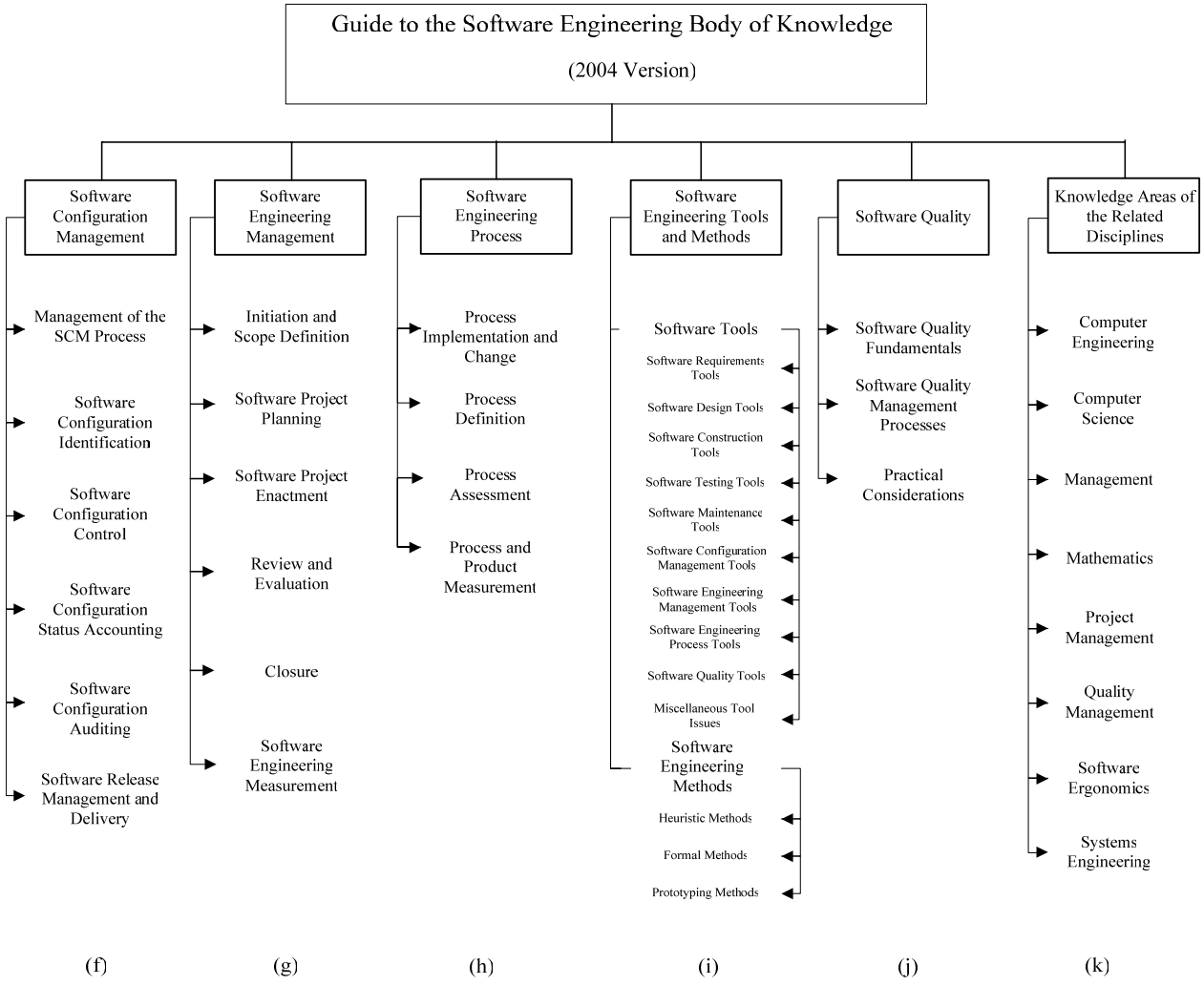


Figure 3 Last six KAs

CHAPTER 2
SOFTWARE REQUIREMENTS

ACRONYMS

DAG	Directed Acyclic Graph
FSM	Functional Size Measurement
INCOSE	International Council on Systems Engineering
SADT	Structured Analysis and Design Technique
UML	Unified Modeling Language

INTRODUCTION

The Software Requirements Knowledge Area (KA) is concerned with the elicitation, analysis, specification, and validation of software requirements. It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly.

Software requirements express the needs and constraints placed on a software product that contribute to the solution of some real-world problem. [Kot00]

The term "requirements engineering" is widely used in the field to denote the systematic handling of requirements. For reasons of consistency, though, this term will not be used in the Guide, as it has been decided that the use of the term "engineering" for activities other than software engineering ones is to be avoided in this edition of the Guide.

For the same reason, "requirements engineer," a term which appears in some of the literature, will not be used either. Instead, the term "software engineer" or, in some specific cases, "requirements specialist" will be used, the latter where the role in question is usually performed by an individual other than a software engineer. This does not imply, however, that a software engineer could not perform the function.

The KA breakdown is broadly compatible with the sections of IEEE 12207 that refer to requirements activities. (IEEE12207.1-96)

A risk inherent in the proposed breakdown is that a waterfall-like process may be inferred. To guard against this, subarea 2 *Requirements process*, is designed to provide a high-level overview of the requirements process by setting out the resources and constraints under which the process operates and which act to configure it.

An alternate decomposition could use a product-based structure (system requirements, software requirements, prototypes, use cases, and so on). The process-based breakdown reflects the fact that the requirements process, if it is to be successful, must be considered as a process involving complex, tightly coupled activities (both sequential and concurrent), rather than as a discrete, one-off activity performed at the outset of a software development project.

The Software Requirements KA is related closely to the Software Design, Software Testing, Software Maintenance, Software Configuration Management, Software Engineering Management, Software Engineering Process, and Software Quality KAs.

BREAKDOWN OF TOPICS FOR SOFTWARE REQUIREMENTS

1. Software Requirements Fundamentals

1.1. Definition of a Software Requirement

At its most basic, a software requirement is a property which must be exhibited in order to solve some problem in the real world. The Guide refers to requirements on "software" because it is concerned with problems to be addressed by software. Hence, a software requirement is a property which must be exhibited by software developed or adapted to solve a particular problem. The problem may be to automate part of a task of someone who will use the software, to support the business processes of the organization that has commissioned the software, to correct shortcomings of existing software, to control a device, and many more. The functioning of users, business processes, and devices is typically complex. By extension, therefore, the requirements on particular software are typically a complex combination of requirements from different people at different levels of an organization and from the environment in which the software will operate.

An essential property of all software requirements is that they be verifiable. It may be difficult or costly to verify certain software requirements. For example, verification of the throughput requirement on the call center may necessitate the development of simulation software. Both the software requirements and software quality personnel must ensure that the requirements can be verified within the available resource constraints.

Requirements have other attributes in addition to the behavioral properties that they express. Common examples include a priority rating to enable trade-offs in the face of finite resources and a status value to enable project progress to be monitored. Typically, software requirements are uniquely identified so that they can be

subjected to software configuration control and managed over the entire software life cycle. [Kot00; Pfl01; Som05; Tha97]

1.2. Product and Process Requirements

A distinction can be drawn between *product* parameters and *process* parameters. Product parameters are requirements on software to be developed (for example, "The software shall verify that a student meets all prerequisites before he or she registers for a course.").

A process parameter is essentially a constraint on the development of the software (for example, "The software shall be written in Ada."). These are sometimes known as process requirements.

Some software requirements generate implicit process requirements. The choice of verification technique is one example. Another might be the use of particularly rigorous analysis techniques (such as formal specification methods) to reduce faults which can lead to inadequate reliability. Process requirements may also be imposed directly by the development organization, their customer, or a third party such as a safety regulator [Kot00; Som97].

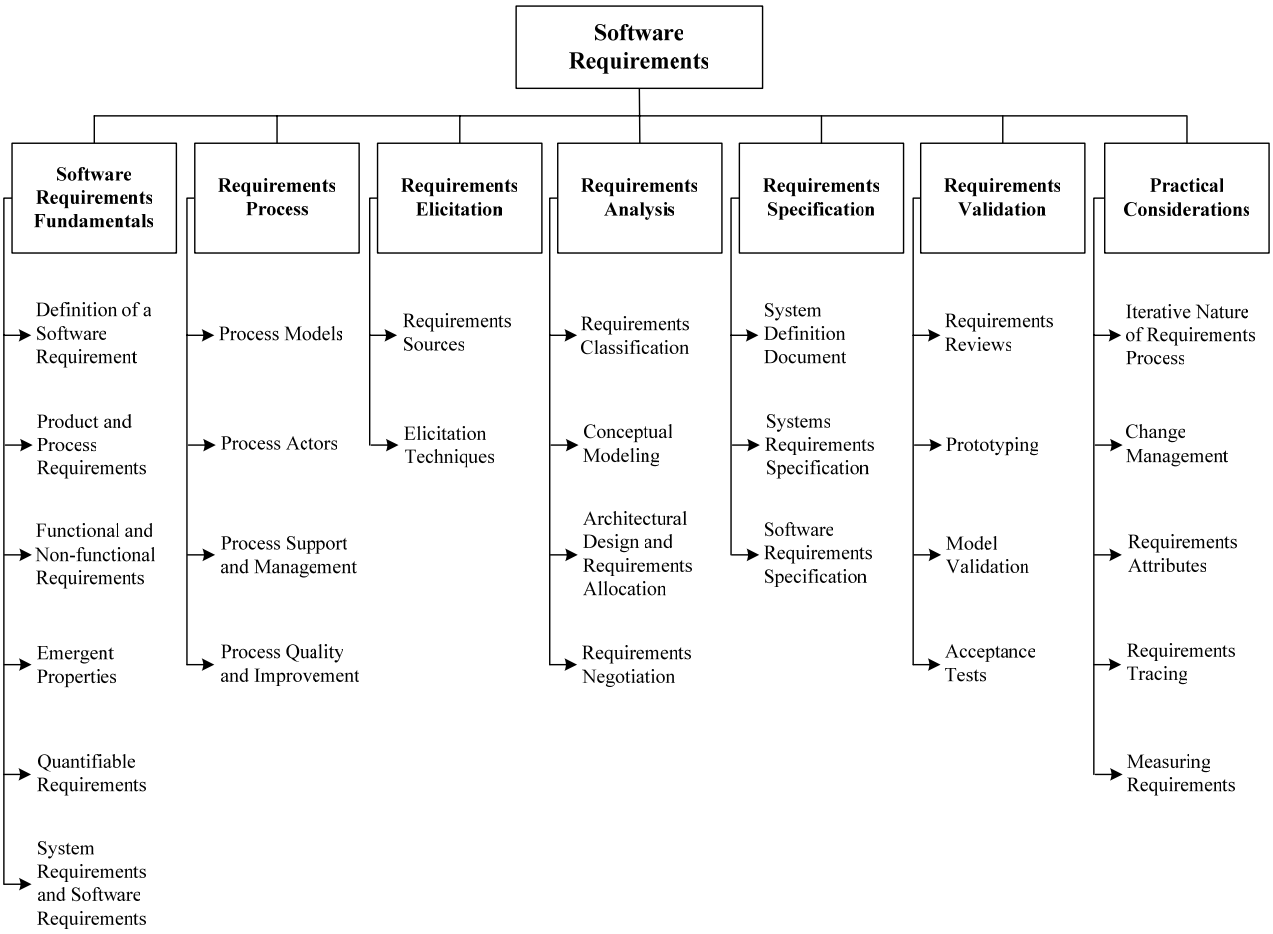


Figure 1 Breakdown of topics for the Software Requirements KA

1.3. Functional and Nonfunctional Requirements

Functional requirements describe the functions that the software is to execute; for example, formatting some text or modulating a signal. They are sometimes known as capabilities.

Nonfunctional requirements are the ones that act to constrain the solution. Nonfunctional requirements are sometimes known as constraints or quality requirements. They can be further classified according to whether they are performance requirements, maintainability requirements, safety requirements, reliability requirements, or one of many other types of software requirements. These topics are also discussed in the Software Quality KA. [Kot00; Som97]

1.4. *Emergent Properties*

Some requirements represent emergent properties of software—that is, requirements which cannot be addressed by a single component, but which depend for their satisfaction on how all the software components interoperate. The throughput requirement for a call center would, for example, depend on how the telephone system, information system, and the operators all interacted under actual operating conditions. Emergent properties are crucially dependent on the system architecture. [Som05]

1.5. *Quantifiable Requirements*

Software requirements should be stated as clearly and as unambiguously as possible, and, where appropriate, quantitatively. It is important to avoid vague and unverifiable requirements which depend for their interpretation on subjective judgment ("the software shall be reliable"; "the software shall be user-friendly"). This is particularly important for nonfunctional requirements. Two examples of quantified requirements are the following: a call center's software must increase the center's throughput by 20%; and a system shall have a probability of generating a fatal error during any hour of operation of less than $1 * 10^{-8}$. The throughput requirement is at a very high level and will need to be used to derive a number of detailed requirements. The reliability requirement will tightly constrain the system architecture. [Dav93; Som05]

1.6. *System Requirements and Software Requirements*

In this topic, system means "*an interacting combination of elements to accomplish a defined objective. These include hardware, software, firmware, people, information, techniques, facilities, services, and other support elements,*" as defined by the International Council on Systems Engineering (INCOSE00).

System requirements are the requirements for the system as a whole. In a system containing software components, *software* requirements are derived from system requirements.

The literature on requirements sometimes calls system requirements "user requirements." The Guide defines "user requirements" in a restricted way as the requirements of the system's customers or end-users. System requirements, by contrast, encompass user requirements, requirements of other stakeholders (such as regulatory authorities), and requirements without an identifiable human source.

2. Requirements Process

This section introduces the software requirements process, orienting the remaining five subareas and showing how the requirements process dovetails with the overall software engineering process. [Dav93; Som05]

2.1. *Process Models*

The objective of this topic is to provide an understanding that the requirements process

- Is not a discrete front-end activity of the software life cycle, but rather a process initiated at the beginning of a project and continuing to be refined throughout the life cycle
- Identifies software requirements as configuration items, and manages them using the same software configuration management practices as other products of the software life cycle processes
- Needs to be adapted to the organization and project context

In particular, the topic is concerned with how the activities of elicitation, analysis, specification, and validation are configured for different types of projects and constraints. The topic also includes activities which provide input into the requirements process, such as marketing and feasibility studies. [Kot00; Rob99; Som97; Som05]

2.2. *Process Actors*

This topic introduces the roles of the people who participate in the requirements process. This process is fundamentally interdisciplinary, and the requirements specialist needs to mediate between the domain of the stakeholder and that of software engineering. There are often many people involved besides the requirements specialist, each of whom has a stake in the software. The stakeholders will vary across projects, but always include users/operators and customers (who need not be the same). [Gog93]

Typical examples of software stakeholders include (but are not restricted to)

- Users: This group comprises those who will operate the software. It is often a heterogeneous group comprising people with different roles and requirements.
- Customers: This group comprises those who have commissioned the software or who represent the software's target market.
- Market analysts: A mass-market product will not have a commissioning customer, so marketing people are often needed to establish what the market needs and to act as proxy customers.
- Regulators: Many application domains such as banking and public transport are regulated. Software in these domains must comply with the requirements of the regulatory authorities.
- Software engineers: These individuals have a legitimate interest in profiting from developing the software by, for example, reusing components in other products. If, in this scenario, a customer of a particular product has specific requirements which compromise the potential for component reuse, the

software engineers must carefully weigh their own stake against those of the customer.

It will not be possible to perfectly satisfy the requirements of every stakeholder, and it is the software engineer's job to negotiate trade-offs which are both acceptable to the principal stakeholders and within budgetary, technical, regulatory, and other constraints. A prerequisite for this is that all the stakeholders be identified, the nature of their "stake" analyzed, and their requirements elicited. [Dav93; Kot00; Rob99; Som97; You01]

2.3. Process Support and Management

This topic introduces the project management resources required and consumed by the requirements process. It establishes the context for the first subarea (*Initiation and scope definition*) of the Software Engineering Management KA. Its principal purpose is to make the link between the process activities identified in 2.1 and the issues of cost, human resources, training, and tools. [Rob99; Som97; You01]

2.4. Process Quality and Improvement

This topic is concerned with the assessment of the quality and improvement of the requirements process. Its purpose is to emphasize the key role the requirements process plays in terms of the cost and timeliness of a software product, and of the customer's satisfaction with it [Som97]. It will help to orient the requirements process with quality standards and process improvement models for software and systems. Process quality and improvement is closely related to both the Software Quality KA and the Software Engineering Process KA. Of particular interest are issues of software quality attributes and measurement, and software process definition. This topic covers

- Requirements process coverage by process improvement standards and models
- Requirements process measures and benchmarking
- Improvement planning and implementation [Kot00; Som97; You01]

3. Requirements Elicitation

[Dav93; Gog93; Lou95; Pfl01]

Requirements elicitation is concerned with where software requirements come from and how the software engineer can collect them. It is the first stage in building an understanding of the problem the software is required to solve. It is fundamentally a human activity, and is where the stakeholders are identified and relationships established between the development team and the customer. It is variously termed "requirements capture," "requirements discovery," and "requirements acquisition."

One of the fundamental tenets of good software engineering is that there be good communication between software users and software engineers. Before development begins, requirements specialists may form the conduit for this communication. They must mediate between the domain of the software users (and other stakeholders) and the technical world of the software engineer.

3.1. *Requirements Sources*

[Dav93; Gog93; Pfl01]

Requirements have many sources in typical software, and it is essential that all potential sources be identified and evaluated for their impact on it. This topic is designed to promote awareness of the various sources of software requirements and of the frameworks for managing them. The main points covered are

- Goals. The term goal (sometimes called "business concern" or "critical success factor") refers to the overall, high-level objectives of the software. Goals provide the motivation for the software, but are often vaguely formulated. Software engineers need to pay particular attention to assessing the value (relative to priority) and cost of goals. A feasibility study is a relatively low-cost way of doing this. [Lou95].
- Domain knowledge. The software engineer needs to acquire, or have available, knowledge about the application domain. This enables them to infer tacit knowledge that the stakeholders do not articulate, assess the trade-offs that will be necessary between conflicting requirements, and, sometimes, to act as a "user" champion.
- Stakeholders (see topic 2.2 *Process actors*). Much software has proved unsatisfactory because it has stressed the requirements of one group of stakeholders at the expense of those of others. Hence, software is delivered which is difficult to use or which subverts the cultural or political structures of the customer organization. The software engineer needs to identify, represent, and manage the "viewpoints" of many different types of stakeholders. [Kot00]
- The operational environment. Requirements will be derived from the environment in which the software will be executed. These may be, for example, timing constraints in real-time software or interoperability constraints in an office environment. These must be actively sought out, because they can greatly affect software feasibility and cost, and restrict design choices. [Tha97]
- The organizational environment. Software is often required to support a business process, the selection of which may be conditioned by the structure, culture, and internal politics of the organization. The software engineer needs to be sensitive to these, since, in general, new software should not force unplanned change on the business process.

3.2. *Elicitation Techniques*

[Dav93; Kot00; Lou95; Pfl01]

Once the requirements sources have been identified, the software engineer can start eliciting requirements from them. This topic concentrates on techniques for getting human stakeholders to articulate their requirements. It is a very difficult area and the software engineer needs to be sensitized to the fact that (for example) users may have difficulty describing their tasks, may leave important information unstated, or may be unwilling or unable to cooperate. It is particularly important to understand that elicitation is not a passive activity, and that, even if cooperative and articulate stakeholders are available, the software engineer has to work hard to elicit the right information. A number of techniques exist for doing this, the principal ones being [Gog93]

- Interviews, a "traditional" means of eliciting requirements. It is important to understand the advantages and limitations of interviews and how they should be conducted.
- Scenarios, a valuable means for providing context to the elicitation of user requirements. They allow the software engineer to provide a framework for questions about user tasks by permitting "what if" and "how is this done" questions to be asked. The most common type of scenario is the use case. There is a link here to topic 4.2 (*Conceptual modeling*) because scenario notations such as use cases and diagrams are common in modeling software.
- Prototypes, a valuable tool for clarifying unclear requirements. They can act in a similar way to scenarios by providing users with a context within which they can better understand what information they need to provide. There is a wide range of prototyping techniques, from paper mock-ups of screen designs to beta-test versions of software products, and a strong overlap of their use for requirements elicitation and the use of prototypes for requirements validation (see topic 6.2 *Prototyping*).
- Facilitated meetings. The purpose of these is to try to achieve a summative effect whereby a group of people can bring more insight into their software requirements than by working individually. They can brainstorm and refine ideas which may be difficult to bring to the surface using interviews. Another advantage is that conflicting requirements surface early on in a way that lets the stakeholders recognize where there is conflict. When it works well, this technique may result in a richer and more consistent set of requirements than might otherwise be achievable. However, meetings need to be handled carefully (hence the need for a facilitator) to prevent a situation from occurring where the critical abilities of the team are eroded by group loyalty, or the requirements reflecting the concerns of a few outspoken (and perhaps senior) people are favored to the detriment of others.
- Observation. The importance of software context within the organizational environment has led to the adaptation of observational techniques for requirements elicitation. Software engineers learn about user tasks by immersing themselves in the

environment and observing how users interact with their software and with each other. These techniques are relatively expensive, but they are instructive because they illustrate that many user tasks and business processes are too subtle and complex for their actors to describe easily.

4. Requirements Analysis

[Som05]

This topic is concerned with the process of analyzing requirements to

- Detect and resolve conflicts between requirements
- Discover the bounds of the software and how it must interact with its environment
- Elaborate system requirements to derive software requirements

The traditional view of requirements analysis has been that it be reduced to conceptual modeling using one of a number of analysis methods such as the Structured Analysis and Design Technique (SADT). While conceptual modeling is important, we include the classification of requirements to help inform trade-offs between requirements (requirements classification) and the process of establishing these trade-offs (requirements negotiation). [Dav93]

Care must be taken to describe requirements precisely enough to enable the requirements to be validated, their implementation to be verified, and their costs to be estimated.

4.1. Requirements Classification

[Dav93; Kot00; Som05]

Requirements can be classified on a number of dimensions. Examples include

- Whether the requirement is functional or nonfunctional (see topic 1.3 *Functional and nonfunctional requirements*).
- Whether the requirement is derived from one or more high-level requirements or an emergent property (see topic 1.4 *Emergent properties*) or is being imposed directly on the software by a stakeholder or some other source.
- Whether the requirement is on the product or the process. Requirements on the process can constrain the choice of contractor, the software engineering process to be adopted, or the standards to be adhered to.
- The requirement priority. In general, the higher the priority, the more essential the requirement is for meeting the overall goals of the software. Often classified on a fixed-point scale such as mandatory, highly desirable, desirable, or optional, the priority often has to be balanced against the cost of development and implementation.
- The scope of the requirement. Scope refers to the extent to which a requirement affects the software and software components. Some requirements, particularly certain nonfunctional ones, have a global scope in that their satisfaction cannot be allocated to a discrete component. Hence, a requirement with global scope may strongly affect the software architecture and the design of many components, whereas one with a narrow scope may offer a number of design choices and have little impact on the satisfaction of other requirements.
- Volatility/stability. Some requirements will change during the life cycle of the software, and even during the development process itself. It is useful if some estimate of the likelihood that a requirement change can be made. For example, in a banking application, requirements for functions to calculate and credit interest to customers' accounts are likely to be more stable than a requirement to support a particular kind of tax-free account. The former reflect a fundamental feature of the banking domain (that accounts can earn interest), while the latter may be rendered obsolete by a change to government legislation. Flagging potentially volatile requirements can help the software engineer establish a design which is more tolerant of change.

Other classifications may be appropriate, depending upon the organization's normal practice and the application itself.

There is a strong overlap between requirements classification and requirements attributes (see topic 7.3 *Requirements attributes*).

4.2. Conceptual Modeling

[Dav93; Kot00; Som05]

The development of models of a real-world problem is key to software requirements analysis. Their purpose is to aid in understanding the problem, rather than to initiate design of the solution. Hence, conceptual models comprise models of entities from the problem domain configured to reflect their real-world relationships and dependencies.

Several kinds of models can be developed. These include data and control flows, state models, event traces, user interactions, object models, data models, and many others. The factors that influence the choice of model include

- The nature of the problem. Some types of software demand that certain aspects be analyzed particularly rigorously. For example, control flow and state models are likely to be more important for real-time software than for management information software, while it would usually be the opposite for data models.
- The expertise of the software engineer. It is often more productive to adopt a modeling notation or

method with which the software engineer has experience.

- The process requirements of the customer. Customers may impose their favored notation or method, or prohibit any with which they are unfamiliar. This factor can conflict with the previous factor.
- The availability of methods and tools. Notations or methods which are poorly supported by training and tools may not achieve widespread acceptance even if they are suited to particular types of problems.

Note that, in almost all cases, it is useful to start by building a model of the software context. The software context provides a connection between the intended software and its external environment. This is crucial to understanding the software's context in its operational environment and to identifying its interfaces with the environment.

The issue of modeling is tightly coupled with that of methods. For practical purposes, a method is a notation (or set of notations) supported by a process which guides the application of the notations. There is little empirical evidence to support claims for the superiority of one notation over another. However, the widespread acceptance of a particular method or notation can lead to beneficial industry-wide pooling of skills and knowledge. This is currently the situation with the UML (Unified Modeling Language). (UML04)

Formal modeling using notations based on discrete mathematics, and which are traceable to logical reasoning, have made an impact in some specialized domains. These may be imposed by customers or standards or may offer compelling advantages to the analysis of certain critical functions or components.

This topic does not seek to "teach" a particular modeling style or notation but rather provides guidance on the purpose and intent of modeling.

Two standards provide notations which may be useful in performing conceptual modeling–IEEE Std 1320.1, IDEF0 for functional modeling; and IEEE Std 1320.2, IDEF1X97 (IDEFObject) for information modeling.

4.3. Architectural Design and Requirements Allocation [Dav93; Som05]

At some point, the architecture of the solution must be derived. Architectural design is the point at which the requirements process overlaps with software or systems design and illustrates how impossible it is to cleanly decouple the two tasks. [Som01] This topic is closely related to the *Software Structure and Architecture* subarea in the Software Design KA. In many cases, the software engineer acts as software architect because the process of analyzing and elaborating the requirements demands that the components that will be responsible for satisfying the requirements be identified. This is requirements allocation–the assignment, to components, of the responsibility for satisfying requirements.

Allocation is important to permit detailed analysis of requirements. Hence, for example, once a set of requirements has been allocated to a component, the individual requirements can be further analyzed to discover further requirements on how the component needs to interact with other components in order to satisfy the allocated requirements. In large projects, allocation stimulates a new round of analysis for each subsystem. As an example, requirements for a particular braking performance for a car (braking distance, safety in poor driving conditions, smoothness of application, pedal pressure required, and so on) may be allocated to the braking hardware (mechanical and hydraulic assemblies) and an anti-lock braking system (ABS). Only when a requirement for an anti-lock braking system has been identified, and the requirements allocated to it, can the capabilities of the ABS, the braking hardware, and emergent properties (such as the car weight) be used to identify the detailed ABS software requirements.

Architectural design is closely identified with conceptual modeling. The mapping from real-world domain entities to software components is not always obvious, so architectural design is identified as a separate topic. The requirements of notations and methods are broadly the same for both conceptual modeling and architectural design.

IEEE Std 1471-2000, *Recommended Practice for Architectural Description of Software Intensive Systems*, suggests a multiple-viewpoint approach to describing the architecture of systems and their software items. (IEEE1471-00)

4.4. Requirements Negotiation

Another term commonly used for this sub-topic is "conflict resolution." This concerns resolving problems with requirements where conflicts occur between two stakeholders requiring mutually incompatible features, between requirements and resources, or between functional and non-functional requirements, for example. [Kot00, Som97] In most cases, it is unwise for the software engineer to make a unilateral decision, and so it becomes necessary to consult with the stakeholder(s) to reach a consensus on an appropriate trade-off. It is often important for contractual reasons that such decisions be traceable back to the customer. We have classified this as a software requirements analysis topic because problems emerge as the result of analysis. However, a strong case can also be made for considering it a requirements validation topic.

5. Requirements Specification

For most engineering professions, the term "specification" refers to the assignment of numerical values or limits to a

product's design goals. (Vin90) Typical physical systems have a relatively small number of such values. Typical software has a large number of requirements, and the emphasis is shared between performing the numerical quantification and managing the complexity of interaction among the large number of requirements. So, in software engineering jargon, "software requirements specification" typically refers to the production of a document, or its electronic equivalent, which can be systematically reviewed, evaluated, and approved. For complex systems, particularly those involving substantial non-software components, as many as three different types of documents are produced: system definition, system requirements, and software requirements. For simple software products, only the third of these is required. All three documents are described here, with the understanding that they may be combined as appropriate. A description of systems engineering can be found in Chapter 12, Related Disciplines of Software Engineering.

5.1. The System Definition Document

This document (sometimes known as the user requirements document or concept of operations) records the system requirements. It defines the high-level system requirements from the domain perspective. Its readership includes representatives of the system users/customers (marketing may play these roles for market-driven software), so its content must be couched in terms of the domain. The document lists the system requirements along with background information about the overall objectives for the system, its target environment and a statement of the constraints, assumptions, and non-functional requirements. It may include conceptual models designed to illustrate the system context, usage scenarios and the principal domain entities, as well as data, information, and workflows. IEEE Std 1362, Concept of Operations Document, provides advice on the preparation and content of such a document. (IEEE1362-98)

5.2. System Requirements Specification

[Dav93; Kot00; Rob99; Tha97]

Developers of systems with substantial software and non-software components, a modern airliner, for example, often separate the description of system requirements from the description of software requirements. In this view, system requirements are specified, the software requirements are derived from the system requirements, and then the requirements for the software components are specified. Strictly speaking, system requirements specification is a systems engineering activity and falls outside the scope of this Guide. IEEE Std 1233 is a guide for developing system requirements. (IEEE1233-98)

5.3. Software Requirements Specification

[Kot00; Rob99]

Software requirements specification establishes the basis for agreement between customers and contractors or suppliers (in market-driven projects, these roles may be played by the marketing and development divisions) on what the software product is to do, as well as what it is not expected to do. For non-technical readers, the software requirements specification document is often accompanied by a software requirements definition document.

Software requirements specification permits a rigorous assessment of requirements before design can begin and reduces later redesign. It should also provide a realistic basis for estimating product costs, risks, and schedules.

Organizations can also use a software requirements specification document to develop their own validation and verification plans more productively.

Software requirements specification provides an informed basis for transferring a software product to new users or new machines. Finally, it can provide a basis for software enhancement.

Software requirements are often written in natural language, but, in software requirements specification, this may be supplemented by formal or semi-formal descriptions. Selection of appropriate notations permits particular requirements and aspects of the software architecture to be described more precisely and concisely than natural language. The general rule is that notations should be used which allow the requirements to be described as precisely as possible. This is particularly crucial for safety-critical and certain other types of dependable software. However, the choice of notation is often constrained by the training, skills and preferences of the document's authors and readers.

A number of quality indicators have been developed which can be used to relate the quality of software requirements specification to other project variables such as cost, acceptance, performance, schedule, reproducibility, etc. Quality indicators for individual software requirements specification statements include imperatives, directives, weak phrases, options, and continuances. Indicators for the entire software requirements specification document include size, readability, specification, depth, and text structure. [Dav93; Tha97] (Ros98)

IEEE has a standard, IEEE Std 830 [IEEE830-98], for the production and content of the software requirements specification. Also, IEEE 1465 (similar to ISO/IEC 12119) is a standard treating quality requirements in software packages. (IEEE1465-98)

6. Requirements validation

[Dav93]

The requirements documents may be subject to validation and verification procedures. The requirements may be validated to ensure that the software engineer has understood the requirements, and it is also important to verify that a requirements document conforms to company

standards, and that it is understandable, consistent, and complete. Formal notations offer the important advantage of permitting the last two properties to be proven (in a restricted sense, at least). Different stakeholders, including representatives of the customer and developer, should review the document(s). Requirements documents are subject to the same software configuration management practices as the other deliverables of the software life cycle processes. (Bry94, Ros98)

It is normal to explicitly schedule one or more points in the requirements process where the requirements are validated. The aim is to pick up any problems before resources are committed to addressing the requirements. Requirements validation is concerned with the process of examining the requirements document to ensure that it defines the right software (that is, the software that the users expect). [Kot00]

6.1. Requirements Reviews

[Kot00; Som05; Tha97]

Perhaps the most common means of validation is by inspection or reviews of the requirements document(s). A group of reviewers is assigned a brief to look for errors, mistaken assumptions, lack of clarity, and deviation from standard practice. The composition of the group that conducts the review is important (at least one representative of the customer should be included for a customer-driven project, for example), and it may help to provide guidance on what to look for in the form of checklists.

Reviews may be constituted on completion of the system definition document, the system specification document, the software requirements specification document, the baseline specification for a new release, or at any other step in the process. IEEE Std 1028 provides guidance on conducting such reviews. (IEEE1028-97) Reviews are also covered in the Software Quality KA, topic 2.3 *Reviews and Audits*.

6.2. Prototyping

[Dav93; Kot00; Som05; Tha97]

Prototyping is commonly a means for validating the software engineer's interpretation of the software requirements, as well as for eliciting new requirements. As with elicitation, there is a range of prototyping techniques and a number of points in the process where prototype validation may be appropriate. The advantage of prototypes is that they can make it easier to interpret the software engineer's assumptions and, where needed, give useful feedback on why they are wrong. For example, the dynamic behavior of a user interface can be better understood through an animated prototype than through textual description or graphical models. There are also disadvantages, however. These include the danger of users' attention being distracted from the core underlying functionality by cosmetic issues or quality problems with the prototype. For this reason, several people recommend prototypes which avoid software, such as flip-chart-based mockups. Prototypes may be costly to develop. However, if they avoid the wastage of resources caused by trying to satisfy erroneous requirements, their cost can be more easily justified.

6.3. Model Validation

[Dav93; Kot00; Tha97]

It is typically necessary to validate the quality of the models developed during analysis. For example, in object models, it is useful to perform a static analysis to verify that communication paths exist between objects which, in the stakeholders' domain, exchange data. If formal specification notations are used, it is possible to use formal reasoning to prove specification properties.

6.4. Acceptance Tests

[Dav93]

An essential property of a software requirement is that it should be possible to validate that the finished product satisfies it. Requirements which cannot be validated are really just "wishes." An important task is therefore planning how to verify each requirement. In most cases, designing acceptance tests does this.

Identifying and designing acceptance tests may be difficult for non-functional requirements (see topic 1.3 *Functional and Non-functional Requirements*). To be validated, they must first be analyzed to the point where they can be expressed quantitatively.

Additional information can be found in the Software Testing KA, sub-topic 2.2.4 *Conformance testing*.

7. Practical Considerations

The first level of decomposition of subareas presented in this KA may seem to describe a linear sequence of activities. This is a simplified view of the process. [Dav93]

The requirements process spans the whole software life cycle. Change management and the maintenance of the requirements in a state which accurately mirrors the software to be built, or that has been built, are key to the success of the software engineering process. [Kot00; Lou95]

Not every organization has a culture of documenting and managing requirements. It is frequent in dynamic start-up companies, driven by a strong "product vision" and limited resources, to view requirements documentation as an unnecessary overhead. Most often, however, as these companies expand, as their customer base grows, and as their product starts to evolve, they discover that they need to recover the requirements that motivated product features in order to assess the impact of proposed changes. Hence, requirements documentation and change management are key to the success of any requirements process.

7.1. *Iterative Nature of the Requirements Process*
[Kot00; You01]

There is general pressure in the software industry for ever shorter development cycles, and this is particularly pronounced in highly competitive market-driven sectors. Moreover, most projects are constrained in some way by their environment, and many are upgrades to, or revisions of, existing software where the architecture is a given. In practice, therefore, it is almost always impractical to implement the requirements process as a linear, deterministic process in which software requirements are elicited from the stakeholders, baselined, allocated, and handed over to the software development team. It is certainly a myth that the requirements for large software projects are ever perfectly understood or perfectly specified. [Som97]

Instead, requirements typically iterate towards a level of quality and detail which is sufficient to permit design and procurement decisions to be made. In some projects, this may result in the requirements being baselined before all their properties are fully understood. This risks expensive rework if problems emerge late in the software engineering process. However, software engineers are necessarily constrained by project management plans and must therefore take steps to ensure that the "quality" of the requirements is as high as possible given the available resources. They should, for example, make explicit any assumptions which underpin the requirements, as well as any known problems.

In almost all cases, requirements understanding continues to evolve as design and development proceeds. This often leads to the revision of requirements late in the life cycle. Perhaps the most crucial point in understanding requirements engineering is that a significant proportion of the requirements *will* change. This is sometimes due to errors in the analysis, but it is frequently an inevitable consequence of change in the "environment": for example, the customer's operating or business environment, or the market into which software must sell. Whatever the cause, it is important to recognize the inevitability of change and take steps to mitigate its effects. Change has to be managed by ensuring that proposed changes go through a defined review and approval process, and, by applying careful requirements tracing, impact analysis, and software configuration management (see the Software Configuration Management KA). Hence, the requirements process is not merely a front-end task in software development, but spans the whole software life cycle. In a typical project, the software requirements activities evolve over time from elicitation to change management.

7.2. *Change Management*
[Kot00]

Change management is central to the management of requirements. This topic describes the role of change management, the procedures that need to be in place, and the analysis that should be applied to proposed changes. It has strong links to the Software Configuration Management KA.

7.3. *Requirements Attributes*
[Kot00]

Requirements should consist not only of a specification of what is required, but also of ancillary information which helps manage and interpret the requirements. This should include the various classification dimensions of the requirement (see topic 4.1 *Requirements Classification*) and the verification method or acceptance test plan. It may also include additional information such as a summary rationale for each requirement, the source of each requirement, and a change history. The most important requirements attribute, however, is an identifier which allows the requirements to be uniquely and unambiguously identified.

7.4. *Requirements Tracing*
[Kot00]

Requirements tracing is concerned with recovering the source of requirements and predicting the effects of requirements. Tracing is fundamental to performing impact analysis when requirements change. A requirement should be traceable backwards to the requirements and stakeholders which motivated it (from a software requirement back to the system requirement(s) that it helps satisfy, for example). Conversely, a requirement should be traceable forwards into the requirements and design entities that satisfy it (for example, from a system requirement into the software requirements that have been elaborated from it, and on into the code modules that implement it).

The requirements tracing for a typical project will form a complex directed acyclic graph (DAG) of requirements.

7.5. *Measuring Requirements*
As a practical matter, it is typically useful to have some concept of the "volume" of the requirements for a particular software product. This number is useful in evaluating the "size" of a change in requirements, in estimating the cost of a development or maintenance task, or simply for use as the denominator in other measurements. Functional Size Measurement (FSM) is a technique for evaluating the size of a body of functional requirements. IEEE Std 14143.1 defines the concept of FSM. [IEEE14143.1-00] Standards from ISO/IEC and other sources describe particular FSM methods.

Additional information on size measurement and standards will be found in the Software Engineering Process KA.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	[Dav93]	[Gog93]	[IEEE830-98]	[IEEE14143.1-00]	[Kot00]	[Lou95]	[Pfl01]	[Rob99]	[Som97]	[Som05]	[Tha97]	[You01]
1. Software Requirements Fundamentals												
1.1 Definition of a Software Requirement					*		*			c5	c1	
1.2 Product and Process Requirements					*				c1			
1.3 Functional and Non-functional Requirements					*				c1			
1.4 Emergent Properties										c2		
1.5 Quantifiable Requirements	c3s4									c6		
1.6 System Requirements and Software Requirements												
2. Requirements Process	*									c5		
2.1 Process Models					c2s1			*	c2	c3		
2.2 Process Actors	c2	*			c2s2			c3	c2			c3
2.3 Process Support and Management								c3	c2			c2,c7
2.4 Process Quality and Improvement					c2s4				c2			c5
3. Requirements Elicitation	*	*				*	*					
3.1 Requirements Sources	c2	*			c3s1	*	*				c1	
3.2 Elicitation Techniques	c2	*			c3s2	*	*					
4. Requirements Analysis	*									c6		
4.1 Requirements Classification	*				c8s1					c6		
4.2 Conceptual Modeling	*				*					c7		
4.3 Architectural Design and Requirements Allocation	*									c10		
4.4 Requirements Negotiation					c3s4				*			
5. Requirements Specification												
5.1 The System Definition Document												
5.2 The System Requirements Specification	*				*			c9			c3	
5.3 The Software Requirements Specification	*		*		*			c9			c3	
6. Requirements Validation	*				*							
6.1 Requirements Reviews					c4s1					c6	c5	
6.2 Prototyping	c6				c4s2					c8	c6	
6.3 Model Validation	*				c4s3						c5	
6.4 Acceptance Tests	*											
7. Practical Considerations	*				*	*						
7.1 Iterative Nature of the Requirements Process					c5s1				c2			c6
7.2 Change Management					c5s3							
7.3 Requirement Attributes					c5s2							
7.4 Requirements Tracing					c5s4							
7.5 Measuring Requirements				*								

RECOMMENDED REFERENCES FOR SOFTWARE REQUIREMENTS

[Dav93] A.M. Davis, *Software Requirements: Objects, Functions and States*, Prentice Hall, 1993.

[Gog93] J. Goguen and C. Linde, "Techniques for Requirements Elicitation," presented at International Symposium on Requirements Engineering, 1993.

[IEEE830-98] IEEE Std 830-1998, *IEEE Recommended Practice for Software Requirements Specifications*, IEEE, 1998.

(IEEE14143.1-00) IEEE Std 14143.1-2000//ISO/IEC14143-1:1998, *Information Technology—Software Measurement—Functional Size Measurement—Part 1: Definitions of Concepts*, IEEE, 2000.

[Kot00] G. Kotonya and I. Sommerville, *Requirements Engineering: Processes and Techniques*, John Wiley & Sons, 2000.

[Lou95] P. Loucopulos and V. Karakostas, *Systems Requirements Engineering*, McGraw-Hill, 1995.

[Pfl01] S.L. Pfleeger, "Software Engineering: Theory and Practice," second ed., Prentice Hall, 2001, Chap. 4.

[Rob99] S. Robertson and J. Robertson, *Mastering the Requirements Process*, Addison-Wesley, 1999.

[Som97] I. Sommerville and P. Sawyer, *Requirements Engineering: A Good Practice Guide*, John Wiley & Sons, 1997, Chap. 1-2.

[Som05] I. Sommerville, *Software Engineering*, seventh ed., Addison-Wesley, 2005.

[Tha97] R.H. Thayer and M. Dorfman, eds., *Software Requirements Engineering*, IEEE Computer Society Press, 1997, pp. 176-205, 389-404.

[You01] R.R. You, *Effective Requirements Practices*, Addison-Wesley, 2001.

APPENDIX A. LIST OF FURTHER READINGS

(Ale02) I. Alexander and R. Stevens, *Writing Better Requirements*, Addison-Wesley, 2002.

(Ard97) M. Ardis, "Formal Methods for Telecommunication System Requirements: A Survey of Standardized Languages," *Annals of Software Engineering*, vol. 3, 1997.

(Ber97) V. Berzins et al., "A Requirements Evolution Model for Computer Aided Prototyping," presented at Ninth IEEE International Conference on Software Engineering and Knowledge Engineering, Knowledge Systems Institute, 1997.

(Bey95) H. Beyer and K. Holtzblatt, "Apprenticing with the Customer," *Communications of the ACM*, vol. 38, iss. 5, May 1995, pp. 45-52.

(Bru95) G. Bruno and R. Agarwal, "Validating Software Requirements Using Operational Models," presented at Second Symposium on Software Quality Techniques and Acquisition Criteria, 1995.

(Bry94) E. Bryne, "IEEE Standard 830: Recommended Practice for Software Requirements Specification," presented at IEEE International Conference on Requirements Engineering, 1994.

(Buc94) G. Bucci et al., "An Object-Oriented Dual Language for Specifying Reactive Systems," presented at IEEE International Conference on Requirements Engineering, 1994.

(Bus95) D. Bustard and P. Lundy, "Enhancing Soft Systems Analysis with Formal Modeling," presented at Second International Symposium on Requirements Engineering, 1995.

(Che94) M. Chechik and J. Gannon, "Automated Verification of Requirements Implementation," presented at Proceedings of the International Symposium on Software Testing and Analysis, special issue, 1994.

(Chu95) L. Chung and B. Nixon, "Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approach," presented at Seventeenth IEEE International Conference on Software Engineering, 1995.

(Cia97) P. Ciancarini et al., "Engineering Formal Requirements: An Analysis and Testing Method for Z Documents," *Annals of Software Engineering*, vol. 3, 1997.

(Cre94) R. Crespo, "We Need to Identify the Requirements of the Statements of Non-Functional Requirements," presented at International Workshop on Requirements Engineering: Foundations of Software Quality, 1994.

(Cur94) P. Curran et al., "BORIS-R Specification of the Requirements of a Large-Scale Software Intensive System," presented at Requirements Elicitation for Software-Based Systems, 1994.

(Dar97) R. Darimont and J. Souquieres, "Reusing Operational Requirements: A Process-Oriented Approach," presented at IEEE International Symposium on Requirements Engineering, 1997.

(Dav94) A. Davis and P. Hsia, "Giving Voice to Requirements Engineering: Guest Editors' Introduction," *IEEE Software*, vol. 11, iss. 2, March 1994, pp. 12-16.

(Def94) J. DeFoe, "Requirements Engineering Technology in Industrial Education," presented at IEEE International Conference on Requirements Engineering, 1994.

(Dem97) E. Demirors, "A Blackboard Framework for Supporting Teams in Software Development," presented at Ninth IEEE International Conference on Software Engineering and Knowledge Engineering, Knowledge Systems Institute, 1997.

(Die95) M. Diepstraten, "Command and Control System Requirements Analysis and System Requirements Specification for a Tactical System," presented at First IEEE International Conference on Engineering of Complex Computer Systems, 1995.

(Dob94) J. Dobson and R. Strens, "Organizational Requirements Definition for Information Technology," presented at IEEE International Conference on Requirements Engineering, 1994.

(Duf95) D. Duffy et al., "A Framework for Requirements Analysis Using Automated Reasoning," presented at Seventh International Conference on Advanced Information Systems Engineering, 1995.

(Eas95) S. Easterbrook and B. Nuseibeh, "Managing Inconsistencies in an Evolving Specification," presented at Second International Symposium on Requirements Engineering, 1995.

(Edw95) M. Edwards et al., "RECAP: A Requirements Elicitation, Capture, and Analysis Process Prototype Tool for Large Complex Systems," presented at First IEEE International Conference on Engineering of Complex Computer Systems, 1995.

(ElE95) K. El-Emam and N. Madhavji, "Requirements Engineering Practices in Information Systems Development: A Multiple Case Study," presented at Second International Symposium on Requirements Engineering, 1995.

(Fai97) R. Fairley and R. Thayer, "The Concept of Operations: The Bridge from Operational Requirements to Technical Specifications," *Annals of Software Engineering*, vol. 3, 1997.

(Fic95) S. Fickas and M. Feather, "Requirements Monitoring in Dynamic Environments," presented at Second International Symposium on Requirements Engineering, 1995.

(Fie95) R. Fields et al., "A Task-Centered Approach to Analyzing Human Error Tolerance Requirements," presented at Second International Symposium on Requirements Engineering, 1995.

(Gha94) J. Ghajar-Dowlatshahi and A. Varnekar, "Rapid Prototyping in Requirements Specification Phase of Software Systems," presented at Fourth International Symposium on Systems Engineering, National Council on Systems Engineering, 1994.

(Gib95) M. Gibson, "Domain Knowledge Reuse During

Requirements Engineering," presented at Seventh International Conference on Advanced Information Systems Engineering (CAiSE '95), 1995.
(Gol94) L. Goldin and D. Berry, "AbstFinder: A Prototype Abstraction Finder for Natural Language Text for Use in Requirements Elicitation: Design, Methodology and Evaluation," presented at IEEE International Conference on Requirements Engineering, 1994.
(Got97) O. Gotel and A. Finkelstein, "Extending Requirements Traceability: Lessons Learned from an Industrial Case Study," presented at IEEE International Symposium on Requirements Engineering, 1997.
(Hei96) M. Heimdahl, "Errors Introduced during the TACS II Requirements Specification Effort: A Retrospective Case Study," presented at Eighteenth IEEE International Conference on Software Engineering, 1996.
(Hei96a) C. Heitmeyer et al., "Automated Consistency Checking Requirements Specifications," *ACM Transactions on Software Engineering and Methodology*, vol. 5, iss. 3, July 1996, pp. 231-261.
(Hol95) K. Holtzblatt and H. Beyer, "Requirements Gathering: The Human Factor," *Communications of the ACM*, vol. 38, iss. 5, May 1995, pp. 31-32.
(Hud96) E. Hudlicka, "Requirements Elicitation with Indirect Knowledge Elicitation Techniques: Comparison of Three Methods," presented at Second IEEE International Conference on Requirements Engineering, 1996.
(Hug94) K. Hughes et al., "A Taxonomy for Requirements Analysis Techniques," presented at IEEE International Conference on Requirements Engineering, 1994.
(Hug95) J. Hughes et al., "Presenting Ethnography in the Requirements Process," presented at Second IEEE International Symposium on Requirements Engineering, 1995.
(Hut94) A.T.F. Hutt, ed., *Object Analysis and Design - Comparison of Methods. Object Analysis and Design - Description of Methods*, John Wiley & Sons, 1994.
(INCOSE00) INCOSE, *How To: Guide for all Engineers, Version 2*, International Council on Systems Engineering, 2000.
(Jac95) M. Jackson, *Software Requirements and Specifications*, Addison-Wesley, 1995.
(Jac97) M. Jackson, "The Meaning of Requirements," *Annals of Software Engineering*, vol. 3, 1997.
(Jon96) S. Jones and C. Britton, "Early Elicitation and Definition of Requirements for an Interactive Multimedia Information System," presented at Second IEEE International Conference on Requirements Engineering, 1996.
(Kir96) T. Kirner and A. Davis, "Nonfunctional Requirements for Real-Time Systems," *Advances in Computers*, 1996.
(Kle97) M. Klein, "Handling Exceptions in Collaborative Requirements Acquisition," presented at IEEE International Symposium on Requirements Engineering, 1997.
(Kos97) R. Kosman, "A Two-Step Methodology to Reduce Requirements Defects," *Annals of Software Engineering*, vol. 3, 1997.
(Kro95) J. Krogstie et al., "Towards a Deeper Understanding of Quality in Requirements Engineering," presented at Seventh International Conference on Advanced Information Systems Engineering (CAiSE '95), 1995.
(Lal95) V. Lalioti and B. Theodoulidis, "Visual Scenarios for Validation of Requirements Specification," presented at Seventh International Conference on Software Engineering and Knowledge Engineering, Knowledge Systems Institute, 1995.
(Lam95) A. v. Lamsweerde et al., "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt," presented at Second International Symposium on Requirements Engineering, 1995.
(Lei97) J. Leite et al., "Enhancing a Requirements Baseline with Scenarios," presented at IEEE International Symposium on Requirements Engineering, 1997.
(Ler97) F. Lerch et al.., "Using Simulation-Based Experiments for Software Requirements Engineering," *Annals of Software Engineering*, vol. 3, 1997.
(Lev94) N. Leveson et al., "Requirements Specification for Process-Control Systems," *IEEE Transactions on Software Engineering*, vol. 20, iss. 9, September 1994, pp. 684-707.
(Lut96a) R. Lutz and R. Woodhouse, "Contributions of SFMEA to Requirements Analysis," presented at Second IEEE International Conference on Requirements Engineering, 1996.
(Lut97) R. Lutz and R. Woodhouse, "Requirements Analysis Using Forward and Backward Search," *Annals of Software Engineering*, vol. 3, 1997.
(Mac96) L. Macaulay, *Requirements Engineering*, Springer-Verlag, 1996.
(Mai95) N. Maiden et al., "Computational Mechanisms for Distributed Requirements Engineering," presented at Seventh International Conference on Software Engineering and Knowledge Engineering, Knowledge Systems Institute, 1995.
(Mar94) B. Mar, "Requirements for Development of Software Requirements," presented at Fourth International Symposium on Systems Engineering, 1994.
(Mas97) P. Massonet and A. v. Lamsweerde, "Analogical Reuse of Requirements Frameworks," presented at IEEE International Symposium on Requirements Engineering, 1997.
(McF95) I. McFarland and I. Reilly, "Requirements Traceability in an Integrated Development Environment," presented at Second International Symposium on Requirements Engineering, 1995.
(Mea94) N. Mead, "The Role of Software Architecture in Requirements Engineering," presented at IEEE

International Conference on Requirements Engineering, 1994.
(Mos95) D. Mostert and S. v. Solms, "A Technique to Include Computer Security, Safety, and Resilience Requirements as Part of the Requirements Specification," *Journal of Systems and Software*, vol. 31, iss. 1, October 1995, pp. 45-53.
(Myl95) J. Mylopoulos et al., "Multiple Viewpoints Analysis of Software Specification Process," *IEEE Transactions on Software Engineering*, 1995.
(Nis92) K. Nishimura and S. Honiden, "Representing and Using Non-Functional Requirements: A Process-Oriented Approach," *IEEE Transactions on Software Engineering*, December 1992.
(Nis97) H. Nissen et al., "View-Directed Requirements Engineering: A Framework and Metamodel," presented at Ninth IEEE International Conference on Software Engineering and Knowledge Engineering, Knowledge Systems Institute, 1997.
(OBr96) L. O'Brien, "From Use Case to Database: Implementing a Requirements Tracking System," *Software Development*, vol. 4, iss. 2, February 1996, pp. 43-47.
(UML04) Object Management Group, *Unified Modeling Language*, www.uml.org, 2004.
(Opd94) A. Opdahl, "Requirements Engineering for Software Performance," presented at International Workshop on Requirements Engineering: Foundations of Software Quality, 1994.
(Pin96) F. Pinheiro and J. Goguen, "An Object-Oriented Tool for Tracing Requirements," *IEEE Software*, vol. 13, iss. 2, March 1996, pp. 52-64.
(Pla96) G. Playle and C. Schroeder, "Software Requirements Elicitation: Problems, Tools, and Techniques," *Crosstalk: The Journal of Defense Software Engineering*, vol. 9, iss. 12, December 1996, pp. 19-24.
(Poh94) K. Pohl et al., "Applying AI Techniques to Requirements Engineering: The NATURE Prototype," presented at IEEE Workshop on Research Issues in the Intersection Between Software Engineering and Artificial Intelligence, 1994.
(Por95) A. Porter et al., "Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment," *IEEE Transactions on Software Engineering*, vol. 21, iss. 6, June 1995, pp. 563-575.
(Pot95) C. Potts et al., "An Evaluation of Inquiry-Based Requirements Analysis for an Internet Server," presented at Second International Symposium on Requirements Engineering, 1995.
(Pot97) C. Potts and I. Hsi, "Abstraction and Context in Requirements Engineering: Toward a Synthesis," *Annals of Software Engineering*, vol. 3, 1997.
(Pot97a) C. Potts and W. Newstetter, "Naturalistic Inquiry and Requirements Engineering: Reconciling Their Theoretical Foundations," presented at IEEE International Symposium on Requirements Engineering, 1997.
(Ram95) B. Ramesh et al., "Implementing Requirements Traceability: A Case Study," presented at Second International Symposium on Requirements Engineering, 1995.
(Reg95) B. Regnell et al., "Improving the Use Case Driven Approach to Requirements Engineering," presented at Second IEEE International Symposium on Requirements Engineering, 1995.
(Reu94) H. Reubenstein, "The Role of Software Architecture in Software Requirements Engineering," presented at IEEE International Conference on Requirements Engineering, 1994.
(Rob94) J. Robertson and S. Robertson, *Complete Systems Analysis*, Vol. 1 and 2, Prentice Hall, 1994.
(Rob94a) W. Robinson and S. Fickas, "Supporting Multi-Perspective Requirements Engineering," presented at IEEE International Conference on Requirements Engineering, 1994.
(Ros98) L. Rosenberg, T.F. Hammer, and L.L. Huffman, "Requirements, testing and metrics," presented at 15th Annual Pacific Northwest Software Quality Conference, 1998.
(Sch94) W. Schoening, "The Next Big Step in Systems Engineering Tools: Integrating Automated Requirements Tools with Computer Simulated Synthesis and Test," presented at Fourth International Symposium on Systems Engineering, 1994.
(She94) M. Shekaran, "The Role of Software Architecture in Requirements Engineering," presented at IEEE International Conference on Requirements Engineering, 1994.
(Sid97) J. Siddiqi et al., "Towards Quality Requirements Via Animated Formal Specifications," *Annals of Software Engineering*, vol. 3, 1997.
(Span97) G. Spanoudakis and A. Finkelstein, "Reconciling Requirements: A Method for Managing Interference, Inconsistency, and Conflict," *Annals of Software Engineering*, vol. 3, 1997.
(Ste94) R. Stevens, "Structured Requirements," presented at Fourth International Symposium on Systems Engineering, 1994.
(Vin90) W.G. Vincenti, *What Engineers Know and How They Know It - Analytical Studies form Aeronautical History*, John Hopkins University Press, 1990.
(Wei03) K. Weigers, *Software Requirements*, second ed., Microsoft Press, 2003.
(Whi95) S. White and M. Edwards, "A Requirements Taxonomy for Specifying Complex Systems," presented at First IEEE International Conference on Engineering of Complex Computer Systems, 1995.
(Wil99) B. Wiley, *Essential System Requirements: A Practical Guide to Event-Driven Methods*, Addison-Wesley, 1999.
(Wyd96) T. Wyder, "Capturing Requirements With Use Cases," *Software Development*, vol. 4, iss. 2, February 1996, pp. 36-40.
(Yen97) J. Yen and W. Tiao, "A Systematic Tradeoff Analysis for Conflicting Imprecise Requirements,"

presented at IEEE International Symposium on Requirements Engineering, 1997.
(Yu97) E. Yu, "Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering," presented at IEEE International Symposium on Requirements Engineering, 1997.
(Zav96) P. Zave and M. Jackson, "Where Do Operations Come From? A Multiparadigm Specification Technique," *IEEE Transactions on Software Engineering,*, vol. 22, iss. 7, July 1996, pp. 508-528.

APPENDIX B. LIST OF STANDARDS

(IEEE830-98) IEEE Std 830-1998, *IEEE Recommended Practice for Software Requirements Specifications*, IEEE, 1998.

(IEEE1028-97) IEEE Std 1028-1997 (R2002), *IEEE Standard for Software Reviews*, IEEE, 1997.

(IEEE1233-98) IEEE Std 1233-1998, *IEEE Guide for Developing System Requirements Specifications*, 1998.

(IEEE1320.1-98) IEEE Std 1320.1-1998, *IEEE Standard for Functional Modeling Language-Syntax and Semantics for IDEF0*, IEEE, 1998.

(IEEE1320.2-98) IEEE Std 1320.2-1998, *IEEE Standard for Conceptual Modeling Language-Syntax and Semantics for IDEFIX97 (IDEFObjetct)*, IEEE, 1998.

(IEEE1362-98) IEEE Std 1362-1998, *IEEE Guide for Information Technology-System Definition-Concept of Operations (ConOps) Document*, IEEE, 1998.

(IEEE1465-98) IEEE Std 1465-1998//ISO/IEC12119:1994, *IEEE Standard Adoption of International Standard ISO/IEC12119:1994(E), Information Technology-Software Packages-Quality requirements and testing*, IEEE, 1998.

(IEEEP1471-00) IEEE Std 1471-2000, *IEEE Recommended Practice for Architectural Descriptionos Software Intensive Systems*, Architecture Working Group of the Software Engineering Standards Committee, 2000.

(IEEE12207.0-96) IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.

(IEEE14143.1-00) IEEE Std 14143.1-2000//ISO/IEC14143-1:1998, *Information Technology-Software Measurement-Functional Size Measurement-Part 1: Definitions of Concepts*, IEEE, 2000.

CHAPTER 3
SOFTWARE DESIGN

ACRONYMS

ADL	Architecture Description Languages
CRC	Class Responsibility Collaborator card
ERD	Entity-Relationship Diagram
IDL	Interface Description Language
DFD	Data Flow Diagram
PDL	Pseudo-Code and Program Design Language
CBD	Component-Based design

INTRODUCTION

Design is defined in [IEEE610.12-90] as both "the process of defining the architecture, components, interfaces, and other characteristics of a system or component" and "the result of [that] process." Viewed as a process, software design is the software engineering life cycle activity in which software requirements are analyzed in order to produce a description of the software's internal structure that will serve as the basis for its construction. More precisely, a software design (the result) must describe the software architecture—that is, how software is decomposed and organized into components—and the interfaces between those components. It must also describe the components at a level of detail that enable their construction.

Software design plays an important role in developing software: it allows software engineers to produce various models that form a kind of blueprint of the solution to be implemented. We can analyze and evaluate these models to determine whether or not they will allow us to fulfill the various requirements. We can also examine and evaluate various alternative solutions and trade-offs. Finally, we can use the resulting models to plan the subsequent development activities, in addition to using them as input and the starting point of construction and testing.

In a standard listing of software life cycle processes such as IEEE/EIA 12207 Software Life Cycle Processes [IEEE12207.0-96], software design consists of two activities that fit between software requirements analysis and software construction:

- *Software architectural design* (sometimes called top-level design): describing software's top-level structure and organization and identifying the various components
- *Software detailed design*: describing each component sufficiently to allow for its construction.

Concerning the scope of the Software Design Knowledge Area (KA), the current KA description does not discuss every topic the name of which contains the word "design." In Tom DeMarco's terminology (DeM99), the KA discussed in this chapter deals mainly with D-design (decomposition design, mapping software into component pieces). However, because of its importance in the growing field of software architecture, we will also address FP-design (family pattern design, whose goal is to establish exploitable commonalities in a family of software). By contrast, the Software Design KA does not address I-design (invention design, usually performed during the software requirements process with the objective of conceptualizing and specifying software to satisfy discovered needs and requirements), since this topic should be considered part of requirements analysis and specification.

The Software Design KA description is related specifically to Software Requirements, Software Construction, Software Engineering Management, Software Quality, and Related Disciplines of Software Engineering.

BREAKDOWN OF TOPICS FOR SOFTWARE DESIGN

1. Software Design Fundamentals

The concepts, notions, and terminology introduced here form an underlying basis for understanding the role and scope of software design.

1.1. General Design Concepts

Software is not the only field where design is involved. In the general sense, we can view design as a form of problem-solving. [Bud03:c1] For example, the concept of a *wicked* problem–a problem with no definitive solution–is interesting in terms of understanding the limits of design. [Bud04:c1] A number of other notions and concepts are also of interest in understanding design in its general sense: goals, constraints, alternatives, representations, and solutions. [Smi93]

1.2. Context of Software Design

To understand the role of software design, it is important to understand the context in which it fits, the software engineering life cycle. Thus, it is important to understand the major characteristics of software requirements analysis vs. software design vs. software construction vs. software testing. [IEEE12207.0-96]; Lis01:c11; Mar02; Pfl01:c2; Pre04:c2]

1.3. Software Design Process

Software design is generally considered a two-step process: [Bas03; Dor02:v1c4s2; Fre83:I; IEEE12207.0-96]; Lis01:c13; Mar02:D]

1.3.1. Architectural design

Architectural design describes how software is decomposed and organized into components (the software architecture) [IEEEP1471-00]

1.3.2. Detailed design

Detailed design describes the specific behavior of these components.

The output of this process is a set of models and artifacts that record the major decisions that have been taken. [Bud04:c2; IEE1016-98; Lis01:c13; Pre04:c9]

1.4. Enabling Techniques

According to the *Oxford English Dictionary*, a *principle* is "a basic truth or a general law … that is used as a basis of reasoning or a guide to action." Software design principles, also called *enabling techniques* [Bus96], are key notions considered fundamental to many different software design approaches and concepts. The enabling techniques are the following: [Bas98:c6; Bus96:c6; IEEE1016-98; Jal97:c5,c6; Lis01:c1,c3; Pfl01:c5; Pre04:c9]

1.4.1. Abstraction

Abstraction is "the process of forgetting information so that things that are different can be treated as if they were the same." [Lis01] In the context of software design, two key abstraction mechanisms are parameterization and specification. Abstraction by specification leads to three major kinds of abstraction: procedural abstraction, data abstraction, and control (iteration) abstraction. [Bas98:c6; Jal97:c5,c6; Lis01:c1,c2,c5,c6; Pre04:c1]

1.4.2. Coupling and cohesion

Coupling is defined as the strength of the relationships between modules, whereas *cohesion* is defined by how the elements making up a module are related. [Bas98:c6; Jal97:c5; Pfl01:c5; Pre04:c9]

1.4.3. Decomposition and modularization

Decomposing and *modularizing* large software into a number of smaller independent ones, usually with the goal of placing different functionalities or responsibilities in different components. [Bas98:c6; Bus96:c6; Jal97 :c5; Pfl01:c5; Pre04:c9]

1.4.4. Encapsulation/information hiding

Encapsulation/information hiding means grouping and packaging the elements and internal details of an abstraction and making those details inaccessible. [Bas98:c6; Bus96:c6; Jal97:c5; Pfl01:c5; Pre04:c9]

1.4.5. Separation of interface and implementation

Separating interface and implementation involves defining a component by specifying a public interface, known to the clients, separate from the details of how the component is realized. [Bas98:c6; Bos00:c10; Lis01:c1,c9]

1.4.6. Sufficiency, completeness and primitiveness

Achieving sufficiency, completeness, and primitiveness means ensuring that a software component captures all the important characteristics of an abstraction, and nothing more. [Bus96:c6; Lis01:c5]

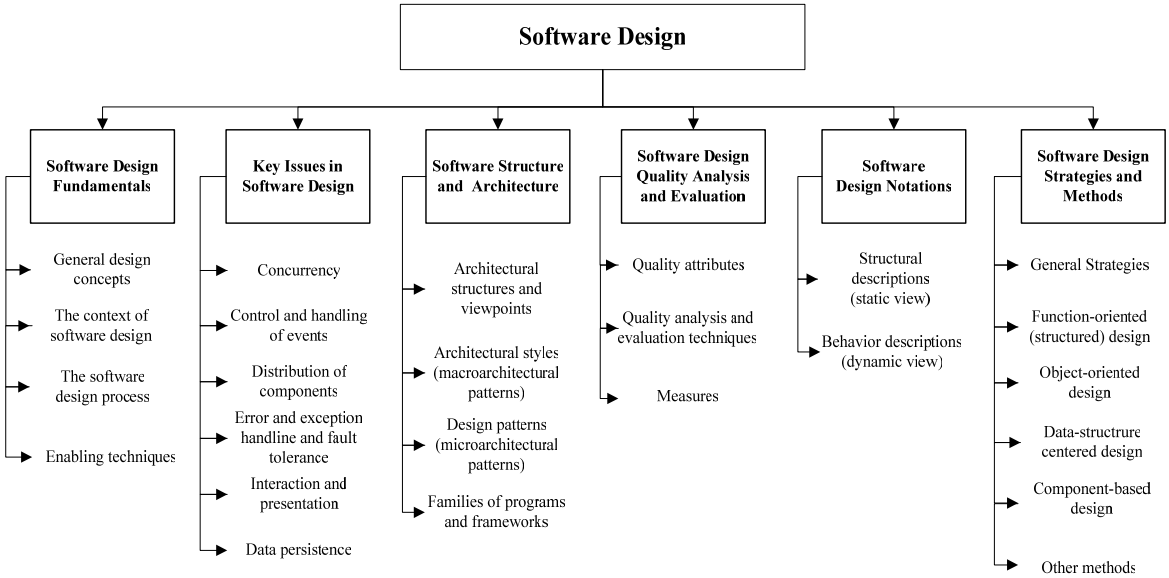


Figure 1 Breakdown of topics for the Software Design KA

2. Key Issues in Software Design

A number of key issues must be dealt with when designing software. Some are quality concerns that all software must address—for example, performance. Another important issue is how to decompose, organize, and package software components. This is so fundamental that all design approaches must address it in one way or another (see topic 1.4 *Enabling Techniques* and subarea 6 *Software Design Strategies and Methods*). In contrast, other issues "deal with some aspect of software's behavior that is not in the application domain, but which addresses some of the supporting domains." [Bos00] Such issues, which often cross-cut the system's functionality, have been referred to as *aspects*: "[aspects] tend not to be units of software's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways" (Kic97). A number of these key, cross-cutting issues are the following (presented in alphabetical order):

2.1. Concurrency

How to decompose the software into processes, tasks, and threads and deal with related efficiency, atomicity, synchronization, and scheduling issues. [Bos00:c5; Mar02:CSD; Mey97:c30; Pre04:c9]

2.2. Control and Handling of Events

How to organize data and control flow, how to handle reactive and temporal events through various mechanisms such as implicit invocation and call-backs. [Bas98:c5; Mey97:c32; Pfl01:c5]

2.3. Distribution of Components

How to distribute the software across the hardware, how the components communicate, how middleware can be used to deal with heterogeneous software. [Bas03:c16; Bos00:c5; Bus96:c2 Mar94:DD; Mey97:c30; Pre04:c30]

2.4. Error and Exception Handling and Fault Tolerance

How to prevent and tolerate faults and deal with exceptional conditions. [Lis01:c4; Mey97:c12; Pfl01:c5]

2.5. Interaction and Presentation

How to structure and organize the interactions with users and the presentation of information (for example, separation of presentation and business logic using the Model-View-Controller approach). [Bas98:c6; Bos00:c5; Bus96:c2; Lis01:c13; Mey97:c32] It is to be noted that this topic is not about specifying user interface details, which is the task of user interface design (a part of *Software Ergonomics*); see Related Disciplines of Software Engineering.

2.6. Data Persistence

How long-lived data are to be handled. [Bos00:c5; Mey97:c31]

3. Software Structure and Architecture

In its strict sense, a *software architecture* is "a description of the subsystems and components of a software system and the relationships between them." (Bus96:c6) Architecture thus attempts to define the internal *structure* — according to the *Oxford English Dictionary*, "the way in which something is constructed or organized" — of the resulting software. During the mid-1990s, however, software *architecture* started to emerge as a broader discipline involving the study of software structures and architectures in a more generic way [Sha96]. This gave rise to a number of interesting ideas about software design at different levels of abstraction. Some of these concepts can be useful during the architectural design (for example, architectural style) of specific software, as well as during its detailed design (for example, lower-level design patterns). But they can also be useful for designing generic systems, leading to the design of families of programs (also known as *product lines*). Interestingly, most of these concepts can be seen as attempts to describe, and thus reuse, generic design knowledge.

3.1. Architectural Structures and Viewpoints

Different high-level facets of a software design can and should be described and documented. These facets are often called *views*: "A view represents a partial aspect of a software architecture that shows specific properties of a software system" [Bus96:c6]. These distinct views pertain to distinct issues associated with software design — for example, the logical view (satisfying the functional requirements) vs. the process view (concurrency issues) vs. the physical view (distribution issues) vs. the development view (how the design is broken down into implementation units). Other authors use different terminologies, like behavioral vs. functional vs. structural vs. data modeling views. In summary, a software design is a multi-faceted artifact produced by the design process and generally composed of relatively independent and orthogonal views. [Bas03:c2; Boo99:c31; Bud04:c5; Bus96:c6; IEEE1016-98; IEEE1471-00]Architectural Styles (macroarchitectural patterns)

An architectural style is "a set of constraints on an architecture [that] defines a set or family of architectures that satisfies them" [Bas03:c2]. An architectural style can thus be seen as a meta-model which can provide software's high-level organization (its macroarchitecture). Various authors have identified a number of major architectural styles. [Bas03:c5; Boo99:c28; Bos00:c6; Bus96:c1,c6; Pfl01:c5]

- General structure (for example, layers, pipes, and filters, blackboard)
- Distributed systems (for example, client-server, three-tiers, broker)
- Interactive systems (for example, Model-View-Controller, Presentation-Abstraction-Control)
- Adaptable systems (for example, micro-kernel, reflection)
- Others (for example, batch, interpreters, process control, rule-based).

3.2. Design Patterns (microarchitectural patterns)

Succinctly described, a pattern is "a common solution to a common problem in a given context." (Jac99) While architectural styles can be viewed as patterns describing the high-level organization of software (their *macro*architecture), other design patterns can be used to describe details at a lower, more local level (their *micro*architecture). [Bas98:c13; Boo99:c28; Bus96:c1; Mar02:DP]

- Creational patterns (for example, builder, factory, prototype, and singleton)
- Structural patterns (for example, adapter, bridge, composite, decorator, façade, flyweight, and proxy)
- Behavioral patterns (for example, command, interpreter, iterator, mediator, memento, observer, state, strategy, template, visitor)

3.3. Families of Programs and Frameworks

One possible approach to allow the reuse of software designs and components is to design families of software, also known as *software product lines.* This can be done by identifying the commonalities among members of such families and by using reusable and customizable components to account for the variability among family members. [Bos00:c7,c10; Bas98:c15; Pre04:c30]

In OO programming, a key related notion is that of the framework*:* a partially complete software subsystem that can be extended by appropriately instantiating specific plug-ins (also known as *hot spots*). [Bos00:c11; Boo99:c28; Bus96:c6]

4. Software Design Quality Analysis and Evaluation

This section includes a number of quality and evaluation topics that are specifically related to software design. Most are covered in a general manner in the Software Quality KA.

4.1. Quality Attributes

Various attributes are generally considered important for obtaining a software design of good quality—various "ilities" (maintainability, portability, testability, traceability), various "nesses" (correctness, robustness), including "fitness of purpose." [Bos00:c5; Bud04:c4; Bus96:c6; ISO9126.1-01; ISO15026-98; Mar94:D; Mey97:c3; Pfl01:c5] An interesting distinction is the one between quality attributes discernable at run-time (performance, security, availability, functionality, usability), those not discernable at run-time (modifiability, portability, reusability, integrability, and testability), and those related to the architecture's intrinsic qualities (conceptual integrity, correctness, and completeness, buildability). [Bas03:c4]

4.2. Quality Analysis and Evaluation Techniques

Various tools and techniques can help ensure a software design's quality.

- *Software design reviews*: informal or semiformal, often group-based, techniques to verify and ensure the quality of design artifacts (for example, architecture reviews [Bas03:c11], design reviews, and inspections [Bud04:c4; Fre83:VIII; IEEE1028-97; Jal97:c5,c7; Lis01:c14; Pfl01:c5], scenario-based techniques [Bas98:c9; Bos00:c5], requirements tracing [Dor02:v1c4s2; Pfl01:c11])
- *Static analysis*: formal or semiformal static (non-executable) analysis that can be used to evaluate a design (for example, fault-tree analysis or automated cross-checking) [Jal97:c5; Pfl01:c5]
- *Simulation and prototyping*: dynamic techniques to evaluate a design (for example, performance simulation or feasibility prototype [Bas98:c10; Bos00:c5; Bud04:c4; Pfl01:c5])

4.3. Measures

Measures can be used to assess or to quantitatively estimate various aspects of a software design's size, structure, or quality. Most measures that have been proposed generally depend on the approach used for producing the design. These measures are classified in two broad categories:

- *Function-oriented (structured) design measures*: the design's structure, obtained mostly through functional decomposition; generally represented as a structure chart (sometimes called a hierarchical diagram) on which various measures can be computed [Jal97:c5,c7, Pre04:c15]
- *Object-oriented design measures*: the design's overall structure is often represented as a class diagram, on which various measures can be computed. Measures on the properties of each class's internal content can also be computed [Jal97:c6,c7; Pre04:c15]

5. Software Design Notations

Many notations and languages exist to represent software design artifacts. Some are used mainly to describe a design's structural organization, others to represent software behavior. Certain notations are used mostly during architectural design and others mainly during detailed design, although some notations can be used in both steps. In addition, some notations are used mostly in the context of specific methods (see the *Software Design Strategies and Methods* subarea). Here, they are categorized into notations for describing the structural (static) view vs. the behavioral (dynamic) view.

5.1. Structural Descriptions (static view)

The following notations, mostly (but not always) graphical, describe and represent the structural aspects of a software design—that is, they describe the major components and how they are interconnected (static view):

- *Architecture description languages* (*ADLs*): textual, often formal, languages used to describe a software architecture in terms of components and connectors [Bas03:c12]

- *Class and object diagrams*: used to represent a set of classes (and objects) and their interrelationships [Boo99:c8,c14; Jal97:c5,c6]
- *Component diagrams*: used to represent a set of components ("physical and replaceable part[s] of a system that [conform] to and [provide] the realization of a set of interfaces" [Boo99]) and their interrelationships [Boo99:c12,c31]
- *Class responsibility collaborator cards* (*CRCs*): used to denote the names of components (class), their responsibilities, and their collaborating components' names [Boo99:c4; Bus96]
- *Deployment diagrams*: used to represent a set of (physical) nodes and their interrelationships, and, thus, to model the physical aspects of a system [Boo99:c30]
- *Entity-relationship diagrams* (*ERDs*): used to represent conceptual models of data stored in information systems [Bud04:c6; Dor02:v1c5; Mar02:DR]
- *Interface description languages* (*IDLs*): programming-like languages used to define the interfaces (names and types of exported operations) of software components [Bas98:c8; Boo99:c11]
- *Jackson structure diagrams*: used to describe the data structures in terms of sequence, selection, and iteration [Bud04:c6; Mar02:DR]
- *Structure charts*: used to describe the calling structure of programs (which module calls, and is called by, which other module) [Bud04:c6; Jal97:c5; Mar02:DR; Pre04:c10]

5.2. Behavioral Descriptions (dynamic view)

The following notations and languages, some graphical and some textual, are used to describe the dynamic behavior of software and components. Many of these notations are useful mostly, but not exclusively, during detailed design.

- *Activity diagrams*: used to show the control flow from activity ("ongoing non-atomic execution within a state machine") to activity [Boo99:c19]
- *Collaboration diagrams*: used to show the interactions that occur among a group of objects, where the emphasis is on the objects, their links, and the messages they exchange on these links [Boo99:c18]
- *Data flow diagrams* (*DFDs*): used to show data flow among a set of processes [Bud04:c6; Mar02:DR; Pre04:c8]
- *Decision tables and diagrams*: used to represent complex combinations of conditions and actions [Pre04:c11]
- *Flowcharts and structured flowcharts*: used to represent the flow of control and the associated actions to be performed [Fre83:VII; Mar02:DR; Pre04:c11]
- *Sequence diagrams*: used to show the interactions among a group of objects, with emphasis on the time-ordering of messages [Boo99:c18]
- *State transition and statechart diagrams*: used to show the control flow from state to state in a state machine [Boo99:c24; Bud04:c6; Mar02:DR; Jal97:c7]
- *Formal specification languages*: textual languages that use basic notions from mathematics (for example, logic, set, sequence) to rigorously and abstractly define software component interfaces and behavior, often in terms of pre- and post-conditions [Bud04:c18; Dor02:v1c6s5; Mey97:c11]
- *Pseudocode and program design languages* (*PDLs*): structured-programming-like languages used to describe, generally at the detailed design stage, the behavior of a procedure or method [Bud04:c6; Fre83:VII; Jal97:c7; Pre04:c8, c11]

6. Software Design Strategies and Methods

There exist various general *strategies* to help guide the design process. [Bud04:c9, Mar02:D] In contrast with general strategies, *methods* are more specific in that they generally suggest and provide a set of notations to be used with the method, a description of the process to be used when following the method and a set of guidelines in using the method. [Bud04:c8] Such methods are useful as a means of transferring knowledge and as a common framework for teams of software engineers. [Bud03:c8] See also the Software Engineering Tools and Methods KA.

6.1. General Strategies

Some often-cited examples of general strategies useful in the design process are divide-and-conquer and stepwise refinement [Bud04:c12; Fre83:V], top-down vs. bottom-up strategies [Jal97:c5; Lis01:c13], data abstraction and information hiding [Fre83:V], use of heuristics [Bud04:c8], use of patterns and pattern languages [Bud04:c10; Bus96:c5], use of an iterative and incremental approach. [Pfl01:c2]

6.2. Function-Oriented (Structured) Design

[Bud04:c14; Dor02:v1c6s4; Fre83:V; Jal97:c5; Pre04:c9, c10]

This is one of the classical methods of software design, where decomposition centers on identifying the major software functions and then elaborating and refining them in a top-down manner. Structured design is generally used after structured analysis, thus producing, among other things, data flow diagrams and associated process descriptions. Researchers have proposed various strategies (for example, transformation analysis, transaction analysis) and heuristics (for example, fan-in/fan-out, scope of effect vs. scope of control) to transform a DFD into a software architecture generally represented as a structure chart.

6.3. *Object-Oriented Design*

[Bud0:c16; Dor02:v1:c6s2,s3; Fre83:VI; Jal97:c6; Mar02:D; Pre04:c9]

Numerous software design methods based on objects have been proposed. The field has evolved from the early object-based design of the mid-1980s (noun = object; verb = method; adjective = attribute) through OO design, where inheritance and polymorphism play a key role, to the field of component-based design, where meta-information can be defined and accessed (through reflection, for example). Although OO design's roots stem from the concept of data abstraction, responsibility-driven design has also been proposed as an alternative approach to OO design.

6.4. *Data-Structure-Centered Design*

[Bud04:c15; Fre83:III,VII; Mar02:D]

Data-structure-centered design (for example, Jackson, Warnier-Orr) starts from the data structures a program manipulates rather than from the function it performs. The software engineer first describes the input and output data structures (using Jackson's structure diagrams, for instance) and then develops the program's control structure based on these data structure diagrams. Various heuristics have been proposed to deal with special cases—for example, when there is a mismatch between the input and output structures.

6.5. *Component-Based Design (CBD)*

A software component is an independent unit, having well-defined interfaces and dependencies that can be composed and deployed independently. Component-based design addresses issues related to providing, developing, and integrating such components in order to improve reuse. [Bud04:c11]

6.6. *Other Methods*

Other interesting but less mainstream approaches also exist: formal and rigorous methods [Bud04:c18; Dor02:c5; Fre83; Mey97:c11; Pre04:c29] and transformational methods. [Pfl98:c2]

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	[Bas03] {Bas98}	[Boo99]	[Bos00]	[Bud03]	[Bus96]	[Dor02]	[Fre83]	[IEEE1016-98]	[IEEE1028-97]	[IEEE1471-00]	[IEEE12207.0-96]	[ISO9126-01]	[ISO15026-98]	[Jal97]	[Lis01]	[Mar02]* {Mar94}	[Mey97]	[Pfl01]	[Pre04]	[Smi93]
1. Software Design Fundamentals																				
1.1 General Design Concepts				C1																*
1.2 The Context of Software Design											*				c11s1	D		c2s2	c2	
1.3 The Software Design Process	c2s1, c2s4			C2		v1c4s2	2-22	*		*	*				c13s1, c13s2	D			c9	
1.4 Enabling Techniques	{c6s1}		c10s3		c6s3			*						c5s1, c5s2, c6s2	c1s1,c1s2, c3s1-c3s3, 77-85, c5s8, 125-128, c9s1-c9s3			c5s2, c5s5	c9	
2. Key Issues in Software Design																				
2.1 Concurrency			c5s4.1													CSD	c30		c9	
2.2 Control and Handling of Events	{c5s2}																c32s4, c32s5	c5s3		
2.3 Distribution of Components	c16s3, c16s4		c5s4.1		c2s3											{DD}	c30		c30	
2.4 Error and Exception Handling and Fault Tolerance															c4s3-c4s5		c12	c5s5		
2.5 Interaction and Presentation	{c6s2}		c5s4.1		c2s4										c13s3		c32s2			
2.6 Data Persistence			c5s4.1														c31			

* see the next section

	[Bas03] {Bas98}	[Boo99]	[Bos00]	[Bud03]	[Bus96]	[Dor02]	[Fre83]	[IEEE1016-98]	[IEEE1028-97]	[IEEE1471-00]	[IEEE12207.0-96]	[ISO9126-01]	[ISO15026-98]	[Jal97]	[Lis01]	[Mar02]* {Mar94}	[Mey97]	[Pfl01]	[Pre04]	[Smi93]
3. Software Structure and Architecture																				
3.1 Architectural Structures and Viewpoints	c2s5	c31		c5	c6s1			*		*										
3.2 Architectural Styles	c5s9	c28	c6s3.1		c1s1-c1s3, c6s2													c5s3		
3.3 Design Patterns	{c13s3}	c28			c1s1-c1s3											DP				
3.4 Families of Programs and Frameworks	{c15s1, c15s3}	c28	c7s1, c7s2, c10s2-c10s4, c11s2, c11s4		c6s2														C30	
4. Software Design Quality Analysis and Evaluation																				
4.1 Quality Attributes	c4s2		c5s2.3	c4	c6s4							*	*			{D}	c3	c5s5		
4.2 Quality Analysis and Evaluation Techniques	c11s3, {c9s1, c9s2, c10s2, c10s3}		c5s2.1, c5s2.2, c5s3, c5s4	c4		v1c4s2	542-576		*					c5s5, c7s3	c14s1			c5s6, c5s7, c11s5		
4.3 Measures														c5s6, c6s5, c7s4					c15	

	[Bas03] {Bas98}	[Boo99]	[Bos00]	[Bud03]	[Bus96]	[Dor02]	[Fre83]	[IEEE1016-98]	[IEEE1028-97]	[IEEE1471-00]	[IEEE12207.0-96]	[ISO9126-01]	[ISO15026-98]	[Jal97]	[Lis01]	[Mar02]* {Mar94}	[Mey97]	[Pfl01]	[Pre04]	[Smi93]
5. Software Design Notations																				
5.1 Structural Descriptions (Static View)	{c8s4} c12s1, c12s2	c4, c8, c11, c12, c14, c30, c31		c6	429									c5s3,c 6s3		DR			c10	
5.2 Behavioral Descriptions (Dynamic View)		c18, c19, c24		c6, c18		v1c5	485-490, 506-513							c7s2		DR	c11		c8, c11	
6. Software Design Strategies and Methods																				
6.1 General Strategies				c8, c10, c12	c5s1-c5s4		304-320, 533-539							c5s1.4	c13s13			c2s2		
6.2 Function-Oriented (Structured) Design				c14		v1c6s4	328-352							c5s4					c9, c10	
6.3 Object-Oriented Design				c16		v1c6s2 , v1c6s3	420-436							c6s4		D			c9	
6.4 Data-Structure-Centered Design				C15			201-210, 514-532									D				
6.5 Component-Based Design (CBD)				c11																
6.6 Other Methods				c18		181-192	395-407										c11	c2s2	C29	

RECOMMENDED REFERENCES FOR SOFTWARE DESIGN

[Bas98] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998.

[Bas03] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, second ed., Addison-Wesley, 2003.

[Boo99] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[Bos00] J. Bosch, *Design & Use of Software Architectures: Adopting and Evolving a Product-Line Approach*, first ed., ACM Press, 2000.

[Bud04] D. Budgen, *Software Design*, second ed., Addison-Wesley, 2004.

[Bus96] F. Buschmann et al., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.

[Dor02] M. Dorfman and R.H. Thayer, eds., *Software Engineering* (Vol. 1 & Vol. 2), IEEE Computer Society Press, 2002.

[Fre83] P. Freeman and A.I. Wasserman, *Tutorial on Software Design Techniques*, fourth ed., IEEE Computer Society Press, 1983.

[IEEE610.12-90] IEEE Std 610.12-1990 (R2002), *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1990.

[IEEE1016-98] IEEE Std 1016-1998, *IEEE Recommended Practice for Software Design Descriptions*, IEEE, 1998.

[IEEE1028-97] IEEE Std 1028-1997 (R2002), *IEEE Standard for Software Reviews*, IEEE, 1997.

[IEEE1471-00] IEEE Std 1471-2000, *IEEE Recommended Practice for Architectural Description of Software Intensive Systems*, Architecture Working Group of the Software Engineering Standards Committee, 2000.

[IEEE12207.0-96] IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.

[ISO9126-01] ISO/IEC 9126-1:2001, *Software Engineering Product Quality—Part 1: Quality Model*, ISO and IEC, 2001.

[ISO15026-98] ISO/IEC 15026-1998, *Information Technology — System and Software Integrity Levels*, ISO and IEC, 1998.

[Jal97] P. Jalote, *An Integrated Approach to Software Engineering*, second ed., Springer-Verlag, 1997.

[Lis01] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, Addison-Wesley, 2001.

[Mar94] J.J. Marciniak, *Encyclopedia of Software Engineering*, J. Wiley & Sons, 1994.

The references to the *Encyclopedia* are as follows:

CBD = Component-Based Design

D = Design

DD = Design of the Distributed System

DR = Design Representation

[Mar02] J.J. Marciniak, *Encyclopedia of Software Engineering*, second ed., J. Wiley & Sons, 2002.

[Mey97] B. Meyer, *Object-Oriented Software Construction*, second ed., Prentice-Hall, 1997.

[Pfl01] S.L. Pfleeger, *Software Engineering: Theory and Practice*, second ed., Prentice-Hall, 2001.

[Pre04] R.S. Pressman, *Software Engineering: A Practitioner's Approach*, sixth ed., McGraw-Hill, 2004.

[Smi93] G. Smith and G. Browne, "Conceptual Foundations of Design Problem-Solving," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 23, iss. 5, 1209-1219, Sep.-Oct. 1993.

APPENDIX A. LIST OF FURTHER READINGS

(Boo94a) G. Booch, *Object Oriented Analysis and Design with Applications*, second ed., The Benjamin/Cummings Publishing Company, 1994.

(Coa91) P. Coad and E. Yourdon, *Object-Oriented Design*, Yourdon Press, 1991.

(Cro84) N. Cross, *Developments in Design Methodology*, John Wiley & Sons, 1984.

(DSo99) D.F. D'Souza and A.C. Wills, *Objects, Components, and Frameworks with UML — The Catalysis Approach*, Addison-Wesley, 1999.

(Dem99) T. DeMarco, "The Paradox of Software Architecture and Design," *Stevens Prize Lecture*, Aug. 1999.

(Fen98) N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, second ed., International Thomson Computer Press, 1998.

(Fow99) M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

(Fow03) M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

(Gam95) E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

(Hut94) A.T.F. Hutt, *Object Analysis and Design — Comparison of Methods. Object Analysis and Design — Description of Methods*, John Wiley & Sons, 1994.

(Jac99) I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.

(Kic97) G. Kiczales et al., "Aspect-Oriented Programming," presented at ECOOP '97 — Object-Oriented Programming, 1997.

(Kru95) P. B. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, vol. 12, iss. 6, 42-50, 1995

(Lar98) C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice-Hall, 1998.

(McC93) S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, 1993.

(Pag00) M. Page-Jones, *Fundamentals of Object-Oriented Design in UML*, Addison-Wesley, 2000.

(Pet92) H. Petroski, *To Engineer Is Human: The Role of Failure in Successful Design*, Vintage Books, 1992.

(Pre95) W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley and ACM Press, 1995.

(Rie96) A.J. Riel, *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.

(Rum91) J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

(Sha96) M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

(Som05) I. Sommerville, *Software Engineering*, seventh ed., Addison-Wesley, 2005.

(Wie98) R. Wieringa, "A Survey of Structured and Object-Oriented Software Specification Methods and Techniques," *ACM Computing Surveys*, vol. 30, iss. 4, 1998, pp. 459-527.

(Wir90) R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, Prentice-Hall, 1990.

APPENDIX B. LIST OF STANDARDS

(IEEE610.12-90) IEEE Std 610.12-1990 (R2002), *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1990.

(IEEE1016-98) IEEE Std 1016-1998, *IEEE Recommended Practice for Software Design Descriptions*, IEEE, 1998.

(IEEE1028-97) IEEE Std 1028-1997 (R2002), *IEEE Standard for Software Reviews*, IEEE, 1997.

(IEEE1471-00) IEEE Std 1471-2000, *IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems*, Architecture Working Group of the Software Engineering Standards Committee, 2000.

(IEEE12207.0-96) IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, vol. IEEE, 1996.

(ISO9126-01) ISO/IEC 9126-1:2001, *Software Engineering-Product Quality-Part 1: Quality Model*, ISO and IEC, 2001.

(ISO15026-98) ISO/IEC 15026-1998 *Information Technology — System and Software Integrity Levels*, ISO and IEC, 1998.

CHAPTER 4
SOFTWARE CONSTRUCTION

ACRONYMS

OMG	Object Management Group
UML	Unified Modeling Language

INTRODUCTION

The term software construction refers to the detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging.

The Software Construction Knowledge Area is linked to all the other KAs, most strongly to Software Design and Software Testing. This is because the software construction process itself involves significant software design and test activity. It also uses the output of design and provides one of the inputs to testing, both design and testing being the activities, not the KAs in this case. Detailed boundaries between design, construction, and testing (if any) will vary depending upon the software life cycle processes that are used in a project.

Although some detailed design may be performed prior to construction, much design work is performed within the construction activity itself. Thus the Software Construction KA is closely linked to the Software Design KA.

Throughout construction, software engineers both unit-test and integration-test their work. Thus, the Software Construction KA is closely linked to the Software Testing KA as well.

Software construction typically produces the highest volume of configuration items that need to be managed in a software project (source files, content, test cases, and so on). Thus, the Software Construction KA is also closely linked to the Software Configuration Management KA.

Since software construction relies heavily on tools and methods and is probably the most tool-intensive of the KAs, it is linked to the Software Engineering Tools and Methods KA.

While software quality is important in all the KAs, code is the ultimate deliverable of a software project, and thus Software Quality is also closely linked to Software Construction.

Among the Related Disciplines of Software Engineering, the Software Construction KA is most akin to computer science in its use of knowledge of algorithms and of detailed coding practices, both of which are often considered to belong to the computer science domain. It is also related to project management, insofar as the management of construction can present considerable challenges.

BREAKDOWN OF TOPICS FOR SOFTWARE CONSTRUCTION

The breakdown of the Software Construction KA is presented below, together with brief descriptions of the major topics associated with it. Appropriate references are also given for each of the topics. Figure 1 gives a graphical representation of the top-level decomposition of the breakdown for this KA.

1. Software Construction Fundamentals

The fundamentals of software construction include

- Minimizing complexity
- Anticipating change
- Constructing for verification
- Standards in construction

The first three concepts apply to design as well as to construction. The following sections define these concepts and describe how they apply to construction.

1.1. Minimizing Complexity

[Bec99; Ben00; Hun00; Ker99; Mag93; McC04]

A major factor in how people convey intent to computers is the severely limited ability of people to hold complex structures and information in their working memories, especially over long periods of time. This leads to one of the strongest drivers in software construction: minimizing complexity. The need to reduce complexity applies to essentially every aspect of software construction, and is particularly critical to the process of verification and testing of software constructions.

In software construction, reduced complexity is achieved through emphasizing the creation of code that is simple and readable rather than clever.

Minimizing complexity is accomplished through making use of standards, which is discussed in topic 1.4 *Standards in Construction*, and through numerous specific techniques which are summarized in topic 3.3 *Coding*. It is also supported by the construction-focused quality techniques summarized in topic 3.5 *Construction Quality*.

1.2. Anticipating Change

[Ben00; Ker99; McC04]

Most software will change over time, and the anticipation of change drives many aspects of software construction. Software is unavoidably part of changing external environments, and changes in those outside environments affect software in diverse ways.

Anticipating change is supported by many specific techniques summarized in topic 3.3 *Coding*.

- Communication methods (for example, standards for document formats and contents)
- Programming languages (for example, language standards for languages like Java and C++)
- Platforms (for example, programmer interface standards for operating system calls)
- Tools (for example, diagrammatic standards for notations like UML (Unified Modeling Language))

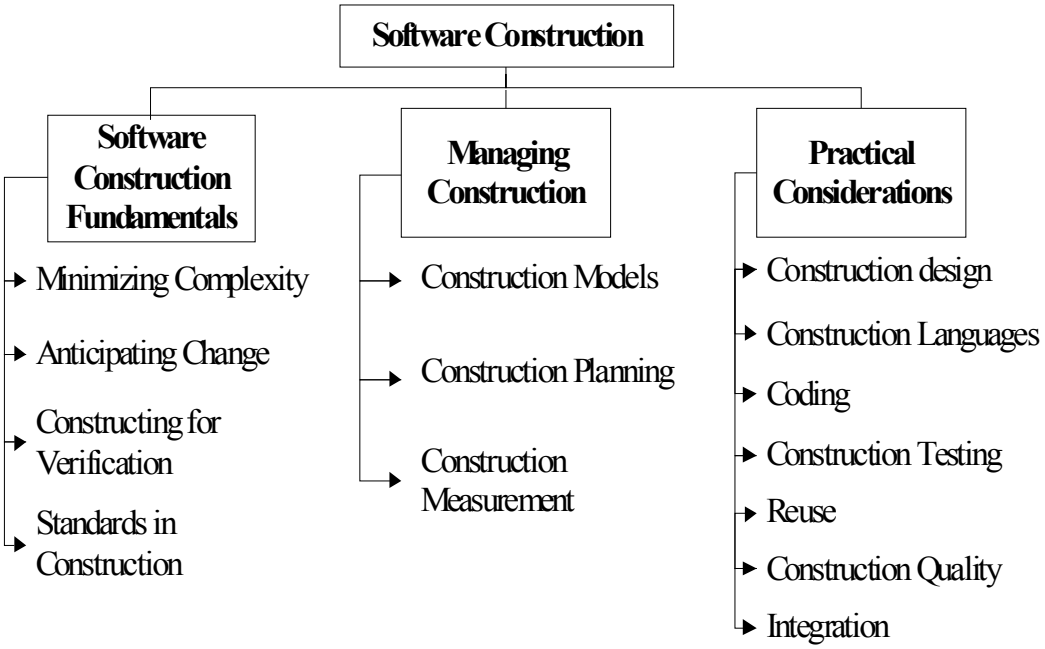


Figure 1. Breakdown of topics for the Software Construction KA.

1.3. Constructing for Verification

[Ben00; Hun00; Ker99; Mag93; McC04]

Constructing for verification means building software in such a way that faults can be ferreted out readily by the software engineers writing the software, as well as during independent testing and operational activities. Specific techniques that support constructing for verification include following coding standards to support code reviews, unit testing, organizing code to support automated testing, and restricted use of complex or hard-to-understand language structures, among others.

1.4. Standards in Construction

[IEEE12207-95; McC04]

Standards that directly affect construction issues include

Use of external standards. Construction depends on the use of external standards for construction languages, construction tools, technical interfaces, and interactions between Software Construction and other KAs. Standards come from numerous sources, including hardware and software interface specifications such as the Object Management Group (OMG) and international organizations such as the IEEE or ISO.

Use of internal standards. Standards may also be created on an organizational basis at the corporate level or for use on specific projects. These standards support coordination of group activities, minimizing complexity, anticipating change, and constructing for verification.

2. Managing Construction

2.1. Construction Models [Bec99; McC04]

Numerous models have been created to develop software, some of which emphasize construction more than others.

Some models are more linear from the construction point of view, such as the waterfall and staged-delivery life cycle models. These models treat construction as an activity which occurs only after significant prerequisite work has been completed—including detailed requirements work, extensive design work, and detailed planning. The more linear approaches tend to emphasize the activities that precede construction (requirements and design), and tend to create more distinct separations between the activities. In these models, the main emphasis of construction may be coding.

Other models are more iterative, such as evolutionary prototyping, Extreme Programming, and Scrum. These approaches tend to treat construction as an activity that occurs concurrently with other software development activities, including requirements, design, and planning, or overlaps them. These approaches tend to mix design, coding, and testing activities, and they often treat the combination of activities as construction.

Consequently, what is considered to be "construction" depends to some degree on the life cycle model used.

2.2. Construction Planning [Bec99; McC04]

The choice of construction method is a key aspect of the construction planning activity. The choice of construction method affects the extent to which construction prerequisites are performed, the order in which they are performed, and the degree to which they are expected to be completed before construction work begins.

The approach to construction affects the project's ability to reduce complexity, anticipate change, and construct for verification. Each of these objectives may also be addressed at the process, requirements, and design levels—but they will also be influenced by the choice of construction method.

Construction planning also defines the order in which components are created and integrated, the software quality management processes, the allocation of task assignments to specific software engineers, and the other tasks, according to the chosen method.

2.3. Construction Measurement [McC04]

Numerous construction activities and artifacts can be measured, including code developed, code modified, code reused, code destroyed, code complexity, code inspection statistics, fault-fix and fault-find rates, effort, and scheduling. These measurements can be useful for purposes of managing construction, ensuring quality during construction, improving the construction process, as well as for other reasons. See the Software Engineering Process KA for more on measurements.

3. Practical considerations

Construction is an activity in which the software has to come to terms with arbitrary and chaotic real-world constraints, and to do so exactly. Due to its proximity to real-world constraints, construction is more driven by practical considerations than some other KAs, and software engineering is perhaps most craft-like in the construction area.

3.1. Construction Design [Bec99; Ben00; Hun00; IEEE12207-95; Mag93; McC04]

Some projects allocate more design activity to construction; others to a phase explicitly focused on design. Regardless of the exact allocation, some detailed design work will occur at the construction level, and that design work tends to be dictated by immovable constraints imposed by the real-world problem that is being addressed by the software.

Just as construction workers building a physical structure must make small-scale modifications to account for unanticipated gaps in the builder's plans, software construction workers must make modifications on a smaller or larger scale to flesh out details of the software design during construction.

The details of the design activity at the construction level are essentially the same as described in the Software Design KA, but they are applied on a smaller scale.

3.2 Construction Languages [Hun00; McC04]

Construction languages include all forms of communication by which a human can specify an executable problem solution to a computer.

The simplest type of construction language is a *configuration language,* in which software engineers choose from a limited set of predefined options to create new or custom software installations. The text-based configuration files used in both the Windows and Unix operating systems are examples of this, and the menu style selection lists of some program generators constitute another.

Toolkit languages are used to build applications out of toolkits (integrated sets of application-specific reusable parts), and are more complex than configuration languages. Toolkit languages may be explicitly defined as application programming languages (for example, scripts), or may simply be implied by the set of interfaces of a toolkit.

Programming languages are the most flexible type of construction languages. They also contain the least amount of information about specific application areas and

development processes, and so require the most training and skill to use effectively.

There are three general kinds of notation used for programming languages, namely:

- Linguistic
- Formal
- Visual

Linguistic notations are distinguished in particular by the use of word-like strings of text to represent complex software constructions, and the combination of such word-like strings into patterns that have a sentence-like syntax. Properly used, each such string should have a strong semantic connotation providing an immediate intuitive understanding of what will happen when the underlying software construction is executed.

Formal notations rely less on intuitive, everyday meanings of words and text strings and more on definitions backed up by precise, unambiguous, and formal (or mathematical) definitions. Formal construction notations and formal methods are at the heart of most forms of system programming, where accuracy, time behavior, and testability are more important than ease of mapping into natural language. Formal constructions also use precisely defined ways of combining symbols that avoid the ambiguity of many natural language constructions.

Visual notations rely much less on the text-oriented notations of both linguistic and formal construction, and instead rely on direct visual interpretation and placement of visual entities that represent the underlying software. Visual construction tends to be somewhat limited by the difficulty of making "complex" statements using only movement of visual entities on a display. However, it can also be a powerful tool in cases where the primary programming task is simply to build and "adjust" a visual interface to a program, the detailed behavior of which has been defined earlier.

3.2. Coding

[Ben00; IEEE12207-95; McC04]

The following considerations apply to the software construction coding activity:

- Techniques for creating understandable source code, including naming and source code layout
- Use of classes, enumerated types, variables, named constants, and other similar entities
- Use of control structures
- Handling of error conditions—both planned errors and exceptions (input of bad data, for example)
- Prevention of code-level security breaches (buffer overruns or array index overflows, for example)
- Resource usage via use of exclusion mechanisms and discipline in accessing serially reusable resources (including threads or database locks)
- Source code organization (into statements, routines, classes, packages, or other structures)
- Code documentation
- Code tuning

3.3. Construction Testing

[Bec99; Hun00; Mag93; McC04]

Construction involves two forms of testing, which are often performed by the software engineer who wrote the code:

- Unit testing
- Integration testing

The purpose of construction testing is to reduce the gap between the time at which faults are inserted into the code and the time those faults are detected. In some cases, construction testing is performed after code has been written. In other cases, test cases may be created before code is written.

Construction testing typically involves a subset of types of testing, which are described in the Software Testing KA. For instance, construction testing does not typically include system testing, alpha testing, beta testing, stress testing, configuration testing, usability testing, or other, more specialized kinds of testing.

Two standards have been published on the topic: IEEE Std 829-1998*, IEEE Standard for Software Test Documentation* and IEEE Std 1008-1987, *IEEE Standard for Software Unit Testing*.

See also the corresponding sub-topics in the Software Testing KA: 2.1.1 *Unit Testing* and 2.1.2 *Integration Testing* for more specialized reference material.

3.4. Reuse

[IEEE1517-99; Som05].

As stated in the introduction of (IEEE1517-99):

"Implementing software reuse entails more than creating and using libraries of assets. It requires formalizing the practice of reuse by integrating reuse processes and activities into the software life cycle." However, reuse is important enough in software construction that it is included here as a topic.

The tasks related to reuse in software construction during coding and testing are:

- The selection of the reusable units, databases, test procedures, or test data
- The evaluation of code or test reusability
- The reporting of reuse information on new code, test procedures, or test data

3.5. Construction Quality

[Bec99; Hun00; IEEE12207-95; Mag93; McC04]

Numerous techniques exist to ensure the quality of code as it is constructed. The primary techniques used for construction include

- Unit testing and integration testing (as mentioned in topic 3.4 *Construction Testing*)
- Test-first development (see also the Software Testing KA, topic 2.2 *Objectives of Testing*)
- Code stepping
- Use of assertions
- Debugging
- Technical reviews (see also the Software Quality KA, sub-topic 2.3.2 *Technical Reviews*)
- Static analysis (IEEE1028) (see also the Software Quality KA, topic 2.3 *Reviews and Audits*)

The specific technique or techniques selected depend on the nature of the software being constructed, as well as on the skills set of the software engineers performing the construction.

Construction quality activities are differentiated from other quality activities by their focus. Construction quality activities focus on code and on artifacts that are closely related to code: small-scale designs—as opposed to other artifacts that are less directly connected to the code, such as requirements, high-level designs, and plans.

3.7 Integration

[Bec99; IEEE12207-95; McC04]

A key activity during construction is the integration of separately constructed routines, classes, components, and subsystems. In addition, a particular software system may need to be integrated with other software or hardware systems.

Concerns related to construction integration include planning the sequence in which components will be integrated, creating scaffolding to support interim versions of the software, determining the degree of testing and quality work performed on components before they are integrated, and determining points in the project at which interim versions of the software are tested.

Matrix of Topics vs. Reference Material

	[Bec99]	[Ben00]	[Hun00]	[IEEE 1517]	[IEEE 12207.0]	[Ker99]	[Mag93]	[Mcc04]	[Som05]
1. Software Construction Fundamentals									
1.1 Minimizing Complexity	c17	c2, c3	c7, c8			c2, c3	c6	c2, c3, c7-c9, c24, c27, c28, c31, c32, c34	
1.2 Anticipating Change		c11, c13, c14				c2, c9		c3-c5, c24, c31, c32, c34	
1.3 Constructing for Verification		c4	c21, c23, c34, c43			c1, c5, c6	c2, c3, c5, c7	c8, c20-c23, c31, c34	
1.4 Standards in Construction					X			c4	
2. Managing Construction									
2.1 Construction Modals	c10							c2, c3, c27, c29	
2.2 Construction Planning	c12, c15, c21							c3, c4, c21, c27-c29	
2.3 Construction Measurement								c25, c28	
3. Practical Considerations									
3.1 Construction Design	c17	c8-c10, p175-6	c33		X		c6	c3, c5, c24	
3.2 Construction Languages			c12, c14-c20					c4	
3.3 Coding		c6-c10			X			c5-c19, c25-c26	
3.4 Construction Testing	c18		c34, c43		X		c4	c22, c23	
3.5 Reuse				X					c14
3.6 Construction Quality	c18		c18		X		c4, c6, c7	c8, c20-c25	
3.7 Integration	c16				X			c29	

RECOMMENDED REFERENCES FOR SOFTWARE CONSTRUCTION

[Bec99] K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999, Chap. 10, 12, 15, 16-18, 21.

[Ben00a] J. Bentley, *Programming Pearls*, second ed., Addison-Wesley, 2000, Chap. 2-4, 6-11, 13, 14, pp. 175-176.

[Hun00] A. Hunt and D. Thomas, *The Pragmatic Programmer*, Addison-Wesley, 2000, Chap. 7, 8 12, 14-21, 23, 33, 34, 36-40, 42, 43.

[IEEE1517-99] IEEE Std 1517-1999, *IEEE Standard for Information Technology-Software Life Cycle Processes-Reuse Processes*, IEEE, 1999.

[IEEE12207.0-96] IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.

[Ker99a] B.W. Kernighan and R. Pike, *The Practice of Programming*, Addison-Wesley, 1999, Chap. 2, 3, 5, 6, 9.

[Mag93] S. Maguire, *Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Software*, Microsoft Press, 1993, Chap. 2-7.

[McC04] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, second ed., 2004.

[Som05] I. Sommerville, *Software Engineering*, seventh ed., Addison-Wesley, 2005.

APPENDIX A. LIST OF FURTHER READINGS

(Bar98) T.T. Barker, *Writing Software Documentation: A Task-Oriented Approach*, Allyn & Bacon, 1998.

(Bec02) K. Beck, *Test-Driven Development: By Example*, Addison-Wesley, 2002.

(Fow99) M. Fowler and al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

(How02) M. Howard and D.C. Leblanc, *Writing Secure Code*, Microsoft Press, 2002.

(Hum97b) W.S. Humphrey, *Introduction to the Personal Software Process*, Addison-Wesley, 1997.

(Mey97) B. Meyer, *Object-Oriented Software Construction*, second ed., Prentice Hall, 1997, Chap. 6, 10, 11.

(Set96) R. Sethi, *Programming Languages: Concepts & Constructs*, second ed., Addison-Wesley, 1996, Parts II-V.

APPENDIX B. LIST OF STANDARDS

(IEEE829-98) IEEE Std 829-1998, *IEEE Standard for Software Test Documentation*, IEEE, 1998.

(IEEE1008-87) IEEE Std 1008-1987 (R2003), *IEEE Standard for Software Unit Testing*, IEEE, 1987.

(IEEE1028-97) IEEE Std 1028-1997 (R2002), *IEEE Standard for Software Reviews*, IEEE, 1997.

(IEEE1517-99) IEEE Std 1517-1999, *IEEE Standard for Information Technology-Software Life Cycle Processes-Reuse Processes*, IEEE, 1999.

(IEEE12207.0-96) IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.

CHAPTER 5
SOFTWARE TESTING

ACRONYM

SRET	Software Reliability Engineered Testing

INTRODUCTION

Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems.

Software testing consists of the *dynamic* verification of the behavior of a program on a *finite* set of test cases, suitably *selected* from the usually infinite executions domain, against the *expected* behavior.

In the above definition, italicized words correspond to key issues in identifying the Knowledge Area of Software Testing. In particular:

- *Dynamic*: This term means that testing always implies executing the program on (valued) inputs. To be precise, the input value alone is not always sufficient to determine a test, since a complex, nondeterministic system might react to the same input with different behaviors, depending on the system state. In this KA, though, the term "input" will be maintained, with the implied convention that its meaning also includes a specified input state, in those cases in which it is needed. Different from testing and complementary to it are static techniques, as described in the Software Quality KA.
- *Finite*: Even in simple programs, so many test cases are theoretically possible that exhaustive testing could require months or years to execute. This is why in practice the whole test set can generally be considered infinite. Testing always implies a trade-off between limited resources and schedules on the one hand and inherently unlimited test requirements on the other.
- *Selected*: The many proposed test techniques differ essentially in how they select the test set, and software engineers must be aware that different selection criteria may yield vastly different degrees of effectiveness. How to identify the most suitable selection criterion under given conditions is a very complex problem; in practice, risk analysis techniques and test engineering expertise are applied.
- *Expected:* It must be possible, although not always easy, to decide whether the observed outcomes of program execution are acceptable or not, otherwise the testing effort would be useless. The observed behavior may be checked against user expectations (commonly referred to as testing for validation), against a specification (testing for verification), or, finally, against the anticipated behavior from implicit requirements or reasonable expectations. See, in the Software Requirements KA, topic 6.4 *Acceptance Tests*.

The view of software testing has evolved towards a more constructive one. Testing is no longer seen as an activity which starts only after the coding phase is complete, with the limited purpose of detecting failures. Software testing is now seen as an activity which should encompass the whole development and maintenance process and is itself an important part of the actual product construction. Indeed, planning for testing should start with the early stages of the requirement process, and test plans and procedures must be systematically and continuously developed, and possibly refined, as development proceeds. These test planning and designing activities themselves constitute useful input for designers in highlighting potential weaknesses (like design oversights or contradictions, and omissions or ambiguities in the documentation).

It is currently considered that the right attitude towards quality is one of prevention: it is obviously much better to avoid problems than to correct them. Testing must be seen, then, primarily as a means for checking not only whether the prevention has been effective, but also for identifying faults in those cases where, for some reason, it has not been effective. It is perhaps obvious but worth recognizing that, even after successful completion of an extensive testing campaign, the software could still contain faults. The remedy for software failures experienced after delivery is provided by corrective maintenance actions. Software maintenance topics are covered in the Software Maintenance KA.

In the Software Quality KA (See topic 3.3 *Software Quality Management Techniques*), software quality management techniques are notably categorized into *static* techniques (no code execution) and *dynamic* techniques (code execution). Both categories are useful. This KA focuses on dynamic techniques.

Software testing is also related to software construction (see topic 3.4 *Construction Testing* in that KA). Unit and integration testing are intimately related to software construction, if not part of it.

BREAKDOWN OF TOPICS

The breakdown of topics for the Software Testing KA is shown in Figure 1.

The first subarea describes *Software Testing Fundamentals*. It covers the basic definitions in the field of software testing, the basic terminology and key issues, and its relationship with other activities.

The second subarea, *Test Levels*, consists of two (orthogonal) topics: 2.1 lists the levels in which the testing

of large software is traditionally subdivided; and 2.2 considers testing for specific conditions or properties and is referred to as *objectives of testing*. Not all types of testing apply to every software product, nor has every possible type been listed.

The test target and test objective together determine how the test set is identified, both with regard to its consistency—*how much testing is enough for achieving the stated objective*—and its composition—*which test cases should be selected for achieving the stated objective* (although usually the "for achieving the stated objective" part is left implicit and only the first part of the two italicized questions above is posed). Criteria for addressing the first question are referred to as *test adequacy* criteria, while those addressing the second question are the *test selection* criteria.

Several *Test Techniques* have been developed in the past few decades, and new ones are still being proposed. Generally accepted techniques are covered in subarea 3.

Test-related Measures are dealt with in subarea 4.

Finally, issues relative to *Test Process* are covered in subarea 5.

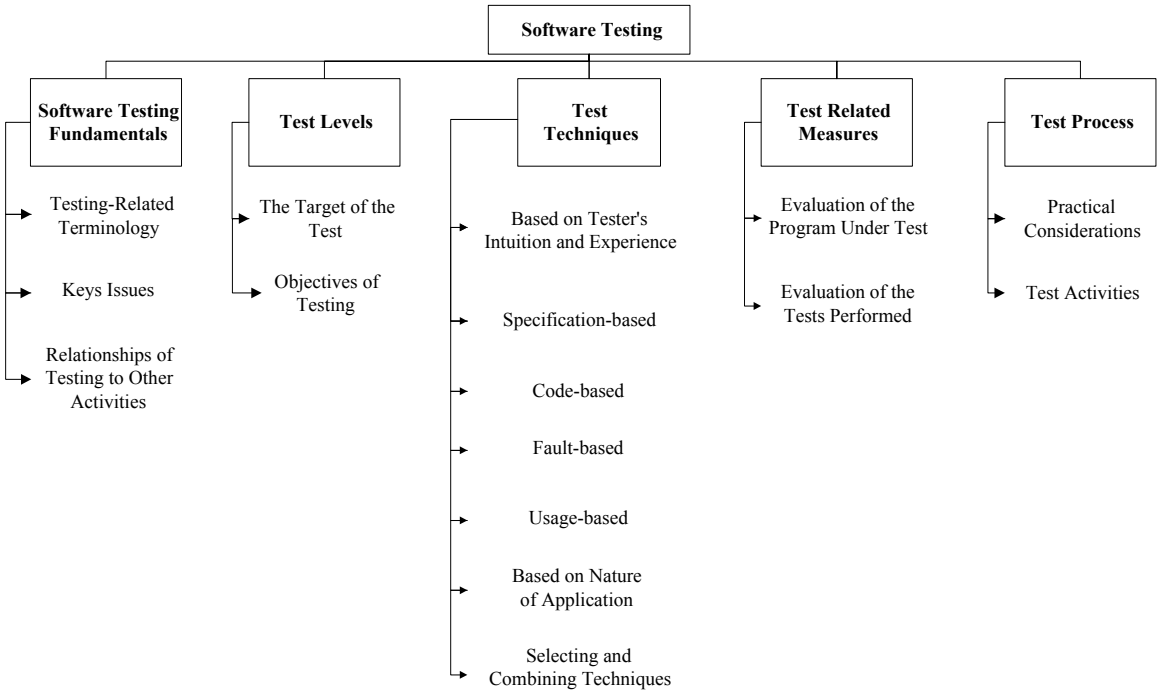


Figure 1 Breakdown of topics for the Software Testing KA

1. Software Testing Fundamentals

1.1. Testing-related terminology

1.1.1. Definitions of testing and related terminology [Bei90:c1; Jor02:c2; Lyu96:c2s2.2] (IEEE610.12-90)

A comprehensive introduction to the Software Testing KA is provided in the recommended references.

1.1.2. Faults vs. Failures

[Jor02:c2; Lyu96:c2s2.2; Per95:c1; Pfl01:c8]

(IEEE610.12-90; IEEE982.1-88)

Many terms are used in the software engineering literature to describe a malfunction, notably *fault*, *failure*, *error*, and several others. This terminology is precisely defined in IEEE Standard 610.12-1990, Standard Glossary of Software Engineering Terminology (IEEE610-90), and is also discussed in the Software Quality KA. It is essential to clearly distinguish between the *cause* of a malfunction, for which the term *fault* or *defect* will be used here, and an undesired effect observed in the system's delivered service, which will be called a *failure*. Testing can reveal failures, but it is the faults that can and must be removed.

However, it should be recognized that the cause of a failure cannot always be unequivocally identified. No theoretical criteria exist to definitively determine what fault caused the observed failure. It might be said that it was the fault that had to be modified to remove the problem, but other modifications could have worked just as well. To avoid

ambiguity, some authors prefer to speak of *failure-causing inputs* (Fra98) instead of faults—that is, those sets of inputs that cause a failure to appear.

1.2. Key issues

1.2.1. Test selection criteria/Test adequacy criteria (or stopping rules)

[Pfl01:c8s7.3; Zhu97:s1.1] (Wey83; Wey91; Zhu97)

A test selection criterion is a means of deciding what a suitable set of test cases should be. A selection criterion can be used for selecting the test cases or for checking whether a selected test suite is adequate—that is, to decide whether the testing can be stopped. See also the sub-topic *Termination*, under topic 5.1 *Practical considerations*.

1.2.2. Testing effectiveness/Objectives for testing [Bei90:c1s1.4; Per95:c21] (Fra98)

Testing is the observation of a sample of program executions. Sample selection can be guided by different objectives: it is only in light of the objective pursued that the effectiveness of the test set can be evaluated.

1.2.3. Testing for defect identification

[Bei90:c1; Kan99:c1]

In testing for defect identification, a successful test is one which causes the system to fail. This is quite different from testing to demonstrate that the software meets its specifications or other desired properties, in which case testing is successful if no (significant) failures are observed.

1.2.4. The oracle problem

[Bei90:c1] (Ber96, Wey83)

An oracle is any (human or mechanical) agent which decides whether a program behaved correctly in a given test, and accordingly produces a verdict of "pass" or "fail." There exist many different kinds of oracles, and oracle automation can be very difficult and expensive.

1.2.5. Theoretical and practical limitations of testing [Kan99:c2] (How76)

Testing theory warns against ascribing an unjustified level of confidence to a series of passed tests. Unfortunately, most established results of testing theory are negative ones, in that they state what testing can never achieve as opposed to what it actually achieved. The most famous quotation in this regard is the Dijkstra aphorism that "program testing can be used to show the presence of bugs, but never to show their absence." The obvious reason is that complete testing is not feasible in real software. Because of this, testing must be driven based on risk and can be seen as a risk management strategy.

1.2.6. The problem of infeasible paths

[Bei90:c3]

Infeasible paths, the control flow paths that cannot be exercised by any input data, are a significant problem in path-oriented testing, and particularly in the automated derivation of test inputs for code-based testing techniques.

1.2.7. Testability

[Bei90:c3, c13] (Bac90; Ber96a; Voa95)

The term "software testability" has two related but different meanings: on the one hand, it refers to the degree to which it is easy for software to fulfill a given test coverage criterion, as in (Bac90); on the other hand, it is defined as the likelihood, possibly measured statistically, that the software will expose a failure under testing, *if* it is faulty, as in (Voa95, Ber96a). Both meanings are important.

1.3. Relationships of testing to other activities

Software testing is related to but different from static software quality management techniques, proofs of correctness, debugging, and programming. However, it is informative to consider testing from the point of view of software quality analysts and of certifiers.

- Testing vs. Static Software Quality Management techniques. See also the Software Quality KA, subarea 2. *Software Quality Management Processes*.

  [Bei90:c1; Per95:c17] (IEEE1008-87)

- Testing vs. Correctness Proofs and Formal Verification [Bei90:c1s5; Pfl01:c8].
- Testing vs. Debugging. See also the Software Construction KA, topic 3.4 *Construction testing* [Bei90:c1s2.1] (IEEE1008-87).
- Testing vs. Programming. See also the Software Construction KA, topic 3.4 *Construction testing* [Bei90:c1s2.3].
- Testing and Certification (Wak99).

2. Test Levels

2.1. The target of the test

Software testing is usually performed at different *levels* along the development and maintenance processes. That is to say, the target of the test can vary: a single module, a group of such modules (related by purpose, use, behavior, or structure), or a whole system. [Bei90:c1; Jor02:c13; Pfl01:c8] Three big test stages can be conceptually distinguished, namely Unit, Integration, and System. No process model is implied, nor are any of those three stages assumed to have greater importance than the other two.

2.1.1. Unit testing

[Bei90:c1; Per95:c17; Pfl01:c8s7.3] (IEEE1008-87)

Unit testing verifies the functioning in isolation of software pieces which are separately testable. Depending on the context, these could be the individual subprograms or a larger component made of tightly related units. A test unit is defined more precisely in the IEEE Standard for Software Unit Testing (IEEE1008-87), which also describes an integrated approach to systematic and documented unit testing. Typically, unit testing occurs with access to the code being tested and with the support of

debugging tools, and might involve the programmers who wrote the code.

2.1.2. Integration testing

[Jor02:c13, 14; Pfl01:c8s7.4]

Integration testing is the process of verifying the interaction between software components. Classical integration testing strategies, such as top-down or bottom-up, are used with traditional, hierarchically structured software.

Modern systematic integration strategies are rather architecture-driven, which implies integrating the software components or subsystems based on identified functional threads. Integration testing is a continuous activity, at each stage of which software engineers must abstract away lower-level perspectives and concentrate on the perspectives of the level they are integrating. Except for small, simple software, systematic, incremental integration testing strategies are usually preferred to putting all the components together at once, which is pictorially called "big bang" testing.

2.1.3. System testing

[Jor02:c15; Pfl01:c9]

System testing is concerned with the behavior of a whole system. The majority of functional failures should already have been identified during unit and integration testing. System testing is usually considered appropriate for comparing the system to the non-functional system requirements, such as security, speed, accuracy, and reliability. External interfaces to other applications, utilities, hardware devices, or the operating environment are also evaluated at this level. See the Software Requirements KA for more information on functional and non-functional requirements.

2.2. Objectives of Testing

[Per95:c8; Pfl01:c9s8.3]

Testing is conducted in view of a specific objective, which is stated more or less explicitly, and with varying degrees of precision. Stating the objective in precise, quantitative terms allows control to be established over the test process.

Testing can be aimed at verifying different properties. Test cases can be designed to check that the functional specifications are correctly implemented, which is variously referred to in the literature as *conformance* testing, *correctness* testing, or *functional* testing. However, several other nonfunctional properties may be tested as well, including performance, reliability, and usability, among many others.

Other important objectives for testing include (but are not limited to) reliability measurement, usability evaluation, and acceptance, for which different approaches would be taken. Note that the test objective varies with the test target; in general, different purposes being addressed at a different level of testing.

References recommended above for this topic describe the set of potential test objectives. The sub-topics listed below are those most often cited in the literature. Note that some kinds of testing are more appropriate for custom-made software packages, *installation* testing, for example; and others for generic products, like *beta* testing.

2.2.1. Acceptance/qualification testing

[Per95:c10; Pfl01:c9s8.5] (IEEE12207.0-96:s5.3.9)

Acceptance testing checks the system behavior against the customer's requirements, however these may have been expressed; the customers undertake, or specify, typical tasks to check that their requirements have been met or that the organization has identified these for the target market for the software. This testing activity may or may not involve the developers of the system.

2.2.2. Installation testing

[Per95:c9; Pfl01:c9s8.6]

Usually after completion of software and acceptance testing, the software can be verified upon installation in the target environment. Installation testing can be viewed as system testing conducted once again according to hardware configuration requirements. Installation procedures may also be verified.

2.2.3. Alpha and beta testing

[Kan99:c13]

Before the software is released, it is sometimes given to a small, representative set of potential users for trial use, either in-house (*alpha* testing) or external (*beta* testing). These users report problems with the product. Alpha and beta use is often uncontrolled, and is not always referred to in a test plan.

2.2.4. Conformance testing/Functional testing/Correctness testing

[Kan99:c7; Per95:c8] (Wak99)

Conformance testing is aimed at validating whether or not the observed behavior of the tested software conforms to its specifications.

2.2.5. Reliability achievement and evaluation

[Lyu96:c7; Pfl01:c9s.8.4] (Pos96)

In helping to identify faults, testing is a means to improve reliability. By contrast, by randomly generating test cases according to the operational profile, statistical measures of reliability can be derived. Using reliability growth models, both objectives can be pursued together (see also sub-topic 4.1.4 *Life test, reliability evaluation*).

2.2.6. Regression testing

[Kan99:c7; Per95:c11, c12; Pfl01:c9s8.1] (Rot96)

According to (IEEE610.12-90), regression testing is the "selective retesting of a system or component to verify that modifications have not caused unintended effects..." In practice, the idea is to show that software which previously

passed the tests still does. Beizer (Bei90) defines it as any repetition of tests intended to show that the software's behavior is unchanged, except insofar as required. Obviously a trade-off must be made between the assurance given by regression testing every time a change is made and the resources required to do that.

Regression testing can be conducted at each of the test levels described in topic 2.1 *The target of the test* and may apply to functional and nonfunctional testing.

2.2.7. Performance testing

[Per95:c17; Pfl01:c9s8.3] (Wak99)

This is specifically aimed at verifying that the software meets the specified performance requirements, for instance, capacity and response time. A specific kind of performance testing is volume testing (Per95:p185, p487; Pfl01:p401), in which internal program or system limitations are tried.

2.2.8. Stress testing

[Per95:c17; Pfl01:c9s8.3]

Stress testing exercises software at the maximum design load, as well as beyond it.

2.2.9. Back-to-back testing

A single test set is performed on two implemented versions of a software product, and the results are compared.

2.2.10. Recovery testing [Per95:c17; Pfl01:c9s8.3]

Recovery testing is aimed at verifying software restart capabilities after a "disaster."

2.2.11. Configuration testing

[Kan99:c8; Pfl01:c9s8.3]

In cases where software is built to serve different users, configuration testing analyzes the software under the various specified configurations.

2.2.12. Usability testing

[Per95:c8; Pfl01:c9s8.3]

This process evaluates how easy it is for end-users to use and learn the software, including user documentation; how effectively the software functions in supporting user tasks; and, finally, its ability to recover from user errors.

2.2.13. Test-driven development

[Bec02]

Test-driven development is not a test technique per se, promoting the use of tests as a surrogate for a requirements specification document rather than as an independent check that the software has correctly implemented the requirements.

3. Test Techniques

One of the aims of testing is to reveal as much potential for failure as possible, and many techniques have been developed to do this, which attempt to "break" the program, by running one or more tests drawn from identified classes of executions deemed equivalent. The leading principle underlying such techniques is to be as systematic as possible in identifying a representative set of program behaviors; for instance, considering subclasses of the input domain, scenarios, states, and dataflow.

It is difficult to find a homogeneous basis for classifying all techniques, and the one used here must be seen as a compromise. The classification is based on how tests are generated from the software engineer's intuition and experience, the specifications, the code structure, the (real or artificial) faults to be discovered, the field usage, or, finally, the nature of the application. Sometimes these techniques are classified as *white-box*, also called *glass-box*, if the tests rely on information about how the software has been designed or coded, or as *black-box* if the test cases rely only on the input/output behavior. One last category deals with combined use of two or more techniques. Obviously, these techniques are not used equally often by all practitioners. Included in the list are those that a software engineer should know.

3.1. Based on the software engineer's intuition and experience

3.1.1. Ad hoc testing

[Kan99:c1]

Perhaps the most widely practiced technique remains ad hoc testing: tests are derived relying on the software engineer's skill, intuition, and experience with similar programs. Ad hoc testing might be useful for identifying special tests, those not easily captured by formalized techniques.

3.1.2. Exploratory testing

Exploratory testing is defined as simultaneous learning, test design, and test execution; that is, the tests are not defined in advance in an established test plan, but are dynamically designed, executed, and modified. The effectiveness of exploratory testing relies on the software engineer's knowledge, which can be derived from various sources: observed product behavior during testing, familiarity with the application, the platform, the failure process, the type of possible faults and failures, the risk associated with a particular product, and so on. [Kan01:c3]

3.2. Specification-based techniques

3.2.1. Equivalence partitioning

[Jor02:c7; Kan99:c7]

The input domain is subdivided into a collection of subsets, or equivalent classes, which are deemed equivalent according to a specified relation, and a representative set of tests (sometimes only one) is taken from each class.

3.2.2. Boundary-value analysis

[Jor02:c6; Kan99:c7]

Test cases are chosen on and near the boundaries of the input domain of variables, with the underlying rationale that many faults tend to concentrate near the extreme values

of inputs. An extension of this technique is *robustness testing*, wherein test cases are also chosen outside the input domain of variables, to test program robustness to unexpected or erroneous inputs.

3.2.3. Decision table

[Bei90:c10s3] (Jor02)

Decision tables represent logical relationships between conditions (roughly, inputs) and actions (roughly, outputs). Test cases are systematically derived by considering every possible combination of conditions and actions. A related technique is *cause-effect graphing*. [Pfl01:c9]

3.2.4. Finite-state machine-based

[Bei90:c11; Jor02:c8]

By modeling a program as a finite state machine, tests can be selected in order to cover states and transitions on it.

3.2.5. Testing from formal specifications

[Zhu97:s2.2] (Ber91; Dic93; Hor95)

Giving the specifications in a formal language allows for automatic derivation of functional test cases, and, at the same time, provides a reference output, an oracle, for checking test results. Methods exist for deriving test cases from model-based (Dic93, Hor95) or algebraic specifications. (Ber91)

3.2.6. Random testing

[Bei90:c13; Kan99:c7]

Tests are generated purely at random, not to be confused with statistical testing from the operational profile as described in sub-topic 3.5.1 *Operational profile*. This form of testing falls under the heading of the specification-based entry, since at least the input domain must be known, to be able to pick random points within it.

3.3. Code-based techniques

3.3.1. Control-flow-based criteria

[Bei90:c3; Jor02:c10] (Zhu97)

Control-flow-based coverage criteria is aimed at covering all the statements or blocks of statements in a program, or specified combinations of them. Several coverage criteria have been proposed, like condition/decision coverage. The strongest of the control-flow-based criteria is path testing, which aims to execute all entry-to-exit control flow paths in the flowgraph. Since path testing is generally not feasible because of loops, other less stringent criteria tend to be used in practice, such as statement testing, branch testing, and condition/decision testing. The adequacy of such tests is measured in percentages; for example, when all branches have been executed at least once by the tests, 100% branch coverage is said to have been achieved.

3.3.2. Data flow-based criteria

[Bei90:c5] (Jor02; Zhu97)

In data-flow-based testing, the control flowgraph is annotated with information about how the program variables are defined, used, and killed (undefined). The strongest criterion, all definition-use paths, requires that, for each variable, every control flow path segment from a definition of that variable to a use of that definition is executed. In order to reduce the number of paths required, weaker strategies such as all-definitions and all-uses are employed.

3.3.3. Reference models for code-based testing (flowgraph, call graph)

[Bei90:c3; Jor02:c5].

Although not a technique in itself, the control structure of a program is graphically represented using a flowgraph in code-based testing techniques. A flowgraph is a directed graph the nodes and arcs of which correspond to program elements. For instance, nodes may represent statements or uninterrupted sequences of statements, and arcs the transfer of control between nodes.

3.4. Fault-based techniques

(Mor90)

With different degrees of formalization, fault-based testing techniques devise test cases specifically aimed at revealing categories of likely or predefined faults.

3.4.1. Error guessing

[Kan99:c7]

In error guessing, test cases are specifically designed by software engineers trying to figure out the most plausible faults in a given program. A good source of information is the history of faults discovered in earlier projects, as well as the software engineer's expertise.

3.4.2. Mutation testing

[Per95:c17; Zhu97:s3.2-s3.3]

A mutant is a slightly modified version of the program under test, differing from it by a small, syntactic change. Every test case exercises both the original and all generated mutants: if a test case is successful in identifying the difference between the program and a mutant, the latter is said to be "killed." Originally conceived as a technique to evaluate a test set (see 4.2), mutation testing is also a testing criterion in itself: either tests are randomly generated until enough mutants have been killed, or tests are specifically designed to kill surviving mutants. In the latter case, mutation testing can also be categorized as a code-based technique. The underlying assumption of mutation testing, the coupling effect, is that by looking for simple syntactic faults, more complex but real faults will be found. For the technique to be effective, a large number of mutants must be automatically derived in a systematic way.

3.5. Usage-based techniques

3.5.1. Operational profile

[Jor02:c15; Lyu96:c5; Pfl01:c9]

In testing for reliability evaluation, the test environment must reproduce the operational environment of the software

as closely as possible. The idea is to infer, from the observed test results, the future reliability of the software when in actual use. To do this, inputs are assigned a probability distribution, or profile, according to their occurrence in actual operation.

3.5.2. Software Reliability Engineered Testing

[Lyu96:c6]

Software Reliability Engineered Testing (SRET) is a testing method encompassing the whole development process, whereby testing is "designed and guided by reliability objectives and expected relative usage and criticality of different functions in the field."

3.6. Techniques based on the nature of the application

The above techniques apply to all types of software. However, for some kinds of applications, some additional know-how is required for test derivation. A list of a few specialized testing fields is provided here, based on the nature of the application under test:

- Object-oriented testing [Jor02:c17; Pfl01:c8s7.5] (Bin00)
- Component-based testing
- Web-based testing
- GUI testing [Jor20]
- Testing of concurrent programs (Car91)
- Protocol conformance testing (Pos96; Boc94)
- Testing of real-time systems (Sch94)
- Testing of safety-critical systems (IEEE1228-94)

3.7. Selecting and combining techniques

3.7.1. Functional and structural

[Bei90:c1s.2.2; Jor02:c2, c9, c12; Per95:c17]

(Pos96)

Specification-based and code-based test techniques are often contrasted as functional vs. structural testing. These two approaches to test selection are not to be seen as alternative but rather as complementary; in fact, they use different sources of information and have proved to highlight different kinds of problems. They could be used in combination, depending on budgetary considerations.

3.7.2. Deterministic vs. random

(Ham92; Lyu96:p541-547)

Test cases can be selected in a deterministic way, according to one of the various techniques listed, or randomly drawn from some distribution of inputs, such as is usually done in reliability testing. Several analytical and empirical comparisons have been conducted to analyze the conditions that make one approach more effective than the other.

4. Test-related measures

Sometimes, test techniques are confused with test objectives. Test techniques are to be viewed as aids which help to ensure the achievement of test objectives. For instance, branch coverage is a popular test technique. Achieving a specified branch coverage measure should not be considered the objective of testing per se: it is a means to improve the chances of finding failures by systematically exercising every program branch out of a decision point. To avoid such misunderstandings, a clear distinction should be made between test-related measures, which provide an evaluation of the program under test based on the observed test outputs, and those which evaluate the thoroughness of the test set. Additional information on measurement programs is provided in the Software Engineering Management KA, subarea 6, *Software engineering measurement*. Additional information on measures can be found in the Software Engineering Process KA, subarea 4, *Process and product measurement*.

Measurement is usually considered instrumental to quality analysis. Measurement may also be used to optimize the planning and execution of the tests. Test management can use several process measures to monitor progress. Measures relative to the test process for management purposes are considered in topic 5.1 *Practical considerations*.

4.1. Evaluation of the program under test (IEEE982.1-98)

4.1.1. Program measurements to aid in planning and designing testing

[Bei90:c7s4.2; Jor02:c9] (Ber96; IEEE982.1-88)

Measures based on program size (for example, source lines of code or function points) or on program structure (like complexity) are used to guide testing. Structural measures can also include measurements among program modules in terms of the frequency with which modules call each other.

4.1.2. Fault types, classification, and statistics

[Bei90:c2; Jor02:c2; Pfl01:c8]

(Bei90; IEEE1044-93; Kan99; Lyu96)

The testing literature is rich in classifications and taxonomies of faults. To make testing more effective, it is important to know which types of faults could be found in the software under test, and the relative frequency with which these faults have occurred in the past. This information can be very useful in making quality predictions, as well as for process improvement. More information can be found in the Software Quality KA, topic 3.2 *Defect characterization*. An IEEE standard exists on how to classify software "anomalies" (IEEE1044-93).

4.1.3. Fault density

[Per95:c20] (IEEE982.1-88; Lyu96:c9)

A program under test can be assessed by counting and classifying the discovered faults by their types. For each fault class, fault density is measured as the ratio between the number of faults found and the size of the program.

4.1.4. Life test, reliability evaluation

[Pfl01:c9] (Pos96:p146-154)

A statistical estimate of software reliability, which can be obtained by reliability achievement and evaluation (see sub-topic 2.2.5), can be used to evaluate a product and decide whether or not testing can be stopped.

4.1.5. Reliability growth models

[Lyu96:c7; Pfl01:c9] (Lyu96:c3, c4)

Reliability growth models provide a prediction of reliability based on the failures observed under reliability achievement and evaluation (see sub-topic 2.2.5). They assume, in general, that the faults that caused the observed failures have been fixed (although some models also accept imperfect fixes), and thus, on average, the product's reliability exhibits an increasing trend. There now exist dozens of published models. Many are laid down on some common assumptions, while others differ. Notably, these models are divided into *failure-count* and *time-between-failure* models.

4.2. Evaluation of the tests performed

4.2.1. Coverage/thoroughness measures

[Jor02:c9; Pfl01:c8] (IEEE982.1-88)

Several test adequacy criteria require that the test cases systematically exercise a set of elements identified in the program or in the specifications (see subarea 3). To evaluate the thoroughness of the executed tests, testers can monitor the elements covered, so that they can dynamically measure the ratio between covered elements and their total number. For example, it is possible to measure the percentage of covered branches in the program flowgraph, or that of the functional requirements exercised among those listed in the specifications document. Code-based adequacy criteria require appropriate instrumentation of the program under test.

4.2.2. Fault seeding

[Pfl01:c8] (Zhu97:s3.1)

Some faults are artificially introduced into the program before test. When the tests are executed, some of these seeded faults will be revealed, and possibly some faults which were already there will be as well. In theory, depending on which of the artificial faults are discovered, and how many, testing effectiveness can be evaluated, and the remaining number of genuine faults can be estimated. In practice, statisticians question the distribution and representativeness of seeded faults relative to genuine faults and the small sample size on which any extrapolations are based. Some also argue that this technique should be used with great care, since inserting faults into software involves the obvious risk of leaving them there.

4.2.3. Mutation score

[Zhu97:s3.2-s3.3]

In mutation testing (see sub-topic 3.4.2), the ratio of killed mutants to the total number of generated mutants can be a measure of the effectiveness of the executed test set.

4.2.4. Comparison and relative effectiveness of different techniques

[Jor02:c9, c12; Per95:c17; Zhu97:s5] (Fra93; Fra98; Pos96: p64-72)

Several studies have been conducted to compare the relative effectiveness of different test techniques. It is important to be precise as to the property against which the techniques are being assessed; what, for instance, is the exact meaning given to the term "effectiveness"? Possible interpretations are: the number of tests needed to find the first failure, the ratio of the number of faults found through testing to all the faults found during and after testing, or how much reliability was improved. Analytical and empirical comparisons between different techniques have been conducted according to each of the notions of effectiveness specified above.

5. Test Process

Testing concepts, strategies, techniques, and measures need to be integrated into a defined and controlled process which is run by people. The test process supports testing activities and provides guidance to testing teams, from test planning to test output evaluation, in such a way as to provide justified assurance that the test objectives will be met cost-effectively.

5.1. Practical considerations

5.1.1. Attitudes/Egoless programming

[Bei90:c13s3.2; Pfl01:c8]

A very important component of successful testing is a collaborative attitude towards testing and quality assurance activities. Managers have a key role in fostering a generally favorable reception towards failure discovery during development and maintenance; for instance, by preventing a mindset of code ownership among programmers, so that they will not feel responsible for failures revealed by their code.

5.1.2. Test guides

[Kan01]

The testing phases could be guided by various aims, for example: in risk-based testing, which uses the product risks to prioritize and focus the test strategy; or in scenario-based testing, in which test cases are defined based on specified software scenarios.

5.1.3. Test process management

[Bec02: III; Per95:c1-c4; Pfl01:c9] (IEEE1074-97; IEEE12207.0-96:s5.3.9, s5.4.2, s6.4, s6.5)

Test activities conducted at different levels (see subarea 2. *Test levels*) must be organized, together with people, tools, policies, and measurements, into a well-defined process which is an integral part of the life cycle. In IEEE/EIA Standard 12207.0, testing is not described as a stand-alone process, but principles for testing activities are included along with both the five primary life cycle processes and the supporting process. In IEEE Std 1074, testing is grouped with other evaluation activities as integral to the entire life cycle.

5.1.4. Test documentation and work products [Bei90:c13s5; Kan99:c12; Per95:c19; Pfl01:c9s8.8] (IEEE829-98)

Documentation is an integral part of the formalization of the test process. The IEEE Standard for Software Test Documentation (IEEE829-98) provides a good description of test documents and of their relationship with one another and with the testing process. Test documents may include, among others, Test Plan, Test Design Specification, Test Procedure Specification, Test Case Specification, Test Log, and Test Incident or Problem Report. The software under test is documented as the Test Item. Test documentation should be produced and continually updated, to the same level of quality as other types of documentation in software engineering.

5.1.5. Internal vs. independent test team

[Bei90:c13s2.2-c13s2.3; Kan99:c15; Per95:c4; Pfl01:c9]

Formalization of the test process may involve formalizing the test team organization as well. The test team can be composed of internal members (that is, on the project team, involved or not in software construction), of external members, in the hope of bringing in an unbiased, independent perspective, or, finally, of both internal and external members. Considerations of costs, schedule, maturity levels of the involved organizations, and criticality of the application may determine the decision.

5.1.6. Cost/effort estimation and other process measures [Per95:c4, c21] (Per95: Appendix B; Pos96:p139-145; IEEE982.1-88)

Several measures related to the resources spent on testing, as well as to the relative fault-finding effectiveness of the various test phases, are used by managers to control and improve the test process. These test measures may cover such aspects as number of test cases specified, number of test cases executed, number of test cases passed, and number of test cases failed, among others.

Evaluation of test phase reports can be combined with root-cause analysis to evaluate test process effectiveness in finding faults as early as possible. Such an evaluation could be associated with the analysis of risks. Moreover, the resources that are worth spending on testing should be commensurate with the use/criticality of the application: different techniques have different costs and yield different levels of confidence in product reliability.

5.1.7. Termination

[Bei90:c2s2.4; Per95:c2]

A decision must be made as to how much testing is enough and when a test stage can be terminated. Thoroughness measures, such as achieved code coverage or functional completeness, as well as estimates of fault density or of operational reliability, provide useful support, but are not sufficient in themselves. The decision also involves considerations about the costs and risks incurred by the potential for remaining failures, as opposed to the costs implied by continuing to test. See also sub-topic 1.2.1 *Test selection criteria/Test adequacy criteria*.

5.1.8. Test reuse and test patterns

[Bei90:c13s5]

To carry out testing or maintenance in an organized and cost-effective way, the means used to test each part of the software should be reused systematically. This repository of test materials must be under the control of software configuration management, so that changes to software requirements or design can be reflected in changes to the scope of the tests conducted.

The test solutions adopted for testing some application types under certain circumstances, with the motivations behind the decisions taken, form a test pattern which can itself be documented for later reuse in similar projects.

5.2. Test Activities

Under this topic, a brief overview of test activities is given; as often implied by the following description, successful management of test activities strongly depends on the Software Configuration Management process.

5.2.1. Planning

[Kan99:c12; Per95:c19; Pfl01:c8s7.6] (IEEE829-98:s4; IEEE1008-87:s1-s3)

Like any other aspect of project management, testing activities must be planned. Key aspects of test planning include coordination of personnel, management of available test facilities and equipment (which may include magnetic media, test plans and procedures), and planning for possible undesirable outcomes. If more than one baseline of the software is being maintained, then a major planning consideration is the time and effort needed to ensure that the test environment is set to the proper configuration.

5.2.2. Test-case generation

[Kan99:c7] (Pos96:c2; IEEE1008-87:s4, s5)

Generation of test cases is based on the level of testing to be performed and the particular testing techniques. Test

cases should be under the control of software configuration management and include the expected results for each test.

5.2.3. Test environment development

[Kan99:c11]

The environment used for testing should be compatible with the software engineering tools. It should facilitate development and control of test cases, as well as logging and recovery of expected results, scripts, and other testing materials.

5.2.4. Execution

[Bei90:c13; Kan99:c11] (IEEE1008-87:s6, s7)

Execution of tests should embody a basic principle of scientific experimentation: everything done during testing should be performed and documented clearly enough that another person could replicate the results. Hence, testing should be performed in accordance with documented procedures using a clearly defined version of the software under test.

5.2.5. Test results evaluation

[Per95:c20,c21] (Pos96:p18-20, p131-138)

The results of testing must be evaluated to determine whether or not the test has been successful. In most cases, "successful" means that the software performed as expected and did not have any major unexpected outcomes. Not all unexpected outcomes are necessarily faults, however, but could be judged to be simply noise. Before a failure can be removed, an analysis and debugging effort is needed to isolate, identify, and describe it. When test results are particularly important, a formal review board may be convened to evaluate them.

5.2.6. Problem reporting/Test log

[Kan99:c5; Per95:c20] (IEEE829-98:s9-s10)

Testing activities can be entered into a test log to identify when a test was conducted, who performed the test, what software configuration was the basis for testing, and other relevant identification information. Unexpected or incorrect test results can be recorded in a problem-reporting system, the data of which form the basis for later debugging and for fixing the problems that were observed as failures during testing. Also, anomalies not classified as faults could be documented in case they later turn out to be more serious than first thought. Test reports are also an input to the change management request process (see the Software Configuration Management KA, subarea 3, *Software configuration control*).

5.2.7. Defect tracking

[Kan99:c6]

Failures observed during testing are most often due to faults or defects in the software. Such defects can be analyzed to determine when they were introduced into the software, what kind of error caused them to be created (poorly defined requirements, incorrect variable declaration, memory leak, programming syntax error, for example), and when they could have been first observed in the software. Defect-tracking information is used to determine what aspects of software engineering need improvement and how effective previous analyses and testing have been.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	[Bec02]	[Bei90]	[Jor02]	[Kan99	[Kan01]	[Lyu96]	[Per95]	[Pfl01]	[Zhu97]
1. Software Testing Fundamentals									
1.1 Testing-Related Terminology									
Definitions of testing and related terminology		c1	c2			c2s2.2			
Faults vs. failures			c2			c2s2.2	c1	c8	
1.2 Key Issues									
Test selection criteria / test adequacy criteria (or stopping rules)								c8s7.3	s1.1
Testing effectiveness/ objectives for testing		c1s1.4					c21		
Testing for defect identification		c1		c1					
The oracle problem		c1							
Theoretical and practical limitations of testing				c2					
The problem of infeasible paths		c3							
Testability		c3,c13							
1.3 Relationships of Testing to other Activities									
Testing vs. static analysis techniques		c1					c17		
Testing vs. correctness proofs and formal verification		c1s5						c8	
Testing vs. debugging		c1s2.1							
Testing vs. programming		c1s2.3							
Testing and certification									
2. Test Levels									
2.1 The Target of the Tests		c1	c13					c8	
Unit testing		c1					c17	c8s7.3	
Integration testing			c13,c14					c8s7.4	
System testing			c15					c9	
2.2 Objectives of Testing							c8	c9s8.3	
Acceptance/qualification testing							c10	c9s8.5	
Installation testing							c9	c9s8.6	
Alpha and beta testing				c13					
Conformance testing / Functional testing/ Correctness testing				c7			c8		
Reliability achievement and evaluation by testing						c7		c9s8.4	
Regression testing				c7			c11,c12	c9s8.1	
Performance testing							c17	c9s8.3	
Stress testing							c17	c9s8.3	
Back-to-back testing									
Recovery testing							c17	c9s8.3	
Configuration testing				c8				c9s8.3	
Usability testing							c8	c9s8.3	
Test-driven development	III								

	[Bec02]	[Bei90]	[Jor02]	[Kan99]	[Kan01]	[Lyu96]	[Per95]	[Pfl01]	[Zhu97]
3. Test Techniques									
3.1 Based on tester's intuition and experience									
Ad hoc testing				c1					
Exploratory testing					c3				
3.2 Specification-based									
Equivalence partitioning			c7	c7					
Boundary-value analysis			c6	c7					
Decision table		c10s3						c9	
Finite-state machine-based		c11	c8						
Testing from formal specifications									s2.2
Random testing		c13		c7					
3.3 Code-based									
Control-flow-based criteria		c3	c10					c8	
Data-flow-based criteria		c5							
Reference models for code-based testing		c3	c5						
3.4 Fault-based									
Error guessing				c7					
Mutation testing							c17		s3.2, s3.3
3.5 Usage-based									
Operational profile			c15			c5		c9	
Software Reliability Engineered Testing						c6			
3.6 Based on Nature of Application									
Object-oriented testing			c17					c8s7.5	
Component-based testing									
Web-based testing									
GUI testing			c20						
Testing of concurrent programs									
Protocol conformance testing									
Testing of distributed systems									
Testing of real-time systems									
3.7 Selecting and Combining Techniques									
Functional and structural		c1s2.2	c1,c11s11.3				c17		
Deterministic vs. Random									

	[Bec02]	[Bei90]	[Jor02]	[Kan99	[Kan01]	[Lyu96]	[Per95]	[Pfl01]	[Zhu97]
4. Test-Related Measures									
4.1 Evaluation of the Program under Test									
Program measurements to aid in planning and designing testing.		c7s4.2	c9						
Types, classification and statistics of faults		c2	c1					c8	
Fault density							c20		
Life test, reliability evaluation								c9	
Reliability growth models						c7		c9	
4.2 Evaluation of the Tests Performed									
Coverage/thoroughness measures			c9					c8	
Fault seeding								c8	
Mutation score									s3.2, s3.3
Comparison and relative effectiveness of different techniques			c8,c11				c17		s5
5. Test Process									
5.1 Practical Considerations									
Attitudes/Egoless programming		c13s3.2						c8	
Test guides	III				C5				
Test process management							c1-c4	c9	
Test documentation and work products		c13s5		c12			c19	c9s8.8	
Internal vs. independent test team		c13s2.2, c1s2.3		c15			c4	c9	
Cost/effort estimation and other process measures							c4,c21		
Termination		c2s2.4					c2		
Test reuse and test patterns		c13s5							
5.2 Test Activities									
Planning				c12			c19	c87s7.6	
Test case generation				c7					
Test environment development				c11					
Execution		c13		c11					
Test results evaluation							c20,c21		
Problem reporting/Test log				c5			c20		
Defect tracking				c6					

RECOMMENDED REFERENCES FOR SOFTWARE TESTING

[Bec02] K. Beck, *Test-Driven Development by Example*, Addison-Wesley, 2002.

[Bei90] B. Beizer, *Software Testing Techniques*, International Thomson Press, 1990, Chap. 1-3, 5, 7s4, 10s3, 11, 13.

[Jor02] P. C. Jorgensen, *Software Testing: A Craftsman's Approach*, second edition, CRC Press, 2004, Chap. 2, 5-10, 12-15, 17, 20.

[Kan99] C. Kaner, J. Falk, and H.Q. Nguyen, Testing Computer Software, second ed., John Wiley & Sons, 1999, Chaps. 1, 2, 5-8, 11-13, 15.

[Kan01] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing*, Wiley Computer Publishing, 2001.

[Lyu96] M.R. Lyu, *Handbook of Software Reliability Engineering*, Mc-Graw-Hill/IEEE, 1996, Chap. 2s2.2, 5-7.

[Per95] W. Perry, *Effective Methods for Software Testing*, John Wiley & Sons, 1995, Chap. 1-4, 9, 10-12, 17, 19-21.

[Pfl01] S. L. Pfleeger, *Software Engineering: Theory and Practice*, second ed., Prentice Hall, 2001, Chap. 8, 9.

[Zhu97] H. Zhu, P.A.V. Hall and J.H.R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, iss. 4 (Sections 1, 2.2, 3.2, 3.3), Dec. 1997, pp. 366-427.

APPENDIX A. LIST OF FURTHER READINGS

(Bac90) R. Bache and M. Müllerburg, "Measures of Testability as a Basis for Quality Assurance," *Software Engineering Journal*, vol. 5, March 1990, pp. 86-92.

(Bei90) B. Beizer, *Software Testing Techniques*, International Thomson Press, second ed., 1990.

(Ber91) G. Bernot, M.C. Gaudel and B. Marre, "Software Testing Based On Formal Specifications: a Theory and a Tool," *Software Engineering Journal*, Nov. 1991, pp. 387-405.

(Ber96) A. Bertolino and M. Marrè, "How Many Paths Are Needed for Branch Testing?" *Journal of Systems and Software*, vol. 35, iss. 2, 1996, pp. 95-106.

(Ber96a) A. Bertolino and L. Strigini, "On the Use of Testability Measures for Dependability Assessment," *IEEE Transactions on Software Engineering*, vol. 22, iss. 2, Feb. 1996, pp. 97-108.

(Bin00) R.V. Binder, *Testing Object-Oriented Systems Models, Patterns, and Tools*, Addison-Wesley, 2000.

(Boc94) G.V. Bochmann and A. Petrenko, "Protocol Testing: Review of Methods and Relevance for Software Testing," presented at *ACM Proc. Int'l Symp. on Software Testing and Analysis* (ISSTA '94), Seattle, Wash., 1994.

(Car91) R.H. Carver and K.C. Tai, "Replay and Testing for Concurrent Programs," *IEEE Software*, March 1991, pp. 66-74.

(Dic93) J. Dick and A. Faivre, "Automating the Generation and Sequencing of Test Cases from Model-Based Specifications," presented at *FME '93: Industrial-Strength Formal Methods*, LNCS 670, Springer-Verlag, 1993.

(Fran93) P. Frankl and E. Weyuker, "A Formal Analysis of the Fault Detecting Ability of Testing Methods," *IEEE Transactions on Software Engineering*, vol. 19, iss. 3, March 1993, p. 202.

(Fran98) P. Frankl, D. Hamlet, B. Littlewood, and L. Strigini, "Evaluating Testing Methods by Delivered Reliability," *IEEE Transactions on Software Engineering*, vol. 24, iss. 8, August 1998, pp. 586-601.

(Ham92) D. Hamlet, "Are We Testing for True Reliability?" *IEEE Software*, July 1992, pp. 21-27.

(Hor95) H. Horcher and J. Peleska, "Using Formal Specifications to Support Software Testing," *Software Quality Journal*, vol. 4, 1995, pp. 309-327.

(How76) W. E. Howden, "Reliability of the Path Analysis Testing Strategy," *IEEE Transactions on Software Engineering*, vol. 2, iss. 3, Sept. 1976, pp. 208-215.

(Jor02) P.C. Jorgensen, *Software Testing: A Craftsman's Approach*, second ed., CRC Press, 2004.

(Kan99) C. Kaner, J. Falk, and H.Q. Nguyen, "Testing Computer Software," second ed., John Wiley & Sons, 1999.

(Lyu96) M.R. Lyu, *Handbook of Software Reliability Engineering*, Mc-Graw-Hill/IEEE, 1996.

(Mor90) L.J. Morell, "A Theory of Fault-Based Testing," *IEEE Transactions on Software Engineering*, vol. 16, iss. 8, August 1990, pp. 844-857.

(Ost88) T.J. Ostrand and M.J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," *Communications of the ACM*, vol. 31, iss. 3, June 1988, pp. 676-686.

(Ost98) T. Ostrand, A. Anodide, H. Foster, and T. Goradia, "A Visual Test Development Environment for GUI Systems," presented at *ACM Proc. Int'l Symp. on Software Testing and Analysis* (ISSTA '98), Clearwater Beach, Florida, 1998.

(Per95) W. Perry, *Effective Methods for Software Testing*, John Wiley & Sons, 1995.

(Pfl01) S.L. Pfleeger, *Software Engineering: Theory and Practice*, second ed., Prentice-Hall, 2001, Chap. 8, 9.

(Pos96) R.M. Poston, *Automating Specification-Based Software Testing*, IEEE, 1996.

(Rot96) G. Rothermel and M.J. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE Transactions on Software Engineering*, vol. 22, iss. 8, Aug. 1996, p. 529.

(Sch94) W. Schütz, "Fundamental Issues in Testing Distributed Real-Time Systems," *Real-Time Systems Journal*, vol. 7, iss. 2, Sept. 1994, pp. 129-157.

(Voa95) J.M. Voas and K.W. Miller, "Software Testability: The New Verification," *IEEE Software*, May 1995, pp. 17-28.

(Wak99) S. Wakid, D.R. Kuhn, and D.R. Wallace, "Toward Credible IT Testing and Certification," *IEEE Software*, July-Aug. 1999, pp. 39-47.

(Wey82) E.J. Weyuker, "On Testing Non-testable Programs," *The Computer Journal*, vol. 25, iss. 4, 1982, pp. 465-470.

(Wey83) E.J. Weyuker, "Assessing Test Data Adequacy through Program Inference," *ACM Trans. on Programming Languages and Systems*, vol. 5, iss. 4, October 1983, pp. 641-655.

(Wey91) E.J. Weyuker, S.N. Weiss, and D. Hamlet, "Comparison of Program Test Strategies," presented at *Proc. Symp. on Testing, Analysis and Verification (TAV 4)*, Victoria, British Columbia, 1991.

(Zhu97) H. Zhu, P.A.V. Hall, and J.H.R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, iss. 4, Dec. 1997, pp. 366-427.

APPENDIX B. LIST OF STANDARDS

(IEEE610.12-90) IEEE Std 610.12-1990 (R2002), *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1990.

(IEEE829-98) IEEE Std 829-1998, *Standard for Software Test Documentation*, IEEE, 1998.

(IEEE982.1-88) IEEE Std 982.1-1988, *IEEE Standard Dictionary of Measures to Produce Reliable Software*, IEEE, 1988.

(IEEE1008-87) IEEE Std 1008-1987 (R2003), *IEEE Standard for Software Unit Testing*, IEEE, 1987.

(IEEE1044-93) IEEE Std 1044-1993 (R2002), *IEEE Standard for the Classification of Software Anomalies*, IEEE, 1993.

(IEEE1228-94) IEEE Std 1228-1994, *Standard for Software Safety Plans*, IEEE, 1994.

(IEEE12207.0-96) IEEE/EIA 12207.0-1996 // ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.

CHAPTER 6

SOFTWARE MAINTENANCE

ACRONYMS

CMMI	Capability Maturity Model Integration
ICSM	International Conference on Software Maintenance
SCM	Software Configuration Management
SQA	Software Quality Assurance
V&V	Verification and Validation
Y2K	Year 2000

INTRODUCTION

Software development efforts result in the delivery of a software product which satisfies user requirements. Accordingly, the software product must change or evolve. Once in operation, defects are uncovered, operating environments change, and new user requirements surface. The maintenance phase of the life cycle begins following a warranty period or post-implementation support delivery, but maintenance activities occur much earlier.

Software maintenance is an integral part of a software life cycle. However, it has not, historically, received the same degree of attention that the other phases have. Historically, software development has had a much higher profile than software maintenance in most organizations. This is now changing, as organizations strive to squeeze the most out of their software development investment by keeping software operating as long as possible. Concerns about the Year 2000 (Y2K) rollover focused significant attention on the software maintenance phase, and the Open Source paradigm has brought further attention to the issue of maintaining software artifacts developed by others.

In the Guide, software maintenance is defined as the totality of activities required to provide cost-effective support to software. Activities are performed during the pre-delivery stage, as well as during the post-delivery stage. Pre-delivery activities include planning for post-delivery operations, for maintainability, and for logistics determination for transition activities. Post-delivery activities include software modification, training, and operating or interfacing to a help desk.

The Software Maintenance KA is related to all other aspects of software engineering. Therefore, this KA description is linked to all other chapters of the Guide.

BREAKDOWN OF TOPICS FOR SOFTWARE MAINTENANCE

The Software Maintenance KA breakdown of topics is shown in Figure 1.

1. Software Maintenance Fundamentals

This first section introduces the concepts and terminology that form an underlying basis to understanding the role and scope of software maintenance. The topics provide definitions and emphasize why there is a need for maintenance. Categories of software maintenance are critical to understanding its underlying meaning.

1.1. Definitions and Terminology

[IEEE1219-98:s3.1.12; IEEE12207.0-96:s3.1,s5.5; ISO14764-99:s6.1]

Software maintenance is defined in the IEEE Standard for Software Maintenance, IEEE 1219, as the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment. The standard also addresses maintenance activities prior to delivery of the software product, but only in an information appendix of the standard.

The IEEE/EIA 12207 standard for software life cycle processes essentially depicts maintenance as one of the primary life cycle processes, and describes maintenance as the process of a software product undergoing "modification to code and associated documentation due to a problem or the need for improvement. The objective is to modify the existing software product while preserving its integrity." ISO/IEC 14764, the international standard for software maintenance, defines software maintenance in the same terms as IEEE/EIA 12207 and emphasizes the pre-delivery aspects of maintenance, planning, for example.

1.2. Nature of Maintenance

[Pfl01:c11s11.2]

Software maintenance sustains the software product throughout its operational life cycle. Modification requests are logged and tracked, the impact of proposed changes is determined, code and other software artifacts are modified, testing is conducted, and a new version of the software product is released. Also, training and daily support are provided to users. Pfleeger [Pfl01] states that "maintenance has a broader scope, with more to track and control" than development.

A maintainer is defined by IEEE/EIA 12207 as an organization which performs maintenance activities [IEEE12207.0-96]. In this KA, the term will sometimes refer to individuals who perform those activities, contrasting them with the developers.

IEEE/EIA 12207 identifies the primary activities of software maintenance as: process implementation; problem and modification analysis; modification implementation; maintenance review/acceptance; migration; and retirement. These activities are discussed in topic 3.2 *Maintenance Activities*.

Maintainers can learn from the developer's knowledge of the software. Contact with the developers and early involvement by the maintainer helps reduce the maintenance effort. In some instances, the software engineer cannot be reached or has moved on to other tasks, which creates an additional challenge for the maintainers. Maintenance must take the products of the development, code, or documentation, for example, and support them immediately and evolve/maintain them progressively over the software life cycle.

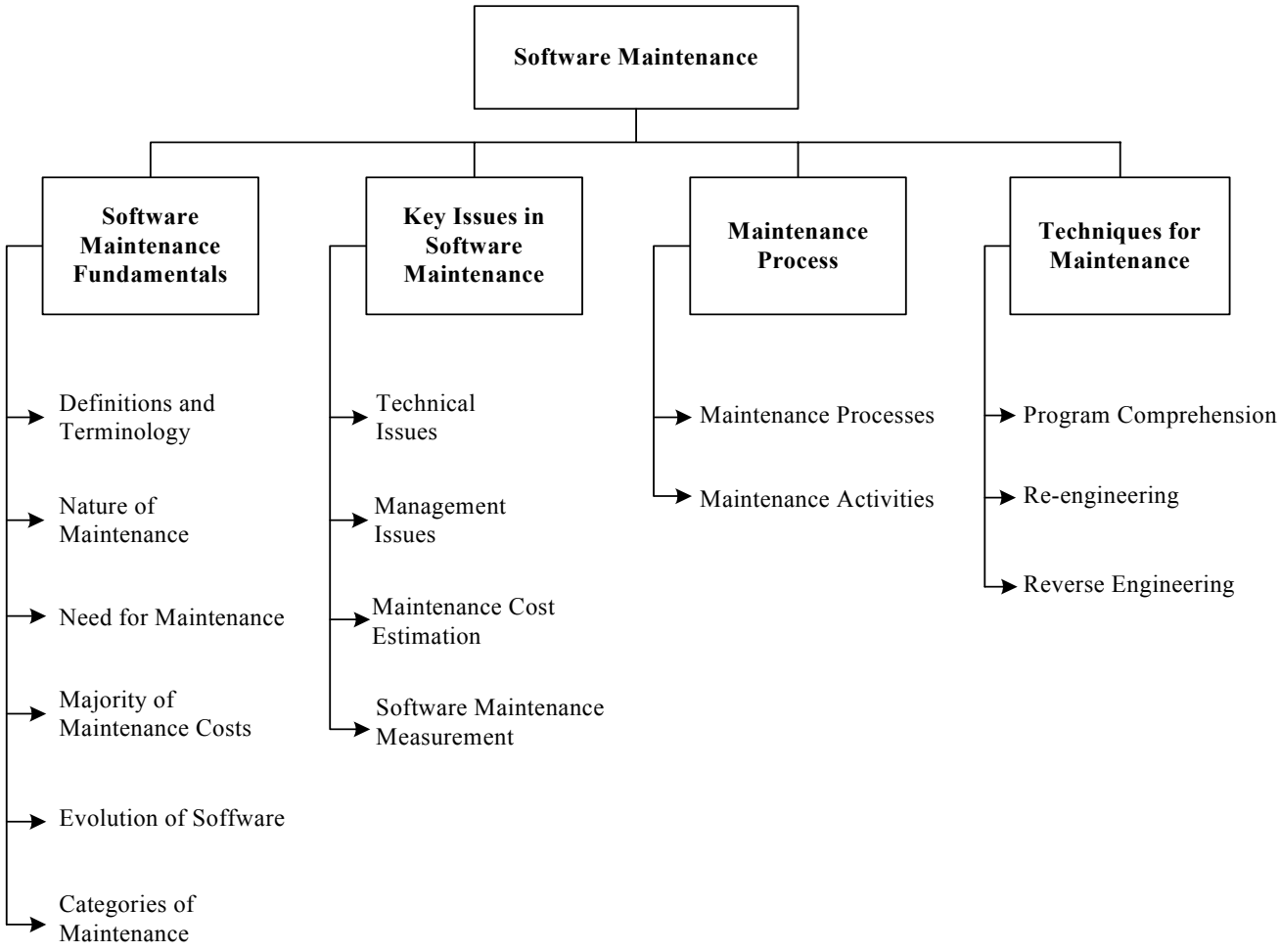


Figure 1 Breakdown of topics for the Software Maintenance KA

1.3. Need for Maintenance

[Pfl01:c11s11.2; Pig97: c2s2.3; Tak97:c1]

Maintenance is needed to ensure that the software continues to satisfy user requirements. Maintenance is applicable to software developed using any software life cycle model (for example, spiral). The system changes due to corrective and non-corrective software actions. Maintenance must be performed in order to:

- Correct faults
- Improve the design
- Implement enhancements

- Interface with other systems
- Adapt programs so that different hardware, software, system features, and telecommunications facilities can be used
- Migrate legacy software
- Retire software

The maintainer's activities comprise four key characteristics, according to Pfleeger [Pfl01]:

- Maintaining control over the software's day-to-day functions
- Maintaining control over software modification
- Perfecting existing functions
- Preventing software performance from degrading to unacceptable levels

1.4. Majority of Maintenance Costs

[Abr93:63-90; Pfl01:c11s11.3; Pig97:c3; Pre01:c30s2.1,c30s2.2]

Maintenance consumes a major share of software life cycle financial resources. A common perception of software maintenance is that it merely fixes faults. However, studies and surveys over the years have indicated that the majority, over 80%, of the software maintenance effort is used for non-corrective actions. [Abr93, Pig97, Pre01] Jones (Jon91) describes the way in which software maintenance managers often group enhancements and corrections together in their management reports. This inclusion of enhancement requests with problem reports contributes to some of the misconceptions regarding the high cost of corrections. Understanding the categories of software maintenance helps to understand the structure of software maintenance costs. Also, understanding the factors that influence the maintainability of a system can help to contain costs. Pfleeger [Pfl01] presents some of the technical and non-technical factors affecting software maintenance costs, as follows:

- Application type
- Software novelty
- Software maintenance staff availability
- Software life span
- Hardware characteristics
- Quality of software design, construction, documentation and testing

1.5. Evolution of Software

[Art88:c1s1.0,s1.1,s1.2,c11s1.1,s1.2; Leh97:108-124], (Bel72)

Lehman first addressed software maintenance and evolution of systems in 1969. Over a period of twenty years, his research led to the formulation of eight "Laws of Evolution". [Leh97] Key findings include the fact that maintenance is evolutionary developments, and that maintenance decisions are aided by understanding what happens to systems (and software) over time. Others state that maintenance is continued development, except that there is an extra input (or constraint)–existing large software is never complete and continues to evolve. As it evolves, it grows more complex unless some action is taken to reduce this complexity.

Since software demonstrates regular behavior and trends, these can be measured. Attempts to develop predictive models to estimate maintenance effort have been made, and, as a result, useful management tools have been developed. [Art88], (Bel72)

1.6. Categories of Maintenance

[Art88:c1s1.2; Lie78; Dor02:v1c9s1.5; IEEE1219-98:s3.1.1,s3.1.2,s3.1.7,A.1.7; ISO14764-99:s4.1,s4.3,s4.10, s4.11,s6.2; Pig97:c2s2.3]

Lientz & Swanson initially defined three categories of maintenance: corrective, adaptive, and perfective. [Lie78; IEEE1219-98] This definition was later updated in the Standard for Software Engineering-Software Maintenance, ISO/IEC 14764 to include four categories, as follows:

- Corrective maintenance: Reactive modification of a software product performed after delivery to correct discovered problems
- Adaptive maintenance: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment
- Perfective maintenance: Modification of a software product after delivery to improve performance or maintainability
- Preventive maintenance: Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults

ISO/IEC 14764 classifies adaptive and perfective maintenance as enhancements. It also groups together the corrective and preventive maintenance categories into a correction category, as shown in Table 1. Preventive maintenance, the newest category, is most often performed on software products where safety is critical.

	Correction	Enhancement
Proactive	Preventive	Perfective
Reactive	Corrective	Adaptive

Table 1: Software maintenance categories

2. Key Issues in Software Maintenance

A number of key issues must be dealt with to ensure the effective maintenance of software. It is important to

understand that software maintenance provides unique technical and management challenges for software engineers. Trying to find a fault in software containing 500K lines of code that the software engineer did not develop is a good example. Similarly, competing with software developers for resources is a constant battle. Planning for a future release, while coding the next release and sending out emergency patches for the current release, also creates a challenge. The following section presents some of the technical and management issues related to software maintenance. They have been grouped under the following topic headings:

- Technical issues
- Management issues
- Cost estimation and
- Measures

2.1. Technical Issues

2.1.1. Limited understanding
[Dor02:v1c9s1.11.4; Pfl01:c11s11.3; Tak97:c3]

Limited understanding refers to how quickly a software engineer can understand where to make a change or a correction in software which this individual did not develop. Research indicates that some 40% to 60% of the maintenance effort is devoted to understanding the software to be modified. Thus, the topic of software comprehension is of great interest to software engineers. Comprehension is more difficult in text-oriented representation, in source code, for example, where it is often difficult to trace the evolution of software through its releases/versions if changes are not documented and when the developers are not available to explain it, which is often the case. Thus, software engineers may initially have a limited understanding of the software, and much has to be done to remedy this.

2.1.2. Testing
[Art88:c9; Pfl01:c11s11.3]

The cost of repeating full testing on a major piece of software can be significant in terms of time and money. Regression testing, the selective retesting of a software or component to verify that the modifications have not caused unintended effects, is important to maintenance. As well, finding time to test is often difficult. There is also the challenge of coordinating tests when different members of the maintenance team are working on different problems at the same time. [Plf01] When software performs critical functions, it may be impossible to bring it offline to test. The Software Testing KA provides additional information and references on the matter in its sub-topic 2.2.6 Regression testing.

2.1.3. Impact analysis
[Art88:c3; Dor02:v1c9s1.10; Pfl01: c11s11.5]

Impact analysis describes how to conduct, cost effectively, a complete analysis of the impact of a change in existing software. Maintainers must possess an intimate knowledge of the software's structure and content [Pfl01]. They use that knowledge to perform impact analysis, which identifies all systems and software products affected by a software change request and develops an estimate of the resources needed to accomplish the change. [Art88] Additionally, the risk of making the change is determined. The change request, sometimes called a modification request (MR) and often called a problem report (PR), must first be analyzed and translated into software terms. [Dor02] It is performed after a change request enters the software configuration management process. Arthur [Art88] states that the objectives of impact analysis are:

- Determination of the scope of a change in order to plan and implement work
- Development of accurate estimates of resources needed to perform the work
- Analysis of the cost/benefits of the requested change
- Communication to others of the complexity of a given change

The severity of a problem is often used to decide how and when a problem will be fixed. The software engineer then identifies the affected components. Several potential solutions are provided and then a recommendation is made as to the best course of action.

Software designed with maintainability in mind greatly facilitates impact analysis. More information can be found in the Software Configuration Management KA.

2.1.4. Maintainability
[ISO14764-99:s6.8s6.8.1; Pfl01: c9s9.4; Pig97:c16]

How does one promote and follow up on maintainability issues during development? The IEEE [IEEE610.12-90] defines maintainability as the ease with which software can be maintained, enhanced, adapted, or corrected to satisfy specified requirements. ISO/IEC defines maintainability as one of the quality characteristics (ISO9126-01).

Maintainability sub-characteristics must be specified, reviewed, and controlled during the software development activities in order to reduce maintenance costs. If this is done successfully, the maintainability of the software will improve. This is often difficult to achieve because the maintainability sub-characteristics are not an important focus during the software development process. The developers are preoccupied with many other things and often disregard the maintainer's requirements. This in turn can, and often does, result in a lack of system documentation, which is a leading cause of difficulties in program comprehension and impact analysis. It has also been observed that the presence of systematic and mature

processes, techniques, and tools helps to enhance the maintainability of a system.

2.2. Management Issues

2.2.1. Alignment with organizational objectives
[Ben00:c6sa; Dor02:v1c9s1.6]

Organizational objectives describe how to demonstrate the return on investment of software maintenance activities. Bennett [Ben00] states that "initial software development is usually project-based, with a defined time scale and budget. The main emphasis is to deliver on time and within budget to meet user needs. In contrast, software maintenance often has the objective of extending the life of software for as long as possible. In addition, it may be driven by the need to meet user demand for software updates and enhancements. In both cases, the return on investment is much less clear, so that the view at senior management level is often of a major activity consuming significant resources with no clear quantifiable benefit for the organization."

2.2.2. Staffing
[Dek92:10-17; Dor02:v1c9s1.6; Par86: c4s8-c4s11] (Lie81)

Staffing refers to how to attract and keep software maintenance staff. Maintenance is often not viewed as glamorous work. Deklava provides a list of staffing-related problems based on survey data. [Dek92] As a result, software maintenance personnel are frequently viewed as "second-class citizens" (Lie81) and morale therefore suffers. [Dor02]

2.2.3. Process
[Pau93; Ben00:c6sb; Dor02:v1c9s1.3]

Software process is a set of activities, methods, practices, and transformations which people use to develop and maintain software and the associated products. [Pau93] At the process level, software maintenance activities share much in common with software development (for example, software configuration management is a crucial activity in both). [Ben00] Maintenance also requires several activities which are not found in software development (see section 3.2 on unique activities for details). These activities present challenges to management. [Dor02]

2.2.4. Organizational aspects of maintenance
[Pfl01:c12s12.1-c12s12.3; Par86:c4s7; Pig97:c2s2.5; Tak97:c8]

Organizational aspects describe how to identify which organization and/or function will be responsible for the maintenance of software. The team that develops the software is not necessarily assigned to maintain the software once it is operational.

In deciding where the software maintenance function will be located, software engineering organizations may, for example, stay with the original developer or go to a separate team (or maintainer). Often, the maintainer option is chosen to ensure that the software runs properly and evolves to satisfy changing user needs. Since there are many pros and cons to each of these options [Par86, Pig97], the decision should be made on a case-by-case basis. What is important is the delegation or assignment of the maintenance responsibility to a single group or person [Pig97], regardless of the organization's structure.

2.2.5. Outsourcing
[Dor02:v1c9s1.7; Pig97:c9s9.1,s9.2], (Car94; McC02)

Outsourcing of maintenance is becoming a major industry. Large corporations are outsourcing entire portfolios of software systems, including software maintenance. More often, the outsourcing option is selected for less mission-critical software, as companies are unwilling to lose control of the software used in their core business. Carey (Car94) reports that some will outsource only if they can find ways of maintaining strategic control. However, control measures are hard to find. One of the major challenges for the outsourcers is to determine the scope of the maintenance services required and the contractual details. McCracken (McC02) states that 50% of outsourcers provide services without any clear service-level agreement. Outsourcing companies typically spend a number of months assessing the software before they will enter into a contractual relationship. [Dor02] Another challenge identified is the transition of the software to the outsourcer. [Pig97]

2.3. Maintenance Cost Estimation

Software engineers must understand the different categories of software maintenance, discussed above, in order to address the question of estimating the cost of software maintenance. For planning purposes, estimating costs is an important aspect of software maintenance.

2.3.1. Cost estimation
[Art88:c3; Boe81:c30; Jon98:c27; Pfl01:c11s11.3; Pig97:c8]

It was mentioned in sub-topic 2.1.3, Impact Analysis, that impact analysis identifies all systems and software products affected by a software change request and develops an estimate of the resources needed to accomplish that change. [Art88]

Maintenance cost estimates are affected by many technical and non-technical factors. ISO/IEC14764 states that "the two most popular approaches to estimating resources for software maintenance are the use of parametric models and the use of experience" [ISO14764-99:s7.4.1]. Most often, a combination of these is used.

2.3.2. Parametric models
[Ben00:s7; Boe81:c30; Jon98:c27; Pfl01:c11s11.3]

Some work has been undertaken in applying parametric cost modeling to software maintenance. [Boe81, Ben00] Of

significance is that data from past projects are needed in order to use the models. Jones [Jon98] discusses all aspects of estimating costs, including function points (IEEE14143.1-00), and provides a detailed chapter on maintenance estimation.

2.3.3. Experience
[ISO14764-00:s7,s7.2,s7.2.1,s7.2.4; Pig97:c8; Sta94]

Experience, in the form of expert judgment (using the Delphi technique, for example), analogies, and a work breakdown structure, are several approaches which should be used to augment data from parametric models. Clearly the best approach to maintenance estimation is to combine empirical data and experience. These data should be provided as a result of a measurement program.

2.4. Software Maintenance Measurement
[IEEE1061-98:A.2; Pig97:c14s14.6; Gra87 ; Tak97: c6s6.1-c6s6.3]

Grady and Caswell [Gra87] discuss establishing a corporate-wide software measurement program, in which software maintenance measurement forms and data collection are described. The Practical Software and Systems Measurement (PSM) project describes an issue-driven measurement process that is used by many organizations and is quite practical. [McG01]

There are software measures that are common to all endeavors, the following categories of which the Software Engineering Institute (SEI) has identified: size; effort; schedule; and quality. [Pig97] These measures constitute a good starting point for the maintainer. Discussion of process and product measurement is presented in the Software Engineering Process KA. The software measurement program is described in the Software Engineering Management KA.

2.4.1. Specific Measures
[Car90:s2-s3; IEEE1219-98:Table3; Sta94:p239-249]

Abran [Abr93] presents internal benchmarking techniques to compare different internal maintenance organizations. The maintainer must determine which measures are appropriate for the organization in question. [IEEE1219-98; ISO9126-01; Sta94] suggests measures which are more specific to software maintenance measurement programs. That list includes a number of measures for each of the four sub-characteristics of maintainability:

- *Analyzability*: Measures of the maintainer's effort or resources expended in trying to diagnose deficiencies or causes of failure, or in identifying parts to be modified
- *Changeability*: Measures of the maintainer's effort associated with implementing a specified modification
- *Stability*: Measures of the unexpected behavior of software, including that encountered during testing
- *Testability*: Measures of the maintainer's and users' effort in trying to test the modified software

Certain measures of the maintainability of software can be obtained using available commercial tools. (Lag96; Apr00)

3. Maintenance Process

The *Maintenance Process* subarea provides references and standards used to implement the software maintenance process. The *Maintenance Activities* topic differentiates maintenance from development and shows its relationship to other software engineering activities.

The need for software engineering process is well documented. CMMI® models apply to software maintenance processes, and are similar to the developers' processes. [SEI01] Software Maintenance Capability Maturity models which address the unique processes of software maintenance are described in (Apr03, Nie02, Kaj01).

3.1. Maintenance Processes
[IEEE1219-98:s4; ISO14764-99:s8; IEEE12207.0-96:s5.5; Par86:c7s1; Pig97:c5; Tak97:c2]

Maintenance processes provide needed activities and detailed inputs/outputs to those activities, and are described in software maintenance standards IEEE 1219 and ISO/IEC 14764.

The maintenance process model described in the Standard for Software Maintenance (IEEE1219) starts with the software maintenance effort during the post-delivery stage and discusses items such as planning for maintenance. That process is depicted in Figure 2.

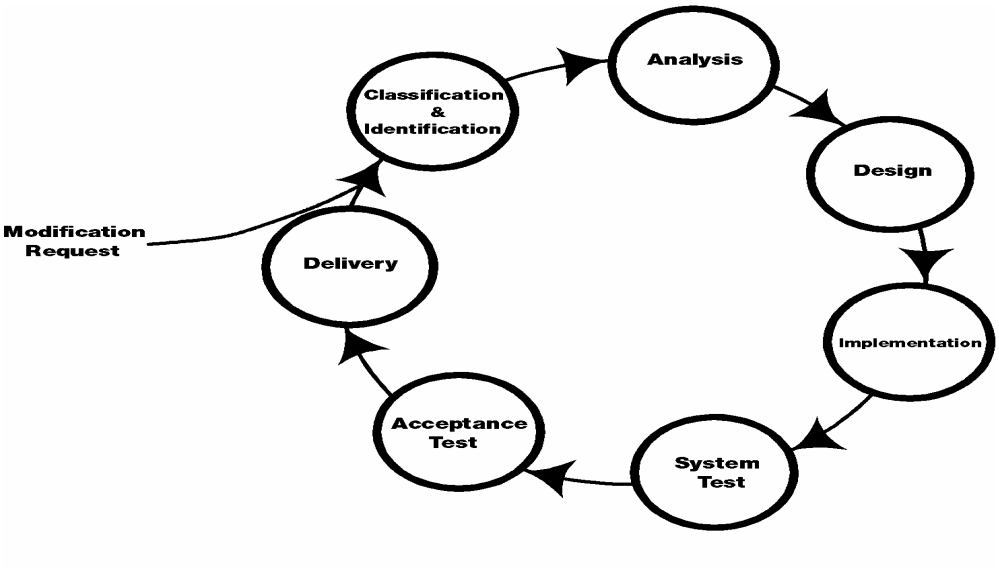


Figure 2 The IEEE1219-98 Maintenance Process Activities

ISO/IEC 14764 [ISO14764-99] is an elaboration of the IEEE/EIA 12207.0-96 maintenance process. The activities of the ISO/IEC maintenance process are similar to those of the IEEE, except that they are aggregated a little differently. The maintenance process activities developed by ISO/IEC are shown in Figure 3.

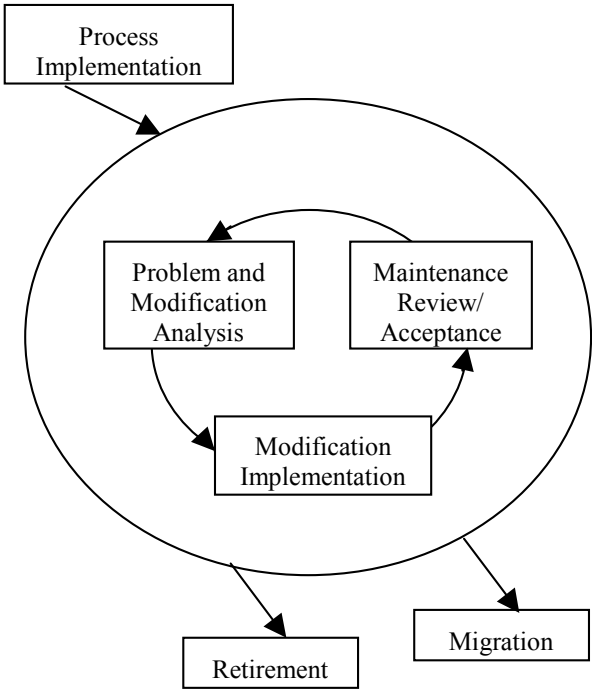


Figure 3 ISO/IEC 14764-00 Software Maintenance Process

Each of the ISO/IEC 14764 primary software maintenance activities is further broken down into tasks, as follows.

- Process Implementation
- Problem and Modification Analysis
- Modification Implementation
- Maintenance Review/Acceptance
- Migration
- Software Retirement

Takang & Grubb [Tak97] provide a history of maintenance process models leading up to the development of the IEEE and ISO/IEC process models. Parikh [Par86] also gives a good overview of a generic maintenance process. Recently, agile methodologies have been emerging which promote light processes. This requirement emerges from the ever-increasing demand for fast turn-around of maintenance services. Some experiments with Extreme maintenance are presented in (Poo01).

3.2. Maintenance Activities

As already noted, many maintenance activities are similar to those of software development. Maintainers perform analysis, design, coding, testing, and documentation. They must track requirements in their activities just as is done in development, and update documentation as baselines change. ISO/IEC14764 recommends that, when a maintainer refers to a similar development process, he must adapt it to meet his specific needs [ISO14764-99:s8.3.2.1, 2]. However, for software maintenance, some activities involve processes unique to software maintenance.

3.2.1. Unique activities
[Art88:c3; Dor02:v1c9s1.9.1; IEEE1219-98:s4.1,s4.2; ISO14764-99:s8.2.2.1,s8.3.2.1; Pfl01:c11s11.2]

There are a number of processes, activities, and practices that are unique to software maintenance, for example:

- Transition: a controlled and coordinated sequence of activities during which software is transferred progressively from the developer to the maintainer [Dek92, Pig97]
- Modification Request Acceptance/Rejection: modification request work over a certain size/effort/complexity may be rejected by maintainers and rerouted to a developer [Dor02], (Apr01)
- Modification Request and Problem Report Help Desk: an end-user support function that triggers the assessment, prioritization, and costing of modification requests [Ben00]
- Impact Analysis (see section 2.1.3 for details)
- Software Support: help and advice to users concerning a request for information (for example, business rules, validation, data meaning and ad-hoc requests/reports) (Apr03)
- Service Level Agreements (SLAs) and specialized (domain-specific) maintenance contracts which are the responsibility of the maintainers (Apr01)

3.2.2. Supporting activities
[IEEE1219-98:A.7,A.11; IEEE12207.0-96:c6,c7; ITI01; Pig97:c10s10.2,c18] ;(Kaj01)

Maintainers may also perform supporting activities, such as software maintenance planning, software configuration management, verification and validation, software quality assurance, reviews, audits, and user training.

Another supporting activity, maintainer training, is also needed. [Pig97; IEEE12207.0-96] (Kaj01)

3.2.3. Maintenance planning activity
[IEEE1219-98:A.3; ISO14764-99:s7; ITI01; Pig97:c7,c8]

An important activity for software maintenance is planning, and maintainers must address the issues associated with a number of planning perspectives:

- Business planning (organizational level)
- Maintenance planning (transition level)
- Release/version planning (software level)
- Individual software change request planning (request level)

At the individual request level, planning is carried out during the impact analysis (refer to sub-topic 2.1.3 Impact Analysis for details). The release/version planning activity requires that the maintainer [ITI01]:

- Collect the dates of availability of individual requests
- Agree with users on the content of subsequent releases/versions
- Identify potential conflicts and develop alternatives
- Assess the risk of a given release and develop a back-out plan in case problems should arise
- Inform all the stakeholders

Whereas software development projects can typically last from some months to a few of years, the maintenance phase usually lasts for many years. Making estimates of resources is a key element of maintenance planning. Those resources should be included in the developers' project planning budgets. Software maintenance planning should begin with the decision to develop a new system and should consider quality objectives (IEEE1061-98). A concept document should be developed, followed by a maintenance plan.

The concept document for maintenance [ISO14764-99:s7.2] should address:

- The scope of the software maintenance
- Adaptation of the software maintenance process
- Identification of the software maintenance organization
- An estimate of software maintenance costs

The next step is to develop a corresponding software maintenance plan. This plan should be prepared during software development, and should specify how users will request software modifications or report problems. Software maintenance planning [Pig97] is addressed in IEEE 1219 [IEEE1219-98] and ISO/IEC 14764. [ISO14764-99] ISO/IEC14764 provides guidelines for a maintenance plan.

Finally, at the highest level, the maintenance organization will have to conduct business planning activities (budgetary, financial, and human resources) just like all the other divisions of the organization. The management knowledge required to do so can be found in the Related Disciplines of Software Engineering chapter.

3.2.4. Software configuration management
[Art88:c2,c10; IEEE1219-98:A.11; IEEE12207.0-96:s6.2; Pfl01:c11s11.5; Tak97:c7]

The IEEE Standard for Software Maintenance, IEEE 1219 [IEEE1219-98], describes software configuration management as a critical element of the maintenance process. Software configuration management procedures should provide for the verification, validation, and audit of each step required to identify, authorize, implement, and release the software product.

It is not sufficient to simply track Modification Requests or Problem Reports. The software product and any changes made to it must be controlled. This control is established by implementing and enforcing an approved software configuration management (SCM) process. The Software Configuration Management KA provides details of SCM and discusses the process by which software change requests are submitted, evaluated, and approved. SCM for software maintenance is different from SCM for software

development in the number of small changes that must be controlled on operational software. The SCM process is implemented by developing and following a configuration management plan and operating procedures. Maintainers participate in Configuration Control Boards to determine the content of the next release/version.

3.2.5. Software quality
[Art98:c7s4; IEEE12207.0-96:s6.3; IEEE1219-98:A.7; ISO14764-99:s5.5.3.2]

It is not sufficient, either, to simply hope that increased quality will result from the maintenance of software. It must be planned and processes implemented to support the maintenance process. The activities and techniques for Software Quality Assurance (SQA), V&V, reviews, and audits must be selected in concert with all the other processes to achieve the desired level of quality. It is also recommended that the maintainer adapt the software development processes, techniques and deliverables, for instance testing documentation, and test results. [ISO14764-99]

More details can be found in the Software Quality KA.

4. Techniques for Maintenance

This subarea introduces some of the generally accepted techniques used in software maintenance.

4.1. Program Comprehension
[Arn92:c14; Dor02:v1c9s1.11.4; Tak97:c3]

Programmers spend considerable time in reading and understanding programs in order to implement changes. Code browsers are key tools for program comprehension. Clear and concise documentation can aid in program comprehension.

4.2. Reengineering
[Arn92:c1,c3-c6; Dor02:v1c9s1.11.4; IEEE1219-98: B.2], (Fow99)

Reengineering is defined as the examination and alteration of software to reconstitute it in a new form, and includes the subsequent implementation of the new form. Dorfman and Thayer [Dor02] state that reengineering is the most radical (and expensive) form of alteration. Others believe that reengineering can be used for minor changes. It is often not undertaken to improve maintainability, but to replace aging legacy software. Arnold [Arn92] provides a comprehensive compendium of topics, for example: concepts, tools and techniques, case studies, and risks and benefits associated with reengineering.

4.3. Reverse engineering
[Arn92:c12; Dor02:v1c9s1.11.3; IEEE1219-98:B.3; Tak97:c4, Hen01]

Reverse engineering is the process of analyzing software to identify the software's components and their inter-relationships and to create representations of the software in another form or at higher levels of abstraction. Reverse engineering is passive; it does not change the software, or result in new software. Reverse engineering efforts produce call graphs and control flow graphs from source code. One type of reverse engineering is redocumentation. Another type is design recovery [Dor02]. Refactoring is program transformation which reorganizes a program without changing its behavior, and is a form of reverse engineering that seeks to improve program structure. (Fow99)

Finally, data reverse engineering has gained in importance over the last few years where logical schemas are recovered from physical databases. (Hen01)

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	[Abr93]	[Arn92]	[Art88]	[Ben00]	[Boe81]	[Car90]	[Dek92]	[Dor97]	[Hen01]	[IEEE6610.12-90]	[IEEE1061-98]	[IEEE1219-98]	[IEEE12207.0-96]	[ISO14764-00]	[Jon98]	[Leh97]	[Par86]	[PfI01]	[Pig97]	[Pre04]	[Sta94]	[Tak97]
1. Software Maintenance Fundamentals																						
1.1 Definitions and Terminology	63-90			s4								s3.1.12		s6.1								
1.2 Nature of Maintenance													s3, s5.5					c11s11.2				
1.3 Need for Maintenance																		c11s11.2	c2s2.3			c1
1.4 Majority of Maintenance Costs	63-90																	c11s11.3	c3	C31		
1.5 Evolution of Software					c1s1.0-c1s1.2, c11s1.1, c11s1.2										108-124							
1.6 Categories of Maintenance	63-90				c1s1.2	s5				v1c9s1.5	s3.1.1, s3.1.2, s3.1.7, A.1.7		s4.1, s4.3s4.10, s4.11, s6.2					c2s2.3			
2. Key Issues in Software Maintenance																						
2.1 Technical Issues						s6c																
Limited understanding						s11.4				v1c9s1.11.4							c11s11.3				c3	
Testing					c9												c11s11.3					
Impact analysis					c3	s10.1, s10.2, s10.3				v1c9s1.10.1-v1c9s1.10.3							c11s11.5					
Maintainability		s3				s5, s9.3, s11.4							s6.8, s6.8.1				c9s9.4	c16				
2.2 Management Issues																						
Alignment with organizational objectives						s6a				v1c9s1.6												
Staffing									10-17	v1c9s1.6						c4s8-c4s11						
Process						s6b				v1c9s1.3												
Organizational aspects of maintenance						s6a, s7										c4s7	c12s12.1 - c12s12.3	c2s2.5			c8	
Outsourcing						s7				v1c9s1.7								c9s9.1-c9s9.2				
2.3 Maintenance Cost Estimation																						
Cost estimation					c3	s7	c30							c27			c11s11.3	c8				
Parametric models						s7	c30							c27			c11s11.3					
Experience													s7,s7.2, s7.2.1, s7.2.4					c8				

2.4 Measures	63-90		A.2			s9.1, s9.2, s10.1												c14s14.6			c6s6.1-c6s6.3
Specific Measures	63-90					s9.2		s2-s3			Table 3									239-249	
3. Maintenance Process																					
3.1 Maintenance Processes						s8					s4		s8			c7s1		c5			c2
3.2 Maintenance Activities						s9.1						s5.5									
Unique Activities	63-90				c3	s6b				v1c9s1.9.1	s4.1-s4.2		s8.2.2.1, s8.3.2.1				c11s11.2				
Supporting Activities											A.7,A.11	c6,c7						c10s10.2, c18			
Maintenance Planning Activity											A.3		s7					c7,c8			
Software Configuration Management					c2,c10	s2, s6b, s9.2, s10.1, s11.4					A.11	s6.2					c11s11.5				c7
Software Quality	63-90				c7s4	s7, s8, s9.2, s9.3, s11.4, s11.5					A.7	s6.3	s5.5.3.2								
4. Techniques for Maintenance																					
4.1 Program Comprehension				c14		s11.4				v1c9s1.11.4											c3
4.2 Re-engineering				c1,c3-c6		s9.2, s11.4, s11.5				v1c9s1.11.4	B.2										
4.3 Reverse Engineering				c12		s9.2, s10.3, s11.2, s11.3, s11.5			*	v1c9s1.11.3	B.3										c4

RECOMMENDED REFERENCES FOR SOFTWARE MAINTENANCE

[Abr93] A. Abran and H. Nguyenkim, "Measurement of the Maintenance Process from a Demand-Based Perspective," *Journal of Software Maintenance: Research and Practice*, vol. 5, iss. 2, 1993, pp. 63-90.

[Arn93] R.S. Arnold, *Software Reengineering*: IEEE Computer Society Press, 1993.

[Art98] L.J. Arthur, *Software Evolution: The Software Maintenance Challenge*, John Wiley & Sons, 1988.

[Ben00] K.H. Bennett, "Software Maintenance: A Tutorial," in *Software Engineering*, M. Dorfman and R. Thayer, eds., IEEE Computer Society Press, 2000.

[Boe81] B.W. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.

[Car90] D.N. Card and R.L. Glass, *Measuring Software Design Quality*, Prentice Hall, 1990.

[Dek92] S. Dekleva, "Delphi Study of Software Maintenance Problems," presented at the International Conference on Software Maintenance, 1992.

[Dor02] M. Dorfman and R.H. Thayer, eds., *Software Engineering (Vol. 1 & Vol. 2)*, IEEE Computer Society Press, 2002.

[Gra87] R.B. Grady and D.L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice Hall, 1987.

[Hen01] J. Henrard and J.-L. Hainaut, "Data Dependency Elicitation in Database Reverse Engineering," *Proc. of the 5th European Conference on Software Maintenance and Reengineering* (CSMR 2001), IEEE Computer Society Press, 2001.

[IEEE610.12-90] IEEE Std 610.12-1990 (R2002), *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1990.

[IEEE1061-98] IEEE Std 1061-1998, *IEEE Standard for a Software Quality Metrics Methodology*, IEEE, 1998.

[IEEE1219-98] IEEE Std 1219-1998, *IEEE Standard for Software Maintenance*, IEEE, 1998.

[IEEE12207.0-96] IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.

[ISO9126-01] ISO/IEC 9126-1:2001, *Software Engineering-Product Quality-Part 1: Quality Model*, ISO and IEC, 2001.

[ISO14764-99] ISO/IEC 14764-1999, *Software Engineering-Software Maintenance*, ISO and IEC, 1999.

[ITI01] IT Infrastructure Library, "Service Delivery and Service Support," Stationary Office, Office of Government of Commerce, 2001.

[Jon98] T.C. Jones, *Estimating Software Costs*, McGraw-Hill, 1998.

[Leh97] M.M. Lehman, "Laws of Software Evolution Revisited," presented at EWSPT96, 1997.

[Lie78] B. Lienz, E.B. Swanson, and G.E. Tompkins, "Characteristics of Applications Software Maintenance," *Communications of the ACM*, vol. 21, 1978.

[Par86] G. Parikh, *Handbook of Software Maintenance*, John Wiley & Sons, 1986.

[Pfl01] S.L. Pfleeger, *Software Engineering: Theory and Practice*, second ed., Prentice Hall, 2001.

[Pig97] T.M. Pigoski, *Practical Software Maintenance: Best Practices for Managing your Software Investment*, first ed., John Wiley & Sons, 1997.

[Pre04] R.S. Pressman, *Software Engineering: A Practitioner's Approach*, sixth ed., McGraw-Hill, 2004.

[SEI01] Software Engineering Institute, "Capability Maturity Model Integration, v1.1," CMU/SEI-2002-TR-002, ESC-TR-2002-002, December 2001.

[Sta94] G.E. Stark, L.C. Kern, and C.V. Vowell, "A Software Metric Set for Program Maintenance Management," *Journal of Systems and Software*, vol. 24, iss. 3, March 1994.

[Tak97] A. Takang and P. Grubb, *Software Maintenance Concepts and Practice*, International Thomson Computer Press, 1997.

APPENDIX A. LIST OF FURTHER READINGS

(Abr93) A. Abran, "Maintenance Productivity & Quality Studies: Industry Feedback on Benchmarking," presented at the Software Maintenance Conference (ICSM93), 1993.

(Apr00) A. April and D. Al-Shurougi, "Software Product Measurement for Supplier Evaluation," presented at FESMA2000, 2000.

(Apr01) A. April, J. Bouman, A. Abran, and D. Al-Shurougi, "Software Maintenance in a Service Level Agreement: Controlling the Customer's Expectations," presented at European Software Measurement Conference, 2001.

(Apr03) A. April, A. Abran, and R. Dumke, "Software Maintenance Capability Maturity Model (SM-CMM): Process Performance Measurement," presented at 13th International Workshop on Software Measurement (IWSM 2003), 2003.

(Bas85) V.R. Basili, "Quantitative Evaluation of Software Methodology," presented at First Pan-Pacific Computer Conference, 1985.

(Bel72) L. Belady and M.M. Lehman, "An Introduction to Growth Dynamics," *Statistical Computer Performance Evaluation*, W. Freiberger, ed., Academic Press, 1972.

(Ben00) K.H. Bennett and V.T. Rajlich, "Software Maintenance and Evolution: A Roadmap," *The Future of Software Engineering*, A. Finklestein, ed., ACM Press, 2000.

(Bol95) C. Boldyreff, E. Burd, R. Hather, R. Mortimer, M. Munro, and E. Younger, "The AMES Approach to Application Understanding: A Case Study," presented at the International Conference on Software Maintenance, 1995.

(Boo94) G. Booch and D. Bryan, *Software Engineering with Ada*, third ed., Benjamin/Cummings, 1994.

(Cap94) M.A. Capretz and M. Munro, "Software Configuration Management Issues in the Maintenance of Existing Systems," *Journal of Software Maintenance: Research and Practice*, vol. 6, iss. 2, 1994.

(Car92) J. Cardow, "You Can't Teach Software Maintenance!" presented at the Sixth Annual Meeting and Conference of the Software Management Association, 1992.

(Car94) D. Carey, "Executive Round-Table on Business Issues in Outsourcing - Making the Decision," *CIO Canada*, June/July 1994.

(Dor97) M. Dorfman and R.H. Thayer, eds., "Software Engineering," IEEE Computer Society Press, 1997.

(Dor02) M. Dorfman and R.H. Thayer, eds., *Software Engineering (Vol. 1 & Vol. 2)*, IEEE Computer Society Press, 2002.

(Fow99) M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

(Gra87) R.B. Grady and D.L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice Hall, 1987.

(Gra92) R.B. Grady, *Practical Software Metrics for Project Management and Process Management*, Prentice Hall, 1992.

(Jon91) C. Jones, *Applied Software Measurement*, McGraw-Hill, 1991.

(Kaj01) M. Kajko-Mattson, "Motivating the Corrective Maintenance Maturity Model (Cm3)," presented at Seventh International Conference on Engineering of Complex Systems, 2001.

(Kaj01a) M. Kajko-Mattson, S. Forssander, and U. Olsson, "Corrective Maintenance Maturity Model: Maintainer's Education and Training," presented at International Conference on Software Engineering, 2001.

(Kho95) T.M. Khoshgoftaar, R.M. Szabo, and J.M. Voas, "Detecting Program Module with Low Testability," presented at the International Conference on Software Maintenance-1995, 1995.

(Lag96) B. Laguë and A. April, "Mapping for the ISO9126 Maintainability Internal Metrics to an Industrial Research Tool," presented at SESS, 1996.

(Leh85) M.M. Lehman and L.A. Belady, *Program Evolution: Processes of Software Change,* Academic Press, 1985.

(Leh97) M.M. Lehman, "Laws of Software Evolution Revisited," presented at EWSPT96, 1997.

(Lie81) B.P. Lientz and E.B. Swanson, "Problems in Application Software Mainteanance," *Communications of the ACM*, vol. 24, iss. 11, 1981, pp. 763-769.

(McC02) B. McCracken, "Taking Control of IT Performance," presented at InfoServer LLC, 2002.

(Nie02) F. Niessink, V. Clerk, and H. v. Vliet, "The IT Capability Maturity Model," release L2+3-0.3 draft, 2002, available at http://www.itservicecmm.org/doc/itscmm-123-0.3.pdf.

(Oma91) P.W. Oman, J. Hagemeister, and D. Ash, "A Definition and Taxonomy for Software Maintainability," University of Idaho, Software Engineering Test Lab Technical Report 91-08 TR, November 1991.

(Oma92) P. Oman and J. Hagemeister, "Metrics for Assessing Software System Maintainability," presented at the International Conference on Software Maintenance '92, 1992.

(Pig93) T.M. Pigoski, "Maintainable Software: Why You Want It and How to Get It," presented at the Third

Software Engineering Research Forum - November 1993, University of West Florida Press, 1993.

(Pig94) T.M. Pigoski, "Software Maintenance," *Encyclopedia of Software Engineering*, John Wiley & Sons, 1994.

(Pol03) M. Polo, M. Piattini, and F. Ruiz, eds., "Advances in Software Maintenance Management: Technologies and Solutions," Idea Group Publishing, 2003.

(Poo00) C. Poole and W. Huisman, "Using Extreme Programming in a Maintenance Environment," *IEEE Software*, vol. 18, iss. 6, November/December 2001, pp. 42-50.

(Put97) L.H. Putman and W. Myers, "Industrial Strength Software - Effective Management Using Measurement," 1997.

(Sch99) S.R. Schach, *Classical and Object-Oriented Software Engineering with UML and C++*, McGraw-Hill, 1999.

(Sch97) S.L. Schneberger, *Client/Server Software Maintenance*, McGraw-Hill, 1997.

(Sch87) N.F. Schneidewind, "The State of Software Maintenance," *Proceedings of the IEEE*, 1987.

(Som96) I. Sommerville, *Software Engineering*, fifth ed., Addison-Wesley, 1996.

(Val94) J.D. Vallett, S.E. Condon, L. Briand, Y.M. Kim, and V.R. Basili, "Building on Experience Factory for Maintenance," presented at the Software Engineering Workshop, Software Engineering Laboratory, 1994.

APPENDIX A. LIST OF STANDARDS

(IEEE610.12-90) IEEE Std 610.12-1990 (R2002), *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1990.

(IEEE1061-98) IEEE Std 1061-1998, *IEEE Standard for a Software Quality Metrics Methodology*, IEEE, 1998.

(IEEE1219-98) IEEE Std 1219-1998, *IEEE Standard for Software Maintenance*, IEEE, 1998.

(IEEE12207.0-96) IEEE/EIA 12207.0-1996//ISO/ IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology - Software Life Cycle Processes*, IEEE, 1996.

(IEEE14143.1-00) IEEE Std 14143.1-2000// ISO/IEC14143-1:1998, *Information Technology - Software Measurement-Functional Size Measurement - Part 1: Definitions of Concepts*, IEEE, 2000.

(ISO9126-01) ISO/IEC 9126-1:2001, *Software Engineering-Product Quality - Part 1: Quality Model*, ISO and IEC, 2001.

(ISO14764-99) ISO/IEC 14764-1999, *Software Engineering - Software Maintenance*, ISO and IEC, 1999.

(ISO15271-98) ISO/IEC TR 15271:1998, *Information Technology - Guide for ISO/IEC 12207, (Software Life Cycle Process)*, ISO and IEC, 1998. [Abr93]

CHAPTER 7

SOFTWARE CONFIGURATION MANAGEMENT

ACRONYMS

CCB	Configuration Control Board
CM	Configuration Management
FCA	Functional Configuration Audit
MTBF	Mean Time Between Failures
PCA	Physical Configuration Audit
SCCB	Software Configuration Control Board
SCI	Software Configuration Item
SCM	Software Configuration Management
SCMP	Software Configuration Management Plan
SCR	Software Change Request
SCSA	Software Configuration Status Accounting
SEI/CMMI	Software Engineering Institute's Capability Maturity Model Integration
SQA	Software Quality Assurance
SRS	Software Requirement Specification
USNRC	U.S. Nuclear Regulatory Commission

INTRODUCTION

A *system* can be defined as a collection of components organized to accomplish a specific function or set of functions (IEEE 610.12-90). The *configuration* of a system is the functional and/or physical characteristics of hardware, firmware, or software, or a combination of these, as set forth in technical documentation and achieved in a product. (Buc96) It can also be thought of as a collection of specific versions of hardware, firmware, or software items combined according to specific build procedures to serve a particular purpose. *Configuration management* (CM), then, is the discipline of identifying the configuration of a system at distinct points in time for the purpose of systematically controlling changes to the configuration, and maintaining the integrity and traceability of the configuration throughout the system life cycle. (Ber97) It is formally defined (IEEE610.12-90) as

"A discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements."

Software configuration management (SCM) is a supporting software life cycle process (IEEE12207.0-96) which benefits project management, development and maintenance activities, assurance activities, and the customers and users of the end product.

The concepts of configuration management apply to all items to be controlled, although there are some differences in implementation between hardware CM and software CM.

SCM is closely related to the software quality assurance (SQA) activity. As defined in the Software Quality KA, SQA processes provide assurance that the software products and processes in the project life cycle conform to their specified requirements by planning, enacting, and performing a set of activities to provide adequate confidence that quality is being built into the software. SCM activities help in accomplishing these SQA goals. In some project contexts (see, for example, IEEE730-02), specific SQA requirements prescribe certain SCM activities.

The SCM activities are: management and planning of the SCM process, software configuration identification, software configuration control, software configuration status accounting, software configuration auditing, and software release management and delivery.

Figure 1 shows a stylized representation of these activities.

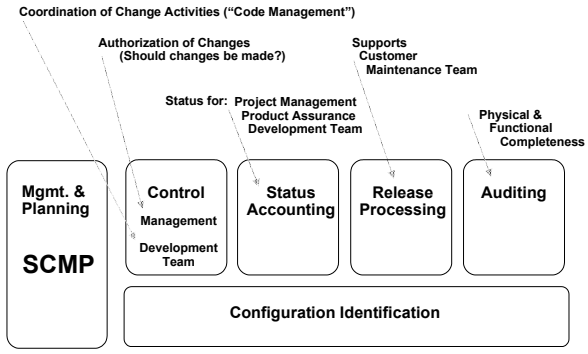


Figure 1. SCM Activities

The Software Configuration Management KA is related to all the other KAs, since the object of configuration management is the artifact produced and used throughout the software engineering process.

BREAKDOWN OF TOPICS FOR SCM

1. Management of the SCM Process

SCM controls the evolution and integrity of a product by identifying its elements, managing and controlling change, and verifying, recording, and reporting on configuration information. From the software engineer's perspective, SCM facilitates development and change implementation activities. A successful SCM implementation requires careful planning and management. This, in turn, requires an understanding of the organizational context for, and the constraints placed on, the design and implementation of the SCM process.

1.1. Organizational Context for SCM
[Ber92 :c4; Dar90:c2; IEEE828-98:c4s2.1]

To plan an SCM process for a project, it is necessary to understand the organizational context and the relationships among the organizational elements. SCM interacts with several other activities or organizational elements.

The organizational elements responsible for the software engineering supporting processes may be structured in various ways. Although the responsibility for performing certain SCM tasks might be assigned to other parts of the organization such as the development organization, the overall responsibility for SCM often rests with a distinct organizational element or designated individual.

Software is frequently developed as part of a larger system containing hardware and firmware elements. In this case, SCM activities take place in parallel with hardware and firmware CM activities, and must be consistent with system-level CM. Buckley [Buc96:c2] describes SCM within this context. Note that firmware contains hardware and software, therefore both hardware and software CM concepts are applicable.

SCM might interface with an organization's quality assurance activity on issues such as records management and non-conforming items. Regarding the former, some items under SCM control might also be project records subject to provisions of the organization's quality assurance program. Managing nonconforming items is usually the responsibility of the quality assurance activity; however, SCM might assist with tracking and reporting on software configuration items falling into this category.

Perhaps the closest relationship is with the software development and maintenance organizations.

It is within this context that many of the software configuration control tasks are conducted. Frequently, the same tools support development, maintenance, and SCM purposes.

1.2. Constraints and Guidance for the SCM Process
[Ber92:c5; IEEE828-98:c4s1,c4s2.3; Moo98]

Constraints affecting, and guidance for, the SCM process come from a number of sources. Policies and procedures set forth at corporate or other organizational levels might influence or prescribe the design and implementation of the SCM process for a given project. In addition, the contract between the acquirer and the supplier might contain provisions affecting the SCM process. For example, certain configuration audits might be required, or it might be specified that certain items be placed under CM. When software products to be developed have the potential to affect public safety, external regulatory bodies may impose constraints (see, for example, USNRC1.169-97). Finally, the particular software life cycle process chosen for a software project and the tools selected to implement the software affect the design and implementation of the SCM process. [Ber92]

Guidance for designing and implementing an SCM process can also be obtained from "best practice," as reflected in the standards on software engineering issued by the various standards organizations. Moore [Moo98] provides a roadmap to these organizations and their standards. Best practice is also reflected in process improvement and process assessment models such as the Software Engineering Institute's Capability Maturity Model Integration (SEI/CMMI) (SEI01) and ISO/IEC15504 Software Engineering–Process Assessment (ISO/IEC 15504-98).

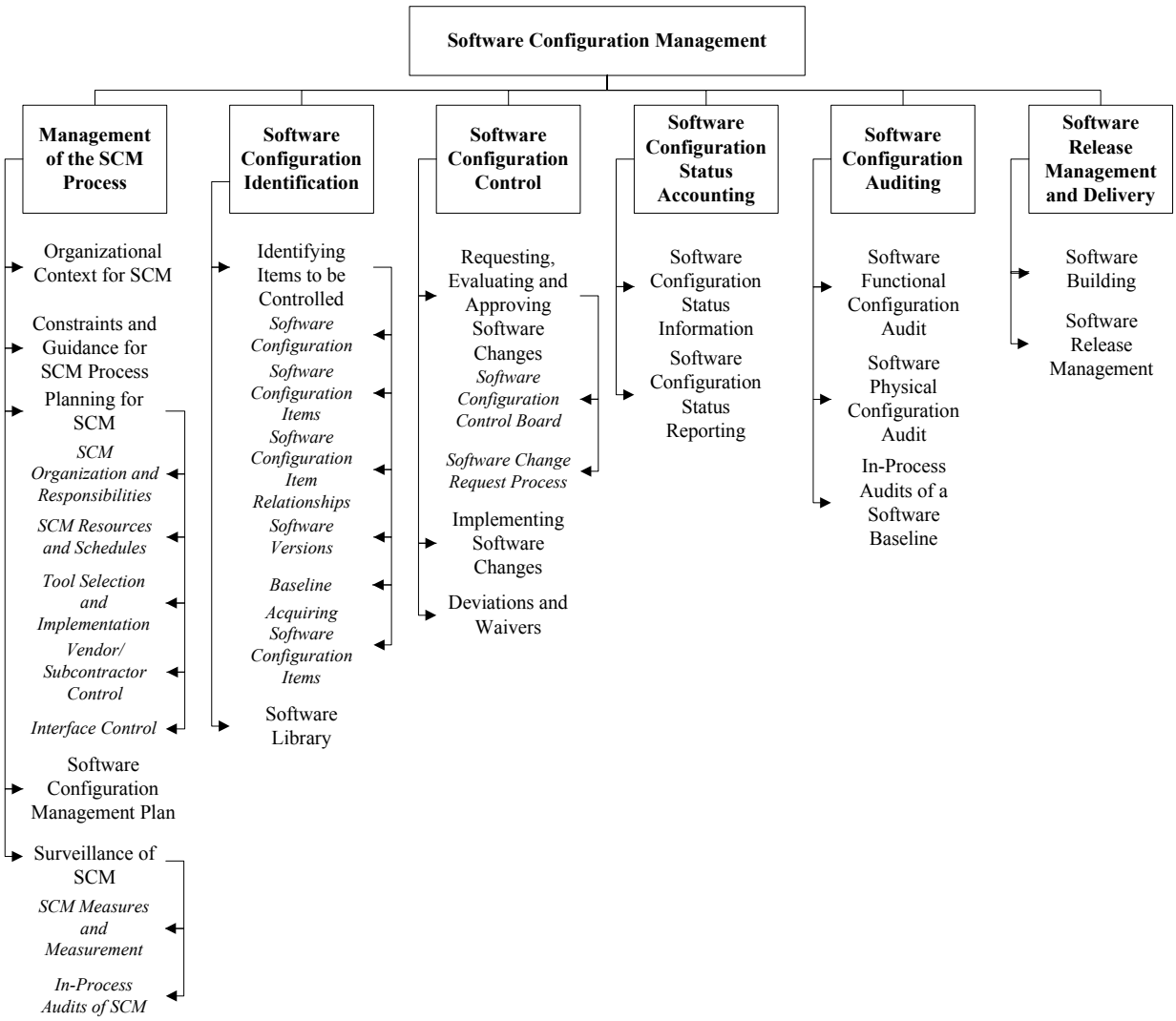


Figure 2 Breakdown of topics for the Software Configuration Management KA

1.3. Planning for SCM
[Dar90:c2; IEEE12207.0-96 :c6.s2.1; Som01:c29]

The planning of an SCM process for a given project should be consistent with the organizational context, applicable constraints, commonly accepted guidance, and the nature of the project (for example, size and criticality). The major activities covered are: Software Configuration Identification, Software Configuration Control, Software Configuration Status Accounting, Software Configuration Auditing, and Software Release Management and Delivery. In addition, issues such as organization and responsibilities, resources and schedules, tool selection and implementation, vendor and subcontractor control, and interface control are typically considered. The results of the planning activity are recorded in an SCM Plan (SCMP), which is typically subject to SQA review and audit.

1.3.1. SCM organization and responsibilities
[Ber92:c7; Buc96:c3; IEEE828-98:c4s2]

To prevent confusion about who will perform given SCM activities or tasks, organizations to be involved in the SCM process need to be clearly identified. Specific responsibilities for given SCM activities or tasks also need to be assigned to organizational entities, either by title or by organizational element. The overall authority and reporting channels for SCM should also be identified, although this might be accomplished at the project management or quality assurance planning stage.

1.3.2. SCM resources and schedules
[Ber92:c7; Buc96:c3; IEEE828-98:c4s4; c4s5]

Planning for SCM identifies the staff and tools involved in carrying out SCM activities and tasks. It addresses scheduling questions by establishing necessary sequences of SCM tasks and identifying their relationships to the project schedules and milestones established at the project management planning stage. Any training requirements necessary for implementing the plans and training new staff members are also specified.

1.3.3. Tool selection and implementation
[Ber92:c15; Con98:c6; Pre01:c31]

Different types of tool capabilities, and procedures for their use, support SCM activities. Depending on the situation, these tool capabilities can be made available with some combination of manual tools, automated tools providing a single SCM capability, automated tools integrating a range of SCM (and perhaps other) capabilities, or integrated tool environments which serve the needs of multiple participants in the software engineering process (for example, SCM, development, V&V). Automated tool support becomes increasingly important, and increasingly difficult to establish, as projects grow in size and as project environments become more complex. These tool capabilities provide support for:

- the SCM Library
- the software change request (SCR) and approval procedures
- code (and related work products) and change management tasks
- reporting software configuration status and collecting SCM measurements
- software configuration auditing
- managing and tracking software documentation
- performing software builds
- managing and tracking software releases and their delivery

The tools used in these areas can also provide measurements for process improvement. Royce [Roy98] describes seven core measures of value in managing software engineering processes. Information available from the various SCM tools relates to Royce's Work and Progress management indicator and to his quality indicators of Change Traffic and Stability, Breakage and Modularity, Rework and Adaptability, and MTBF (mean time between failures) and Maturity. Reporting on these indicators can be organized in various ways, such as by software configuration item or by type of change requested.

Figure 3 shows a representative mapping of tool capabilities and procedures to SCM Activities.

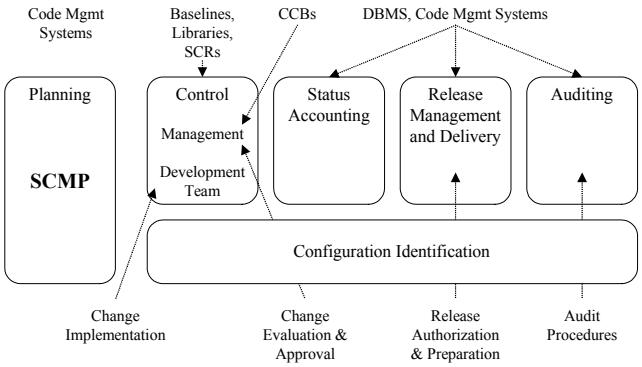


Figure 3 Characterization of SCM Tools and related procedures

In this example, code management systems support the operation of software libraries by controlling access to library elements, coordinating the activities of multiple users, and helping to enforce operating procedures. Other tools support the process of building software and release documentation from the software elements contained in the libraries. Tools for managing software change requests support the change control procedures applied to controlled software items. Other tools can provide database management and reporting capabilities for management, development, and quality assurance activities. As mentioned above, the capabilities of several tool types might be integrated into SCM systems, which in turn are closely coupled to various other software activities.

In planning, the software engineer picks SCM tools fit for the job. Planning considers issues that might arise in the implementation of these tools, particularly if some form of culture change is necessary. An overview of SCM systems and selection considerations is given in [Dar90:c3,AppA], and a case study on selecting an SCM system is given in [Mid97]. Complementary information on SCM tools can be found in the Software Engineering Tools and Methods KA.

1.3.4. Vendor/Subcontractor Control
[Ber92:c13; Buc96:c11; IEEE828-98:c4s3.6]

A software project might acquire or make use of purchased software products, such as compilers or other tools. SCM planning considers if and how these items will be taken under configuration control (for example, integrated into the project libraries) and how changes or updates will be evaluated and managed.

Similar considerations apply to subcontracted software. In this case, the SCM requirements to be imposed on the subcontractor's SCM process as part of the subcontract and the means for monitoring compliance also need to be established. The latter includes consideration of what SCM information must be available for effective compliance monitoring.

1.3.5. Interface control
[IEEE828-98:c4s3.5]

When a software item will interface with another software or hardware item, a change to either item can affect the other. The planning for the SCM process considers how the interfacing items will be identified and how changes to the items will be managed and communicated. The SCM role may be part of a larger, system-level process for interface specification and control, and may involve interface specifications, interface control plans, and interface control documents. In this case, SCM planning for interface control takes place within the context of the system-level process. A discussion of the performance of interface control activities is given in [Ber92:c12].

1.4. SCM Plan [Ber92:c7; Buc96:c3; Pau93:L2-81]

The results of SCM planning for a given project are recorded in a Software Configuration Management Plan (SCMP), a "living document" which serves as a reference for the SCM process. It is maintained (that is, updated and approved) as necessary during the software life cycle. In implementing the SCMP, it is typically necessary to develop a number of more detailed, subordinate procedures defining how specific requirements will be carried out during day-to-day activities.

Guidance on the creation and maintenance of an SCMP, based on the information produced by the planning activity, is available from a number of sources, such as [IEEE828-98:c4]. This reference provides requirements for the information to be contained in an SCMP. It also defines and describes six categories of SCM information to be included in an SCMP:

- Introduction (purpose, scope, terms used)
- SCM Management (organization, responsibilities, authorities, applicable policies, directives, and procedures)
- SCM Activities (configuration identification, configuration control, and so on)
- SCM Schedules (coordination with other project activities)
- SCM Resources (tools, physical resources, and human resources)
- SCMP Maintenance

1.5. Surveillance of Software Configuration Management
[Pau93:L2-87]

After the SCM process has been implemented, some degree of surveillance may be necessary to ensure that the provisions of the SCMP are properly carried out (see, for example [Buc96]). There are likely to be specific SQA requirements for ensuring compliance with specified SCM processes and procedures. This could involve an SCM authority ensuring that those with the assigned responsibility perform the defined SCM tasks correctly. The software quality assurance authority, as part of a compliance auditing activity, might also perform this surveillance.

The use of integrated SCM tools with process control capability can make the surveillance task easier. Some tools facilitate process compliance while providing flexibility for the software engineer to adapt procedures. Other tools enforce process, leaving the software engineer with less flexibility. Surveillance requirements and the level of flexibility to be provided to the software engineer are important considerations in tool selection.

1.5.1. SCM measures and measurement
[Buc96:c3; Roy98]

SCM measures can be designed to provide specific information on the evolving product or to provide insight into the functioning of the SCM process. A related goal of monitoring the SCM process is to discover opportunities for process improvement. Measurements of SCM processes provide a good means for monitoring the effectiveness of SCM activities on an ongoing basis. These measurements are useful in characterizing the current state of the process, as well as in providing a basis for making comparisons over time. Analysis of the measurements may produce insights leading to process changes and corresponding updates to the SCMP.

Software libraries and the various SCM tool capabilities provide sources for extracting information about the characteristics of the SCM process (as well as providing project and management information). For example, information about the time required to accomplish various types of changes would be useful in an evaluation of the criteria for determining what levels of authority are optimal for authorizing certain types of changes.

Care must be taken to keep the focus of the surveillance on the insights that can be gained from the measurements, not on the measurements themselves. Discussion of process and product measurement is presented in the Software Engineering Process KA. The software measurement program is described in the Software Engineering Management KA.

1.5.2. In-process audits of SCM
[Buc96:c15]

Audits can be carried out during the software engineering process to investigate the current status of specific elements of the configuration or to assess the implementation of the SCM process. In-process auditing of SCM provides a more formal mechanism for monitoring selected aspects of the process and may be coordinated with the SQA function. See also subarea 5 *Software Configuration Auditing*.

2. Software Configuration Identification
[IEEE12207.0-96:c6s2.2]

The software configuration identification activity identifies items to be controlled, establishes identification schemes for the items and their versions, and establishes the tools and techniques to be used in acquiring and managing controlled items. These activities provide the basis for the other SCM activities.

2.1. Identifying Items to Be Controlled
[Ber92:c8; IEEE828-98:c4s3.1; Pau93:L2-83; Som05:c29]

A first step in controlling change is to identify the software items to be controlled. This involves understanding the software configuration within the context of the system configuration, selecting software configuration items, developing a strategy for labeling software items and describing their relationships, and identifying the baselines to be used, along with the procedure for a baseline's acquisition of the items.

2.1.1. Software configuration
[Buc96:c4; c6, Pre04:c27]

A software configuration is the set of functional and physical characteristics of software as set forth in the technical documentation or achieved in a product (IEEE610.12-90). It can be viewed as a part of an overall system configuration.

2.1.2. Software configuration item
[Buc96:c4;c6; Con98:c2; Pre04:c27]

A software configuration item (SCI) is an aggregation of software designated for configuration management and is treated as a single entity in the SCM process (IEEE610.12-90). A variety of items, in addition to the code itself, is typically controlled by SCM. Software items with potential to become SCIs include plans, specifications and design documentation, testing materials, software tools, source and executable code, code libraries, data and data dictionaries, and documentation for installation, maintenance, operations, and software use.

Selecting SCIs is an important process in which a balance must be achieved between providing adequate visibility for project control purposes and providing a manageable number of controlled items. A list of criteria for SCI selection is given in [Ber92].

2.1.3. Software configuration item relationships
[Con98:c2; Pre04:c27]

The structural relationships among the selected SCIs, and their constituent parts, affect other SCM activities or tasks, such as software building or analyzing the impact of proposed changes. Proper tracking of these relationships is also important for supporting traceability. The design of the identification scheme for SCIs should consider the need to map the identified items to the software structure, as well as the need to support the evolution of the software items and their relationships.

2.1.4. Software version
[Bab86:c2]

Software items evolve as a software project proceeds. A *version* of a software item is a particular identified and specified item. It can be thought of as a state of an evolving item. [Con98:c3-c5] A *revision* is a new version of an item that is intended to replace the old version of the item. A *variant* is a new version of an item that will be added to the configuration without replacing the old version.

2.1.5. Baseline
[Bab86:c5; Buc96:c4; Pre04:c27]

A software baseline is a set of software configuration items formally designated and fixed at a specific time during the software life cycle. The term is also used to refer to a particular version of a software configuration item that has been agreed on. In either case, the baseline can only be changed through formal change control procedures. A baseline, together with all approved changes to the baseline, represents the current approved configuration.

Commonly used baselines are the functional, allocated, developmental, and product baselines (see, for example, [Ber92]). The functional baseline corresponds to the reviewed system requirements. The allocated baseline corresponds to the reviewed software requirements specification and software interface requirements specification. The developmental baseline represents the evolving software configuration at selected times during the software life cycle. Change authority for this baseline typically rests primarily with the development organization, but may be shared with other organizations (for example, SCM or Test). The product baseline corresponds to the completed software product delivered for system integration. The baselines to be used for a given project, along with their associated levels of authority needed for change approval, are typically identified in the SCMP.

2.1.6. Acquiring software configuration items
[Buc96:c4]

Software configuration items are placed under SCM control at different times; that is, they are incorporated into a particular baseline at a particular point in the software life cycle. The triggering event is the completion of some form of formal acceptance task, such as a formal review. Figure 2 characterizes the growth of baselined items as the life cycle proceeds. This figure is based on the waterfall model for purposes of illustration only; the subscripts used in the figure indicate versions of the evolving items. The software change request (SCR) is described in topic 3.1 *Requesting, Evaluating, and Approving Software Changes*.

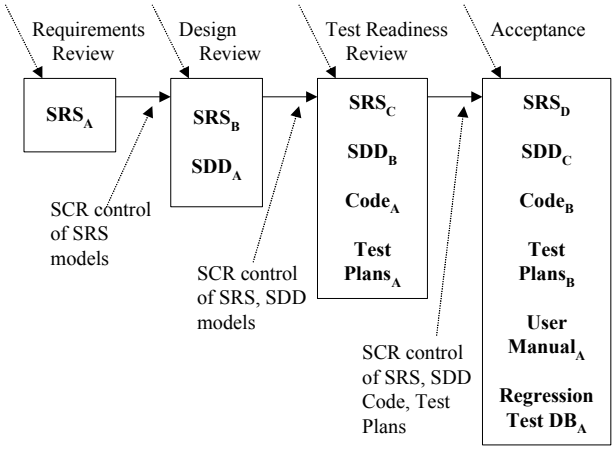


Figure 4 Acquisition of items

Following the acquisition of an SCI, changes to the item must be formally approved as appropriate for the SCI and the baseline involved, as defined in the SCMP. Following approval, the item is incorporated into the software baseline according to the appropriate procedure.

2.2. Software Library
[Bab86:c2; c5; Buc96:c4; IEEE828- 98:c4s3.1; Pau93:L2-82; Som01:c29]

A software library is a controlled collection of software and related documentation designed to aid in software development, use, and maintenance (IEEE610.12-90). It is also instrumental in software release management and delivery activities. Several types of libraries might be used, each corresponding to a particular level of maturity of the software item. For example, a working library could support coding and a project support library could support testing, while a master library could be used for finished products. An appropriate level of SCM control (associated baseline and level of authority for change) is associated with each library. Security, in terms of access control and the backup facilities, is a key aspect of library management. A model of a software library is described in [Ber92:c14].

The tool(s) used for each library must support the SCM control needs for that library, both in terms of controlling SCIs and controlling access to the library. At the working library level, this is a code management capability serving developers, maintainers, and SCM. It is focused on managing the versions of software items while supporting the activities of multiple developers. At higher levels of control, access is more restricted and SCM is the primary user.

These libraries are also an important source of information for measurements of work and progress.

3. Software Configuration Control

[IEEE12207.0-96:c6s2.3; Pau93:L2-84]

Software configuration control is concerned with managing changes during the software life cycle. It covers the process for determining what changes to make, the authority for approving certain changes, support for the implementation of those changes, and the concept of formal deviations from project requirements, as well as waivers of them. Information derived from these activities is useful in measuring change traffic and breakage, and aspects of rework.

3.1. Requesting, Evaluating, and Approving Software Changes
[IEEE828-98:c4s3.2; Pre04:c27; Som05:c29]

The first step in managing changes to controlled items is determining what changes to make. The software change request process (see Figure 5) provides formal procedures for submitting and recording change requests, evaluating the potential cost and impact of a proposed change, and accepting, modifying, or rejecting the proposed change. Requests for changes to software configuration items may be originated by anyone at any point in the software life cycle and may include a suggested solution and requested priority. One source of change requests is the initiation of corrective action in response to problem reports. Regardless of the source, the type of change (for example, defect or enhancement) is usually recorded on the SCR.

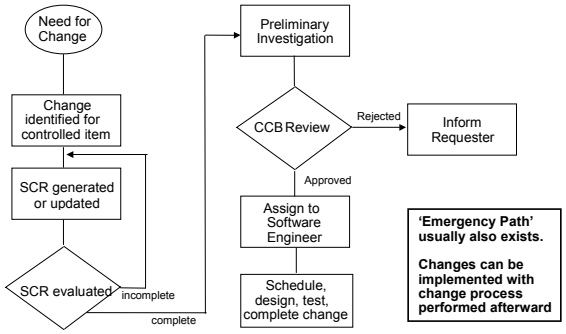


Figure 5 Flow of a Change Control Process

This provides an opportunity for tracking defects and collecting change activity measurements by change type. Once an SCR is received, a technical evaluation (also known as an impact analysis) is performed to determine the extent of the modifications that would be necessary should the change request be accepted. A good understanding of the relationships among software (and possibly, hardware) items is important for this task. Finally, an established authority, commensurate with the affected baseline, the SCI involved, and the nature of the change, will evaluate the technical and managerial aspects of the change request and either accept, modify, reject, or defer the proposed change.

3.1.1. Software Configuration Control Board
[Ber92:c9; Buc96:c9,c11; Pre04:c27]

The authority for accepting or rejecting proposed changes rests with an entity typically known as a Configuration Control Board (CCB). In smaller projects, this authority may actually reside with the leader or an assigned individual rather than a multi-person board. There can be multiple levels of change authority depending on a variety

of criteria, such as the criticality of the item involved, the nature of the change (for example, impact on budget and schedule), or the current point in the life cycle. The composition of the CCBs used for a given system varies depending on these criteria (an SCM representative would always be present). All stakeholders, appropriate to the level of the CCB, are represented. When the scope of authority of a CCB is strictly software, it is known as a Software Configuration Control Board (SCCB). The activities of the CCB are typically subject to software quality audit or review.

3.1.2. Software change request process
[Buc96:c9,c11; Pre04:c27]

An effective software change request (SCR) process requires the use of supporting tools and procedures ranging from paper forms and a documented procedure to an electronic tool for originating change requests, enforcing the flow of the change process, capturing CCB decisions, and reporting change process information. A link between this tool capability and the problem-reporting system can facilitate the tracking of solutions for reported problems. Change process descriptions and supporting forms (information) are given in a variety of references, for example [Ber92:c9].

3.2. Implementing Software Changes
[Bab86:c6; Ber92:c9; Buc96:c9,c11; IEEE828-98:c4s3.2.4; Pre04:c27; Som05:c29]

Approved SCRs are implemented using the defined software procedures in accordance with the applicable schedule requirements. Since a number of approved SCRs might be implemented simultaneously, it is necessary to provide a means for tracking which SCRs are incorporated into particular software versions and baselines. As part of the closure of the change process, completed changes may undergo configuration audits and software quality verification. This includes ensuring that only approved changes have been made. The change request process described above will typically document the SCM (and other) approval information for the change.

The actual implementation of a change is supported by the library tool capabilities, which provide version management and code repository support. At a minimum, these tools provide check-in/out and associated version control capabilities. More powerful tools can support parallel development and geographically distributed environments. These tools may be manifested as separate specialized applications under the control of an independent SCM group. They may also appear as an integrated part of the software engineering environment. Finally, they may be as elementary as a rudimentary change control system provided with an operating system.

3.3. Deviations and Waivers
[Ber92:c9; Buc96:c12]

The constraints imposed on a software engineering effort or the specifications produced during the development activities might contain provisions which cannot be satisfied at the designated point in the life cycle. A deviation is an authorization to depart from a provision prior to the development of the item. A waiver is an authorization to use an item, following its development, that departs from the provision in some way. In these cases, a formal process is used for gaining approval for deviations from, or waivers of, the provisions.

4. Software Configuration Status Accounting
[IEEE12207.0-96:c6s2.4; Pau93:L2-85; Pre04:c27; Som05:c29]

Software configuration status accounting (SCSA) is the recording and reporting of information needed for effective management of the software configuration.

4.1. Software Configuration Status Information
[Buc96:c13; IEEE828-98:c4s3.3]

The SCSA activity designs and operates a system for the capture and reporting of necessary information as the life cycle proceeds. As in any information system, the configuration status information to be managed for the evolving configurations must be identified, collected, and maintained. Various information and measurements are needed to support the SCM process and to meet the configuration status reporting needs of management, software engineering, and other related activities. The types of information available include the approved configuration identification, as well as the identification and current implementation status of changes, deviations, and waivers. A partial list of important data elements is given in [Ber92:c10].

Some form of automated tool support is necessary to accomplish the SCSA data collection and reporting tasks. This could be a database capability, or it could be a stand-alone tool or a capability of a larger, integrated tool environment.

4.2. Software Configuration Status Reporting
[Ber92:c10; Buc96:c13]

Reported information can be used by various organizational and project elements, including the development team, the maintenance team, project management, and software quality activities. Reporting can take the form of ad hoc queries to answer specific questions or the periodic production of predesigned reports. Some information produced by the status accounting activity during the course of the life cycle might become quality assurance records.

In addition to reporting the current status of the configuration, the information obtained by the SCSA can serve as a basis for various measurements of interest to management, development, and SCM. Examples include the number of change requests per SCI and the average time needed to implement a change request.

5. Software Configuration Auditing

[IEEE828-98:c4s3.4; IEEE12207.0-96:c6s2.5; Pau93:L2-86; Pre04:c26c27]

A software audit is an activity performed to independently evaluate the conformance of software products and processes to applicable regulations, standards, guidelines, plans, and procedures (IEEE1028-97). Audits are conducted according to a well-defined process consisting of various auditor roles and responsibilities. Consequently, each audit must be carefully planned. An audit can require a number of individuals to perform a variety of tasks over a fairly short period of time. Tools to support the planning and conduct of an audit can greatly facilitate the process. Guidance for conducting software audits is available in various references, such as [Ber92:c11; Buc96:c15] and (IEEE1028-97).

The software configuration auditing activity determines the extent to which an item satisfies the required functional and physical characteristics. Informal audits of this type can be conducted at key points in the life cycle. Two types of formal audits might be required by the governing contract (for example, in contracts covering critical software): the Functional Configuration Audit (FCA) and the Physical Configuration Audit (PCA). Successful completion of these audits can be a prerequisite for the establishment of the product baseline. Buckley [Buc96:c15] contrasts the purposes of the FCA and PCA in hardware versus software contexts, and recommends careful evaluation of the need for a software FCA and PCA before performing them.

5.1. Software Functional Configuration Audit

The purpose of the software FCA is to ensure that the audited software item is consistent with its governing specifications. The output of the software verification and validation activities is a key input to this audit.

5.2. Software Physical Configuration Audit

The purpose of the software physical configuration audit (PCA) is to ensure that the design and reference documentation is consistent with the as-built software product.

5.3. In-process Audits of a Software Baseline

As mentioned above, audits can be carried out during the development process to investigate the current status of specific elements of the configuration. In this case, an audit could be applied to sampled baseline items to ensure that performance is consistent with specifications or to ensure that evolving documentation continues to be consistent with the developing baseline item.

6. Software Release Management and Delivery

[IEEE12207.0-96:c6s2.6]

The term “release” is used in this context to refer to the distribution of a software configuration item outside the development activity. This includes internal releases as well as distribution to customers. When different versions of a software item are available for delivery, such as versions for different platforms or versions with varying capabilities, it is frequently necessary to recreate specific versions and package the correct materials for delivery of the version. The software library is a key element in accomplishing release and delivery tasks.

6.1. Software Building

[Bab86:c6; Som05:c29]

Software building is the activity of combining the correct versions of software configuration items, using the appropriate configuration data, into an executable program for delivery to a customer or other recipient, such as the testing activity. For systems with hardware or firmware, the executable program is delivered to the system-building activity. Build instructions ensure that the proper build steps are taken and in the correct sequence. In addition to building software for new releases, it is usually also necessary for SCM to have the capability to reproduce previous releases for recovery, testing, maintenance, or additional release purposes.

Software is built using particular versions of supporting tools, such as compilers. It might be necessary to rebuild an exact copy of a previously built software configuration item. In this case, the supporting tools and associated build instructions need to be under SCM control to ensure availability of the correct versions of the tools.

A tool capability is useful for selecting the correct versions of software items for a given target environment and for automating the process of building the software from the selected versions and appropriate configuration data. For large projects with parallel development or distributed development environments, this tool capability is necessary. Most software engineering environments provide this capability. These tools vary in complexity from requiring the software engineer to learn a specialized scripting language to graphics-oriented approaches that hide much of the complexity of an “intelligent” build facility.

The build process and products are often subject to software quality verification. Outputs of the build process might be needed for future reference and may become quality assurance records.

6.2. Software Release Management

[Som05:c29]

Software release management encompasses the identification, packaging, and delivery of the elements of a product, for example, executable program, documentation, release notes, and configuration data. Given that product changes can occur on a continuing basis, one concern for release management is determining when to issue a release. The severity of the problems addressed by the release and measurements of the fault densities of prior releases affect this decision. (Som01) The packaging task must identify which product items are to be delivered, and then select the

correct variants of those items, given the intended application of the product. The information documenting the physical contents of a release is known as a version description document. The release notes typically describe new capabilities, known problems, and platform requirements necessary for proper product operation. The package to be released also contains installation or upgrading instructions. The latter can be complicated by the fact that some current users might have versions that are several releases old. Finally, in some cases, the release management activity might be required to track the distribution of the product to various customers or target systems. An example would be a case where the supplier was required to notify a customer of newly reported problems.

A tool capability is needed for supporting these release management functions. It is useful to have a connection with the tool capability supporting the change request process in order to map release contents to the SCRs that have been received. This tool capability might also maintain information on various target platforms and on various customer environments.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

		[Bab86]	[Ber92]	[Buc96]	[Con98]	[Dar90]	[IEEE828-98]	[IEEE12207.0-96]	[Mid97]	[Moo98]	[Pau93]	[Pre04]	[Roy98]	[Som05]
1.	**Management of the SCM Process**													
1.1	*Organizational Context for SCM*		c4	c2		c2	c4s2.1							
1.2	*Constraints and Guidance for SCM Process*		c5				c4s1, c4s2.3			*				
1.3	*Planning for SCM*					c2		6.2.1						c29
	SCM organization and responsibilities		c7	c3			c4s2							
	SCM resources and schedules		c7	c3			c4s4, c4s5							
	Tool selection and implementation		c15		c6	c3, App A			*				*	
	Vendor/subcontractor control		c13	c11			c4s3.6							
	Interface control		c12				c4s3.5							
1.4	*SCM Plan*		c7	c3			c4				L2-81			
1.5	*Surveillance of SCM*			*			c4				L2-87			
	SCM measures and measurement			c3									188-202, 283-298	
	In-process audits of SCM			c15										
2.	**Software Configuration Identification**							c6s2.2						
2.1	*Identifying Items to be Controlled*		c8				c4s3.1				L2-83			c29
	Software configuration			c4,c6								c27		
	Software configuration item		*	c4,c6	c2							c27		
	Software configuration item relationships				c2							c27		
	Software versions	c2										c27		
	Baseline	c5	*	c4								c27		
	Acquiring software configuration items			c4										
2.2	*Software Library*	c2,c5	c14	c4			c4s3.1				L2-82			c29
3.	**Software Configuration Control**							c6s2.3			L2-84			
3.1	*Requesting, Evaluating and Approving Software Changes*						c4s3.2					c27		c29
	Software configuration control board		c9	c9,c11								c27		
	Software change request process		c9	c9,c11								c27		
3.2	*Implementing Software Changes*	c6	c9	c9,c11			c4s3.2.4					c27		c29
3.3	*Deviations and Waivers*		c9	c12										
4.	**Software Configuration Status Accounting**							c6s2.4			L2-85	c27		c29
4.1	*Software Configuration Status Information*		c10	c13			c4s3.3							
4.2	*Software Configuration Status Reporting*		c10	c13										
5.	**Software Configuration Auditing**		c11	c15			c4s3.4	c6s2.5			L2-86	c26, c27		
5.1	*Software Functional Configuration Audit*													
5.2	*Software Physical Configuration Audit*													
5.3	*In-Process Audits of a Software Baseline*													
6.	**Software Release Management and Delivery**							c6s2.6						
6.1	*Software Building*	c6												c29
6.2	*Software Release Management*													c29

RECOMMENDED REFERENCES FOR SCM

[Bab86] W.A. Babich, *Software Configuration Management, Coordination for Team Productivity*, Addison-Wesley, 1986.

[Ber92] H.R. Berlack, *Software Configuration Management*, John Wiley & Sons, 1992.

[Buc96] F.J. Buckley, *Implementing Configuration Management: Hardware, Software, and Firmware*, second ed., IEEE Computer Society Press, 1996.

[Con98] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," *ACM Computing Surveys*, vol. 30, iss. 2, June 1998.

[Dar90] S.A. Dart, *Spectrum of Functionality in Configuration Management Systems*, Software Engineering Institute, Carnegie Mellon University, 1990.

[IEEE828-98] IEEE Std 828-1998, *IEEE Standard for Software Configuration Management Plans*, IEEE, 1998.

[IEEE12207.0-96] IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.

[Mid97] A.K. Midha, "Software Configuration Management for the 21st Century," *Bell Labs Technical Journal*, vol. 2, iss. 1, Winter 1997, pp. 154-165.

[Moo98] J.W. Moore, *Software Engineering Standards: A User's Roadmap*, IEEE Computer Society, 1998.

[Pau93] M.C. Paulk et al., "Key Practices of the Capability Maturity Model, Version 1.1," technical report CMU/SEI-93-TR-025, Software Engineering Institute, Carnegie Mellon University, 1993.

[Pre04] R.S. Pressman, *Software Engineering: A Practitioner's Approach*, Sixth ed, McGraw-Hill, 2004.

[Roy98] W. Royce, *Software Project Management, A United Framework*, Addison-Wesley, 1998.

[Som05] I. Sommerville, *Software Engineering*, seventh ed., Addison-Wesley, 2005.

APPENDIX A. LIST OF FURTHER READINGS

(Bab86) W.A. Babich, *Software Configuration Management, Coordination for Team Productivity*, Addison-Wesley, 1986.

(Ber92) H.R. Berlack, *Software Configuration Management*, John Wiley & Sons, 1992.

(Ber97) E.H. Bersoff, "Elements of Software Configuration Management," in *Software Engineering*, M. Dorfman and R.H. Thayer, eds., IEEE Computer Society Press, 1997.

(Buc96) F.J. Buckley, *Implementing Configuration Management: Hardware, Software, and Firmware*, second ed., IEEE Computer Society Press, 1996.

(ElE98) K. El-Emam et al., "SPICE, The Theory and Practice of Software Process Improvement and Capability Determination," presented at IEEE Computer Society, 1998.

(Est95) J. Estublier, "Software Configuration Management," presented at ICSE SCM-4 and SCM-5 Workshops, Berlin, 1995.

(Gra92) R.B. Grady, *Practical Software Metrics for Project Management and Process Management*, Prentice Hall, 1992.

(Hoe02) A. v. d. Hoek, "Configuration Management Yellow Pages," 2002, available at http://www.cmtoday.com/yp/configuration_management.html.

(Hum89) W. Humphrey, *Managing the Software Process*, Addison-Wesley, 1989.

(Pau95) M.C. Paulk et al., *The Capability Maturity Model, Guidelines for Improving the Software Process*, Addison-Wesley, 1995.

(Som01a) I. Sommerville, "Software Configuration Management," presented at ICSE SCM-6 Workshop, Berlin, 2001.

(USNRC1.169-97) USNRC Regulatory Guide 1.169, "Configuration Management Plans for Digital Computer Software Used in Safety Systems of Nuclear Power Plants," presented at U.S. Nuclear Regulatory Commission, Washington, D.C., 1997.

(Vin88) J. Vincent, A. Waters, and J. Sinclair, *Software Quality Assurance: Practice and Implementation*, Prentice Hall, 1988.

(Whi91) D. Whitgift, *Methods and Tools for Software Configuration Management*, John Wiley & Sons, 1991.

APPENDIX B. LIST OF STANDARDS

(IEEE730-02) IEEE Std 730-2002, *IEEE Standard for Software Quality Assurance Plans*, IEEE, 2002.

(IEEE828-98) IEEE Std 828-1998, *IEEE Standard for Software Configuration Management Plans*, IEEE, 1998.

(IEEE1028-97) IEEE Std 1028-1997 (R2002), *IEEE Standard for Software Reviews*, IEEE, 1997.

(IEEE12207.0-96) IEEE/EIA 12207.0-1996//ISO/ IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.

(IEEE12207.1-96) IEEE/EIA 12207.1-1996, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes - Life Cycle Data*, IEEE, 1996.

(IEEE12207.2-97) IEEE/EIA 12207.2-1997, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes - Implementation Considerations*, IEEE, 1997.

(ISO15846-98) ISO/IEC TR 15846:1998, *Information Technology - Software Life Cycle Processes - Configuration Management*, ISO and IEC, 1998.

CHAPTER 8

SOFTWARE ENGINEERING MANAGEMENT

ACRONYM

PMBOK	Guide to the Project Management Body of Knowledge
SQA	Software Quality Assurance

INTRODUCTION

Software Engineering Management can be defined as the application of management activities—planning, coordinating, measuring, monitoring, controlling, and reporting—to ensure that the development and maintenance of software is systematic, disciplined, and quantified (IEEE610.12-90).

The Software Engineering Management KA therefore addresses the management and measurement of software engineering. While measurement is an important aspect of all KAs, it is here that the topic of measurement programs is presented.

While it is true to say that in one sense it should be possible to manage software engineering in the same way as any other (complex) process, there are aspects specific to software products and the software life cycle processes which complicate effective management—just a few of which are as follows:

- The perception of clients is such that there is often a lack of appreciation for the complexity inherent in software engineering, particularly in relation to the impact of changing requirements.
- It is almost inevitable that the software engineering processes themselves will generate the need for new or changed client requirements.
- As a result, software is often built in an iterative process rather than a sequence of closed tasks.
- Software engineering necessarily incorporates aspects of creativity and discipline—maintaining an appropriate balance between the two is often difficult.
- The degree of novelty and complexity of software is often extremely high.
- There is a rapid rate of change in the underlying technology.

With respect to software engineering, management activities occur at three levels: organizational and infrastructure management, project management, and measurement program planning and control. The last two are covered in detail in this KA description. However, this is not to diminish the importance of organizational management issues.

Since the link to the related disciplines—obviously management—is important, it will be described in more detail than in the other KA descriptions. Aspects of organizational management are important in terms of their impact on software engineering—on policy management, for instance: organizational policies and standards provide the framework in which software engineering is undertaken. These policies may need to be influenced by the requirements of effective software development and maintenance, and a number of software engineering-specific policies may need to be established for effective management of software engineering at an organizational level. For example, policies are usually necessary to establish specific organization-wide processes or procedures for such software engineering tasks as designing, implementing, estimating, tracking, and reporting. Such policies are essential to effective long-term software engineering management, by establishing a consistent basis on which to analyze past performance and implement improvements, for example.

Another important aspect of management is personnel management: policies and procedures for hiring, training, and motivating personnel and mentoring for career development are important not only at the project level but also to the longer-term success of an organization. Software engineering personnel may present unique training or personnel management challenges (for example, maintaining currency in a context where the underlying technology undergoes continuous and rapid change). Communication management is also often mentioned as an overlooked but major aspect of the performance of individuals in a field where precise understanding of user needs and of complex requirements and designs is necessary. Finally, portfolio management, which is the capacity to have an overall vision not only of the set of software under development but also of the software already in use in an organization, is necessary. Furthermore, software reuse is a key factor in maintaining and improving productivity and competitiveness. Effective reuse requires a strategic vision that reflects the unique power and requirements of this technique.

In addition to understanding the aspects of management that are uniquely influenced by software, software engineers must have some knowledge of the more general aspects, even in the first four years after graduation that is targeted in the Guide.

Organizational culture and behavior, and functional enterprise management in terms of procurement, supply chain management, marketing, sales, and distribution, all have an influence, albeit indirectly, on an organization's software engineering process.

Relevant to this KA is the notion of project management, as "the construction of useful software artifacts" is normally managed in the form of (perhaps programs of) individual projects. In this regard, we find extensive support in the Guide to the Project Management Body of Knowledge (PMBOK) (PMI00), which itself includes the following project management KAs: project integration management, project scope management, project time management, project cost management, project quality management, project human resource management, and project communications management. Clearly, all these topics have direct relevance to the Software Engineering Management KA. To attempt to duplicate the content of the Guide to the PMBOK here would be both impossible and inappropriate. Instead, we suggest that the reader interested in project management beyond what is specific to software engineering projects consult the PMBOK itself. Project management is also found in the Related Disciplines of Software Engineering chapter.

The Software Engineering Management KA consists of both the software project management process, in its first five subareas, and software engineering measurement in the last subarea. While these two subjects are often regarded as being separate, and indeed they do possess many unique aspects, their close relationship has led to their combined treatment in this KA. Unfortunately, a common perception of the software industry is that it delivers products late, over budget, and of poor quality and uncertain functionality. Measurement-informed management — an assumed principle of any true engineering discipline — can help to turn this perception around. In essence, management without measurement, qualitative and quantitative, suggests a lack of rigor, and measurement without management suggests a lack of purpose or context. In the same way, however, management and measurement without expert knowledge is equally ineffectual, so we must be careful to avoid over-emphasizing the quantitative aspects of Software Engineering Management (SEM). Effective management requires a combination of both numbers and experience.

The following working definitions are adopted here:

- *Management process* refers to the activities that are undertaken in order to ensure that the software engineering processes are performed in a manner consistent with the organization's policies, goals, and standards.
- *Measurement* refers to the assignment of values and labels to aspects of software engineering (products, processes, and resources as defined by [Fen98]) and the models that are derived from them, whether these models are developed using statistical, expert knowledge or other techniques.

The software engineering project management subareas make extensive use of the software engineering measurement subarea.

Not unexpectedly, this KA is closely related to others in the Guide to the SWEBOK, and reading the following KA descriptions in conjunction with this one would be particularly useful.

- *Software Requirements*, where some of the activities to be performed during the Initiation and Scope definition phase of the project are described
- *Software Configuration Management***,** as this deals with the identification, control, status accounting, and audit of the software configuration along with software release management and delivery
- *Software Engineering Process***,** because processes and projects are closely related (this KA also describes process and product measurement)
- *Software Quality***,** as quality is constantly a goal of management and is an aim of many activities that must be managed

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING MANAGEMENT

As the Software Engineering Management KA is viewed here as an organizational process which incorporates the notion of process and project management, we have created a breakdown that is both topic-based and life cycle-based. However, the primary basis for the top-level breakdown is the process of managing a software engineering project. There are six major subareas. The first five subareas largely follow the IEEE/EIA 12207 Management Process. The six subareas are:

- *Initiation and scope definition*, which deals with the decision to initiate a software engineering project
- *Software project planning*, which addresses the activities undertaken to prepare for successful software engineering from a management perspective
- *Software project enactment*, which deals with generally accepted software engineering management activities that occur during software engineering
- *Review and evaluation*, which deal with assurance that the software is satisfactory
- *Closure*, which addresses the post-completion activities of a software engineering project
- *Software engineering measurement*, which deals with the effective development and implementation of measurement programs in software engineering organizations (IEEE12207.0-96)

The breakdown of topics for the Software Engineering Management KA is shown in Figure 1.

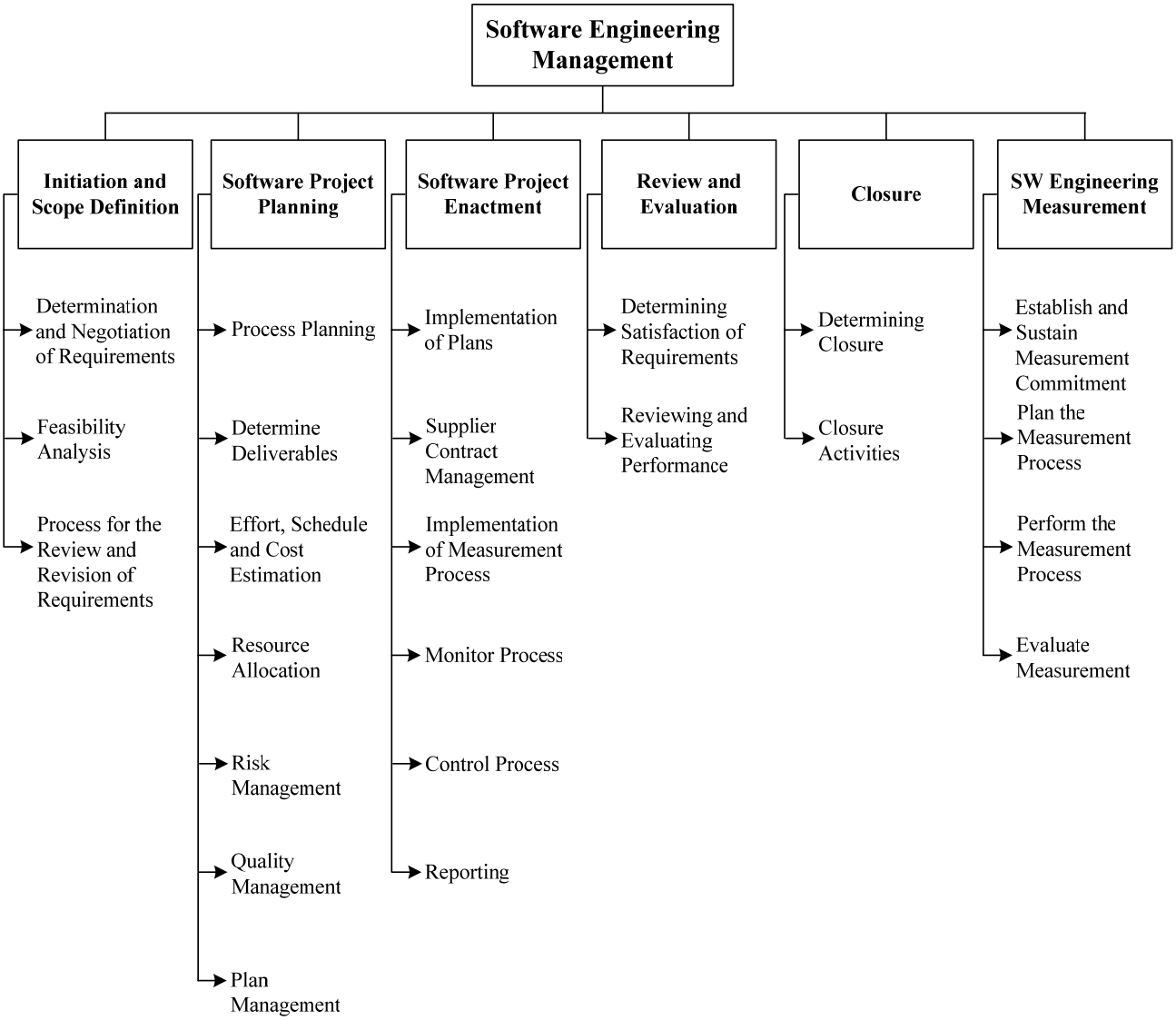


Figure 1 Breakdown of topics for the Software Engineering Management KA

1. Initiation and Scope Definition

The focus of this set of activities is on the effective determination of software requirements via various elicitation methods and the assessment of the project's feasibility from a variety of standpoints. Once feasibility has been established, the remaining task within this process is the specification of requirements validation and change procedures (see also the Software Requirements KA).

1.1. Determination and Negotiation of Requirements

[Dor02: v2c4; Pfl01: c4; Pre04: c7; Som05: c5]

Software requirement methods for requirements elicitation (for example, observation), analysis (for example, data modeling, use-case modeling), specification, and validation (for example, prototyping) must be selected and applied, taking into account the various stakeholder perspectives. This leads to the determination of project scope, objectives, and constraints. This is always an important activity, as it sets the visible boundaries for the set of tasks being undertaken, and is particularly so where the novelty of the undertaking is high. Additional information can be found in the Software Requirements KA.

1.2. Feasibility Analysis (Technical, Operational, Financial, Social/Political)

[Pre04: c6; Som05: c6]

Software engineers must be assured that adequate capability and resources are available in the form of people, expertise, facilities, infrastructure, and support (either internally or externally) to ensure that the project can be successfully completed in a timely and cost-effective manner (using, for example, a requirement-capability matrix). This often requires some "ballpark" estimation of effort and cost based on appropriate methods (for example, expert-informed analogy techniques).

1.3. Process for the Review and Revision of Requirements

Given the inevitability of change, it is vital that agreement among stakeholders is reached at this early point as to the means by which scope and requirements are to be reviewed and revised (for example, via agreed change management procedures). This clearly implies that scope and requirements will not be "set in stone" but can and should be revisited at predetermined points as the process unfolds (for example, at design reviews, management reviews). If changes are accepted, then some form of traceability analysis and risk analysis (see topic 2.5 *Risk Management*) should be used to ascertain the impact of those changes. A managed-change approach should also be useful when it comes time to review the outcome of the project, as the scope and requirements should form the basis for the evaluation of success. [Som05: c6] See also the software configuration control subarea of the Software Configuration Management KA.

2. Software Project Planning

The iterative planning process is informed by the scope and requirements and by the establishment of feasibility. At this point, software life cycle processes are evaluated and the most appropriate (given the nature of the project, its degree of novelty, its functional and technical complexity, its quality requirements, and so on) is selected. Where relevant, the project itself is then planned in the form of a hierarchical decomposition of tasks, the associated deliverables of each task are specified and characterized in terms of quality and other attributes in line with stated requirements, and detailed effort, schedule, and cost estimation is undertaken. Resources are then allocated to tasks so as to optimize personnel productivity (at individual, team, and organizational levels), equipment and materials utilization, and adherence to schedule. Detailed risk management is undertaken and the "risk profile" of the project is discussed among, and accepted by, all relevant stakeholders. Comprehensive software quality management processes are determined as part of the planning process in the form of procedures and responsibilities for software quality assurance, verification and validation, reviews, and audits (see the Software Quality KA). As an iterative process, it is vital that the processes and responsibilities for ongoing plan management, review, and revision are also clearly stated and agreed.

2.1. Process Planning

Selection of the appropriate software life cycle model (for example, spiral, evolutionary prototyping) and the adaptation and deployment of appropriate software life cycle processes are undertaken in light of the particular scope and requirements of the project. Relevant methods and tools are also selected. [Dor02: v1c6,v2c8; Pfl01: c2; Pre04: c2; Rei02: c1,c3,c5; Som05: c3; Tha97: c3] At the project level, appropriate methods and tools are used to decompose the project into tasks, with associated inputs, outputs, and completion conditions (for example, work breakdown structure). [Dor02: v2c7; Pfl01: c3; Pre04: c21; Rei02: c4,c5; Som05: c4; Tha97: c4,c6] This in turn influences decisions on the project's high-level schedule and organization structure.

2.2. Determine Deliverables

The product(s) of each task (for example, architectural design, inspection report) are specified and characterized. [Pfl01: c3; Pre04: c24; Tha97: c4] Opportunities to reuse software components from previous developments or to utilize off-the-shelf software products are evaluated. Use of third parties and procured software are planned and suppliers are selected.

2.3. Effort, Schedule, and Cost Estimation

Based on the breakdown of tasks, inputs, and outputs, the expected effort range required for each task is determined using a calibrated estimation model based on historical size-effort data where available and relevant, or other methods like expert judgment. Task dependencies are established and potential bottlenecks are identified using suitable methods (for example, critical path analysis). Bottlenecks are resolved where possible, and the expected schedule of tasks with projected start times, durations, and end times is produced (for example, PERT chart). Resource requirements (people, tools) are translated into cost estimates. [Dor02: v2c7; Fen98: c12; Pfl01: c3; Pre04: c23, c24; Rei02: c5,c6; Som05: c4,c23; Tha97: c5] This is a highly iterative activity which must be negotiated and revised until consensus is reached among affected stakeholders (primarily engineering and management).

2.4. Resource Allocation

[Pfl01: c3; Pre04: c24; Rei02: c8,c9; Som05: c4; Tha97: c6,c7]

Equipment, facilities, and people are associated with the scheduled tasks, including the allocation of responsibilities for completion (using, for example, a Gantt chart). This activity is informed and constrained by the availability of resources and their optimal use under these circumstances, as well as by issues relating to personnel (for example, productivity of individuals/teams, team dynamics, organizational and team structures).

2.5. Risk Management

Risk identification and analysis (what can go wrong, how and why, and what are the likely consequences), critical risk assessment (which are the most significant risks in terms of exposure, which can we do something about in terms of leverage), risk mitigation and contingency planning (formulating a strategy to deal with risks and to manage the risk profile) are all undertaken. Risk assessment methods (for example, decision trees and process simulations) should be used in order to highlight and evaluate risks. Project abandonment policies should also be determined at this point in discussion with all other stakeholders. [Dor02: v2c7; Pfl01: c3; Pre04: c25; Rei02: c11; Som05: c4; Tha97: c4] Software-unique aspects of risk, such as software engineers' tendency to add unwanted features or the risks attendant in software's intangible nature, must influence the project's risk management.

2.6. Quality Management

[Dor02: v1c8,v2c3-c5; Pre04: c26; Rei02: c10;

Som05: c24,c25; Tha97: c9,c10]

Quality is defined in terms of pertinent attributes of the specific project and any associated product(s), perhaps in both quantitative and qualitative terms. These quality characteristics will have been determined in the specification of detailed software requirements. See also the Software Requirements KA.

Thresholds for adherence to quality are set for each indicator as appropriate to stakeholder expectations for the software at hand. Procedures relating to ongoing SQA throughout the process and for product (deliverable) verification and validation are also specified at this stage (for example, technical reviews and inspections) (see also the Software Quality KA).

2.7. Plan Management

[Som05: c4; Tha97: c4]

How the project will be managed and how the plan will be managed must also be planned. Reporting, monitoring, and control of the project must fit the selected software engineering process and the realities of the project, and must be reflected in the various artifacts that will be used for managing it. But, in an environment where change is an expectation rather than a shock, it is vital that plans are themselves managed. This requires that adherence to plans be systematically directed, monitored, reviewed, reported, and, where appropriate, revised. Plans associated with other management-oriented support processes (for example, documentation, software configuration management, and problem resolution) also need to be managed in the same manner.

3. Software Project Enactment

The plans are then implemented, and the processes embodied in the plans are enacted. Throughout, there is a focus on adherence to the plans, with an overriding expectation that such adherence will lead to the successful satisfaction of stakeholder requirements and achievement of the project objectives. Fundamental to enactment are the ongoing management activities of measuring, monitoring, controlling, and reporting.

3.1. Implementation of Plans

[Pfl01: c3; Som05: c4]

The project is initiated and the project activities are undertaken according to the schedule. In the process, resources are utilized (for example, personnel effort, funding) and deliverables are produced (for example, architectural design documents, test cases).

3.2. Supplier Contract Management

[Som05:c4]

Prepare and execute agreements with suppliers, monitor supplier performance, and accept supplier products, incorporating them as appropriate.

3.3. Implementation of measurement process

[Fen98: c13,c14; Pre04: c22; Rei02: c10,c12;
Tha97: c3,c10]

The measurement process is enacted alongside the software project, ensuring that relevant and useful data are collected (see also topics 6.2 *Plan the Measurement Process* and 6.3 *Perform the Measurement Process*).

3.4. Monitor Process

[Dor02: v1c8, v2c2-c5,c7; Rei02: c10;

Som05: c25; Tha97: c3;c9]

Adherence to the various plans is assessed continually and at predetermined intervals. Outputs and completion conditions for each task are analyzed. Deliverables are evaluated in terms of their required characteristics (for example, via reviews and audits). Effort expenditure, schedule adherence, and costs to date are investigated, and resource usage is examined. The project risk profile is revisited, and adherence to quality requirements is evaluated.

Measurement data are modeled and analyzed. Variance analysis based on the deviation of actual from expected outcomes and values is undertaken. This may be in the form of cost overruns, schedule slippage, and the like. Outlier identification and analysis of quality and other measurement data are performed (for example, defect density analysis). Risk exposure and leverage are recalculated, and decisions trees, simulations, and so on are rerun in the light of new data. These activities enable problem detection and exception identification based on exceeded thresholds. Outcomes are reported as needed and certainly where acceptable thresholds are surpassed.

3.5. Control Process

[Dor02: v2c7; Rei02: c10; Tha97: c3,c9]

The outcomes of the process monitoring activities provide the basis on which action decisions are taken. Where

appropriate, and where the impact and associated risks are modeled and managed, changes can be made to the project. This may take the form of corrective action (for example, retesting certain components), it may involve the incorporation of contingencies so that similar occurrences are avoided (for example, the decision to use prototyping to assist in software requirements validation), and/or it may entail the revision of the various plans and other project documents (for example, requirements specification) to accommodate the unexpected outcomes and their implications.

In some instances, it may lead to abandonment of the project. In all cases, change control and software configuration management procedures are adhered to (see also the Software Configuration Management KA), decisions are documented and communicated to all relevant parties, plans are revisited and revised where necessary, and relevant data is recorded in the central database (see also topic 6.3 *Perform the Measurement Process*).

3.6. Reporting

[Rei02: c10; Tha97: c3,c10]

At specified and agreed periods, adherence to the plans is reported, both within the organization (for example to the project portfolio steering committee) and to external stakeholders (for example, clients, users). Reports of this nature should focus on overall adherence as opposed to the detailed reporting required frequently within the project team.

4. Review and Evaluation

At critical points in the project, overall progress towards achievement of the stated objectives and satisfaction of stakeholder requirements are evaluated. Similarly, assessments of the effectiveness of the overall process to date, the personnel involved, and the tools and methods employed are also undertaken at particular milestones.

4.1. Determining Satisfaction of Requirements

[Rei02: c10; Tha97: c3,c10]

Since attaining stakeholder (user and customer) satisfaction is one of our principal aims, it is important that progress towards this aim be formally and periodically assessed. This occurs on achievement of major project milestones (for example, confirmation of software design architecture, software integration technical review). Variances from expectations are identified and appropriate action is taken. As in the control process activity above (see topic 3.5 *Control Process*), in all cases change control and software configuration management procedures are adhered to (see the Software Configuration Management KA), decisions are documented and communicated to all relevant parties, plans are revisited and revised where necessary, and relevant data are recorded in the central database (see also topic 6.3 *Perform the Measurement Process*). More information can also be found in the Software Testing KA, in topic 2.2 *Objectives of Testing* and in the Software Quality KA, in topic 2.3 *Reviews and Audits*.

4.2. Reviewing and Evaluating Performance

[Dor02: v1c8,v2c3,c5; Pfl01: c8,c9; Rei02: c10; Tha97: c3,c10]

Periodic performance reviews for project personnel provide insights as to the likelihood of adherence to plans as well as possible areas of difficulty (for example, team member conflicts). The various methods, tools, and techniques employed are evaluated for their effectiveness and appropriateness, and the process itself is systematically and periodically assessed for its relevance, utility, and efficacy in the project context. Where appropriate, changes are made and managed.

5. Closure

The project reaches closure when all the plans and embodied processes have been enacted and completed. At this stage, the criteria for project success are revisited. Once closure is established, archival, post mortem, and process improvement activities are performed.

5.1. Determining Closure

[Dor02: v1c8,v2c3,c5; Rei02: c10; Tha97: c3,c10]

The tasks as specified in the plans are complete, and satisfactory achievement of completion criteria is confirmed. All planned products have been delivered with acceptable characteristics. Requirements are checked off and confirmed as satisfied, and the objectives of the project have been achieved. These processes generally involve all stakeholders and result in the documentation of client acceptance and any remaining known problem reports.

5.2. Closure Activities

[Pfl01: c12; Som05: c4]

After closure has been confirmed, archival of project materials takes place in line with stakeholder-agreed methods, location, and duration. The organization's measurement database is updated with final project data and post-project analyses are undertaken. A project post mortem is undertaken so that issues, problems, and opportunities encountered during the process (particularly via review and evaluation, see subarea 4 *Review and evaluation*) are analyzed, and lessons are drawn from the process and fed into organizational learning and improvement endeavors (see also the Software Engineering Process KA).

6. Software Engineering Measurement

[ISO 15939-02]

The importance of measurement and its role in better management practices is widely acknowledged, and so its importance can only increase in the coming years. Effective measurement has become one of the cornerstones of organizational maturity.

Key terms on software measures and measurement methods have been defined in [ISO15939-02] on the basis of the ISO international vocabulary of metrology [ISO93]. Nevertheless, readers will encounter terminology differences in the literature; for example, the term "metrics" is sometimes used in place of "measures."

This topic follows the international standard ISO/IEC 15939, which describes a process which defines the activities and tasks necessary to implement a software measurement process and includes, as well, a measurement information model.

6.1. Establish and Sustain Measurement Commitment

- Accept requirements for measurement. Each measurement endeavor should be guided by organizational objectives and driven by a set of measurement requirements established by the organization and the project. For example, an organizational objective might be "first-to-market with new products." [Fen98: c3,c13; Pre04: c22] This in turn might engender a requirement that factors contributing to this objective be measured so that projects might be managed to meet this objective.
  - Define scope of measurement. The organizational unit to which each measurement requirement is to be applied must be established. This may consist of a functional area, a single project, a single site, or even the whole enterprise. All subsequent measurement tasks related to this requirement should be within the defined scope. In addition, the stakeholders should be identified.
  - Commitment of management and staff to measurement. The commitment must be formally established, communicated, and supported by resources (see next item).
- Commit resources for measurement. The organization's commitment to measurement is an essential factor for success, as evidenced by assignment of resources for implementing the measurement process. Assigning resources includes allocation of responsibility for the various tasks of the measurement process (such as user, analyst, and librarian) and providing adequate funding, training, tools, and support to conduct the process in an enduring fashion.

6.2. Plan the Measurement Process

- Characterize the organizational unit. The organizational unit provides the context for measurement, so it is important to make this context explicit and to articulate the assumptions that it embodies and the constraints that it imposes. Characterization can be in terms of organizational processes, application domains, technology, and organizational interfaces. An organizational process model is also typically an element of the organizational unit characterization [ISO15939-02: 5.2.1].
- Identify information needs. Information needs are based on the goals, constraints, risks, and problems of the organizational unit. They may be derived from business, organizational, regulatory, and/or product objectives. They must be identified and prioritized. Then, a subset to be addressed must be selected and the results documented, communicated, and reviewed by stakeholders [ISO 15939-02: 5.2.2].
- Select measures. Candidate measures must be selected, with clear links to the information needs. Measures must then be selected based on the priorities of the information needs and other criteria such as cost of collection, degree of process disruption during collection, ease of analysis, ease of obtaining accurate, consistent data, and so on [ISO15939-02: 5.2.3 and Appendix C].
- Define data collection, analysis, and reporting procedures. This encompasses collection procedures and schedules, storage, verification, analysis, reporting, and configuration management of data [ISO15939-02: 5.2.4].
- Define criteria for evaluating the information products. Criteria for evaluation are influenced by the technical and business objectives of the organizational unit. Information products include those associated with the product being produced, as well as those associated with the processes being used to manage and measure the project [ISO15939-02: 5.2.5 and Appendices D, E].
- Review, approve, and provide resources for measurement tasks.
  - The measurement plan must be reviewed and approved by the appropriate stakeholders. This includes all data collection procedures, storage, analysis, and reporting procedures; evaluation criteria; schedules; and responsibilities. Criteria for reviewing these artifacts should have been established at the organizational unit level or higher and should be used as the basis for these reviews. Such criteria should take into consideration previous experience, availability of resources, and potential disruptions to projects when changes from current practices are proposed. Approval demonstrates commitment to the measurement process [ISO15939-02: 5.2.6.1 and Appendix F].
  - Resources should be made available for implementing the planned and approved measurement tasks. Resource availability may be staged in cases where changes are to be piloted before widespread deployment. Consideration should be paid to the resources necessary for successful deployment of new procedures or measures [ISO15939-02: 5.2.6.2].
- Acquire and deploy supporting technologies. This includes evaluation of available supporting technologies, selection of the most appropriate

technologies, acquisition of those technologies, and deployment of those technologies [ISO 15939-02: 5.2.7].

6.3. Perform the Measurement Process

- Integrate measurement procedures with relevant processes. The measurement procedures, such as data collection, must be integrated into the processes they are measuring. This may involve changing current processes to accommodate data collection or generation activities. It may also involve analysis of current processes to minimize additional effort and evaluation of the effect on employees to ensure that the measurement procedures will be accepted. Morale issues and other human factors need to be considered. In addition, the measurement procedures must be communicated to those providing the data, training may need to be provided, and support must typically be provided. Data analysis and reporting procedures must typically be integrated into organizational and/or project processes in a similar manner [ISO 15939-02: 5.3.1].
- Collect data. The data must be collected, verified, and stored [ISO 15939-02 :5.3.2].
- Analyze data and develop information products. Data may be aggregated, transformed, or recoded as part of the analysis process, using a degree of rigor appropriate to the nature of the data and the information needs. The results of this analysis are typically indicators such as graphs, numbers, or other indications that must be interpreted, resulting in initial conclusions to be presented to stakeholders. The results and conclusions must be reviewed, using a process defined by the organization (which may be formal or informal). Data providers and measurement users should participate in reviewing the data to ensure that they are meaningful and accurate, and that they can result in reasonable actions [ISO 15939-02: 5.3.3 and Appendix G].
- Communicate results. Information products must be documented and communicated to users and stakeholders [ISO 15939-02: 5.3.4].

6.4. Evaluate Measurement

- Evaluate information products. Evaluate information products against specified evaluation criteria and determine strengths and weaknesses of the information products. This may be performed by an internal process or an external audit and should include feedback from measurement users. Record lessons learned in an appropriate database [ISO 15939-02: 5.4.1 and Appendix D].
- Evaluate the measurement process. Evaluate the measurement process against specified evaluation criteria and determine the strengths and weaknesses of the process. This may be performed by an internal process or an external audit and should include feedback from measurement users. Record lessons learned in an appropriate database [ISO 15939-02: 5.4.1 and Appendix D].
- Identify potential improvements. Such improvements may be changes in the format of indicators, changes in units measured, or reclassification of categories. Determine the costs and benefits of potential improvements and select appropriate improvement actions. Communicate proposed improvements to the measurement process owner and stakeholders for review and approval. Also communicate lack of potential improvements if the analysis fails to identify improvements [ISO 15939-02: 5.4.2].

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	[Dor02]	[ISO15939-02]	[Fen98]	[Pfl01]	[Pre04]	[Rei02]	[Som05]	[Tha97]
1. Initiation and scope definition								
1.1 Determination and negotiation of requirements	v2c4			c4	c7		c5	
1.2 Feasibility analysis					c6		c6	
1.3 Process for the review and revision of requirements							c6	
2. Software Project Planning								
2.1 Process planning	v1c6,v2c7, v2c8			c2,c3	c2,c21	c1,c3,c5	c3,c4	c3,c4,c6
2.2 Determine deliverables				c3	c24			c4
23 Effort, schedule and cost estimation	v2c7		c12	c3	C23,c24	c5,c6	c4,c23	c5
2.4 Resource allocation				c3	c24	c8,c9	c4	c6,c7
2.5 Risk management	v2c7			c3	c25	c11	c4	c4
2.6 Quality management	v1c8,v2c3-c5				c26	c10	c24,c25	c9,c10
2.7 *Plan management*							c4	c4
3. Software Project Enactment								
3.1 Implementation of plans				c3			c4	
3.2 Supplier contract management							c4	
3.3 Implementation of measurement process			c13c,14		c22	c10,c12		c3,c10
3.4 Monitor process	v1c8,v2c2-c5,c7					c10	c25	c3,c9
3.5 Control process	v2c7					c10		c3,c9
3.6 Reporting						c10		c3,c10
4. Review and evaluation								
4.1 Determining satisfaction of requirements						c10		c3,c10
4.2 Reviewing and evaluating performance	v1c8,v2c3, c5			c8,c9		c10		c3,c10
5. Closure								
5.1 Determining closure	v1c8,v2c3, c5					c10		c3,c10
5.2 Closure activities				c12			c4	
6. Software Engineering Measurement		*						
6.1 Establish and sustain measurement commitment			c3,c13		c22			
6.2 Plan the measurement process		c5,C,D,E,F						
6.3 Perform the measurement process		c5,G						
6.4 Evaluate measurement		c5,D						

RECOMMENDED REFERENCES FOR SOFTWARE ENGINEERING MANAGEMENT

[Dor02] M. Dorfman and R.H. Thayer, eds., *Software Engineering*, IEEE Computer Society Press, 2002, Vol. 1, Chap. 6, 8, Vol. 2, Chap. 3, 4, 5, 7, 8.

[Fen98] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, second ed., International Thomson Computer Press, 1998, Chap. 1-14.

[ISO15939-02] ISO/IEC 15939:2002, *Software Engineering — Software Measurement Process*, ISO and IEC, 2002.

[Pfl01] S.L. Pfleeger, *Software Engineering: Theory and Practice*, second ed., Prentice Hall, 2001, Chap. 2-4, 8, 9, 12, 13.

[Pre04] R.S. Pressman, *Software Engineering: A Practitioner's Approach*, sixth ed., McGraw-Hill, 2004, Chap. 2, 6, 7, 22-26.

[Rei02] D.J. Reifer, ed., *Software Management*, IEEE Computer Society Press, 2002, Chap. 1-6, 7-12, 13.

[Som05] I. Sommerville, *Software Engineering*, seventh ed., Addison-Wesley, 2005, Chap. 3-6, 23-25.

[Tha97] R.H. Thayer, ed., *Software Engineering Project Management*, IEEE Computer Society Press, 1997, Chap. 1-10.

APPENDIX A. LIST OF FURTHER READINGS

(Adl99) T.R. Adler, J.G. Leonard, and R.K. Nordgren, "Improving Risk Management: Moving from Risk Elimination to Risk Avoidance," *Information and Software Technology*, vol. 41, 1999, pp. 29-34.

(Bai98) R. Baines, "Across Disciplines: Risk, Design, Method, Process, and Tools," *IEEE Software*, July/August 1998, pp. 61-64.

(Bin97) R.V. Binder, "Can a Manufacturing Quality Model Work for Software?" *IEEE Software*, September/October 1997, pp. 101-102,105.

(Boe97) B.W. Boehm and T. DeMarco, "Software Risk Management," *IEEE Software*, May/June 1997, pp. 17-19.

(Bri96) L.C. Briand, S. Morasca, and V.R. Basili, "Property-Based Software Engineering Measurement," *IEEE Transactions on Software Engineering*, vol. 22, iss. 1, 1996, pp. 68-86.

(Bri96a) L. Briand, K.E. Emam, and S. Morasca, "On the Application of Measurement Theory in Software Engineering," *Empirical Software Engineering*, vol. 1, 1996, pp. 61-88.

(Bri97) L.C. Briand, S. Morasca, and V.R. Basili, "Response to: Comments on 'Property-based Software Engineering Measurement: Refining the Addivity Properties,'" *IEEE Transactions on Software Engineering*, vol. 23, iss. 3, 1997, pp. 196-197.

(Bro87) F.P.J. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, Apr. 1987, pp. 10-19.

(Cap96) J. Capers, *Applied Software Measurement: Assuring Productivity and Quality*, second ed., McGraw-Hill, 1996.

(Car97) M.J. Carr, "Risk Management May Not Be For Everyone," *IEEE Software*, May/June 1997, pp. 21-24.

(Cha96) R.N. Charette, "Large-Scale Project Management Is Risk Management," *IEEE Software*, July 1996, pp. 110-117.

(Cha97) R.N. Charette, K.M. Adams, and M.B. White, "Managing Risk in Software Maintenance," *IEEE Software*, May/June 1997, pp. 43-50.

(Col96) B. Collier, T. DeMarco,and P. Fearey, "A Defined Process for Project Postmortem Review," *IEEE Software*, July 1996, pp. 65-72.

(Con97) E.H. Conrow and P.S. Shishido, "Implementing Risk Management on Software Intensive Projects," *IEEE Software*, May/June 1997, pp. 83-89.

(Dav98) A.M. Davis, "Predictions and Farewells," *IEEE Software*, July/August 1998, pp. 6-9.

(Dem87) T. DeMarco and T. Lister, *Peopleware: Productive Projects and Teams*, Dorset House Publishing, 1987.

(Dem96) T. DeMarco and A. Miller, "Managing Large Software Projects," *IEEE Software*, July 1996, pp. 24-27.

(Fav98) J. Favaro and S.L. Pfleeger, "Making Software Development Investment Decisions," *ACM SIGSoft Software Engineering Notes*, vol. 23, iss. 5, 1998, pp. 69-74.

(Fay96) M.E. Fayad and M. Cline, "Managing Object-Oriented Software Development," *Computer*, September 1996, pp. 26-31.

(Fen98) N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, second ed., International Thomson Computer Press, 1998.

(Fle99) R. Fleming, "A Fresh Perspective on Old Problems," *IEEE Software*, January/February 1999, pp. 106-113.

(Fug98) A. Fuggetta et al., "Applying GQM in an Industrial Software Factory," *ACM Transactions on Software Engineering and Methodology*, vol. 7, iss. 4, 1998, pp. 411-448.

(Gar97) P.R. Garvey, D.J. Phair, and J.A. Wilson, "An Information Architecture for Risk Assessment and Management," *IEEE Software*, May/June 1997, pp. 25-34.

(Gem97) A. Gemmer, "Risk Management: Moving beyond Process," *Computer*, May 1997, pp. 33-43.

(Gla97) R.L. Glass, "The Ups and Downs of Programmer Stress," *Communications of the ACM*, vol. 40, iss. 4, 1997, pp. 17-19.

(Gla98) R.L. Glass, "Short-Term and Long-Term Remedies for Runaway Projects," *Communications of the ACM*, vol. 41, iss. 7, 1998, pp. 13-15.

(Gla98a) R.L. Glass, "How Not to Prepare for a Consulting Assignment, and Other Ugly Consultancy Truths," *Communications of the ACM*, vol. 41, iss. 12, 1998, pp. 11-13.

(Gla99) R.L. Glass, "The Realities of Software Technology Payoffs," *Communications of the ACM*, vol. 42, iss. 2, 1999, pp. 74-79.

(Gra99) R. Grable et al., "Metrics for Small Projects: Experiences at the SED," *IEEE Software*, March/April 1999, pp. 21-29.

(Gra87) R.B. Grady and D.L. Caswell, *Software Metrics: Establishing A Company-Wide Program*. Prentice Hall, 1987.

(Hal97) T. Hall and N. Fenton, "Implementing Effective Software Metrics Programs," *IEEE Software*, March/April 1997, pp. 55-64.

(Hen99) S.M. Henry and K.T. Stevens, "Using Belbin's Leadership Role to Improve Team Effectiveness: An Empirical Investigation," *Journal of Systems and Software*, vol. 44, 1999, pp. 241-250.

(Hoh99) L. Hohmann, "Coaching the Rookie Manager," *IEEE Software*, January/February 1999, pp. 16-19.

(Hsi96) P. Hsia, "Making Software Development Visible," *IEEE Software*, March 1996, pp. 23-26.

(Hum97) W.S. Humphrey, *Managing Technical People: Innovation, Teamwork, and the Software Process*: Addison-Wesley, 1997.

(IEEE12207.0-96) IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.

(Jac98) M. Jackman, "Homeopathic Remedies for Team Toxicity," *IEEE Software*, July/August 1998, pp. 43-45.

(Kan97) K. Kansala, "Integrating Risk Assessment with Cost

Estimation," *IEEE Software*, May/June 1997, pp. 61-67.
(Kar97) J. Karlsson and K. Ryan, "A Cost-Value Aproach for Prioritizing Requirements," *IEEE Software*, September/October 1997, pp. 87-74.
(Kar96) D.W. Karolak, *Software Engineering Risk Management*, IEEE Computer Society Press, 1996.
(Kau99) K. Kautz, "Making Sense of Measurement for Small Organizations," *IEEE Software*, March/April 1999, pp. 14-20.
(Kei98) M. Keil et al., "A Framework for Identifying Software Project Risks," *Communications of the ACM*, vol. 41, iss. 11, 1998, pp. 76-83.
(Ker99) B. Kernighan and R. Pike, "Finding Performance Improvements," *IEEE Software*, March/April 1999, pp. 61-65.
(Kit97) B. Kitchenham and S. Linkman, "Estimates, Uncertainty, and Risk," *IEEE Software*, May/June 1997, pp. 69-74.
(Lat98) F. v. Latum et al., "Adopting GQM-Based Measurement in an Industrial Environment," *IEEE Software*, January-February 1998, pp. 78-86.
(Leu96) H.K.N. Leung, "A Risk Index for Software Producers," *Software Maintenance: Research and Practice*, vol. 8, 1996, pp. 281-294.
(Lis97) T. Lister, "Risk Management Is Project Management for Adults," *IEEE Software*, May/June 1997, pp. 20-22.
(Mac96) K. Mackey, "Why Bad Things Happen to Good Projects," *IEEE Software*, May 1996, pp. 27-32.
(Mac98) K. Mackey, "Beyond Dilbert: Creating Cultures that Work," *IEEE Software*, January/February 1998, pp. 48-49.
(Mad97) R.J. Madachy, "Heuristic Risk Assessment Using Cost Factors," *IEEE Software*, May/June 1997, pp. 51-59.
(McC96) S.C. McConnell, *Rapid Development: Taming Wild Software Schedules*, Microsoft Press, 1996.
(McC97) S.C. McConnell, *Software Project Survival Guide*, Microsoft Press, 1997.
(McC99) S.C. McConnell, "Software Engineering Principles," *IEEE Software*, March/April 1999, pp. 6-8.
(Moy97) T. Moynihan, "How Experienced Project Managers Assess Risk," *IEEE Software*, May/June 1997, pp. 35-41.
(Ncs98) P. Ncsi, "Managing OO Projects Better," *IEEE Software*, July/August 1998, pp. 50-60.
(Nol99) A.J. Nolan, "Learning From Success," *IEEE Software*, January/February 1999, pp. 97-105.
(Off97) R.J. Offen and R. Jeffery, "Establishing Software Measurement Programs," *IEEE Software*, March/April 1997, pp. 45-53.
(Par96) K.V.C. Parris, "Implementing Accountability," *IEEE Software*, July/August 1996, pp. 83-93.
(Pfl97) S.L. Pfleeger, "Assessing Measurement (Guest Editor's Introduction)," *IEEE Software*, March/April 1997, pp. 25-26.
(Pfl97a) S.L. Pfleeger et al., "Status Report on Software Measurement," *IEEE Software*, March/April 1997, pp. 33-43.
(Put97) L.H. Putman and W. Myers, *Industrial Strength Software — Effective Management Using Measurement*, IEEE Computer Society Press, 1997.
(Rob99) P.N. Robillard, "The Role of Knowledge in Software Development," *Communications of the ACM*, vol. 42, iss. 1, 1999, pp. 87-92.
(Rod97) A.G. Rodrigues and T.M. Williams, "System Dynamics in Software Project Management: Towards the Development of a Formal Integrated Framework," *European Journal of Information Systems*, vol. 6, 1997, pp. 51-66.
(Rop97) J. Ropponen and K. Lyytinen, "Can Software Risk Management Improve System Development: An Exploratory Study," *European Journal of Information Systems*, vol. 6, 1997, pp. 41-50.
(Sch99) C. Schmidt et al., "Disincentives for Communicating Risk: A Risk Paradox," *Information and Software Technology*, vol. 41, 1999, pp. 403-411.
(Sco92) R.L. v. Scoy, "Software Development Risk: Opportunity, Not Problem," Software Engineering Institute, Carnegie Mellon University CMU/SEI-92-TR-30, 1992.
(Sla98) S.A. Slaughter, D.E. Harter, and M.S. Krishnan, "Evaluating the Cost of Software Quality," *Communications of the ACM*, vol. 41, iss. 8, 1998, pp. 67-73.
(Sol98) R. v. Solingen, R. Berghout, and F. v. Latum, "Interrupts: Just a Minute Never Is," *IEEE Software*, September/October 1998, pp. 97-103.
(Whi95) N. Whitten, *Managing Software Development Projects: Formulas for Success*, Wiley, 1995.
(Wil99) B. Wiley, *Essential System Requirements: A Practical Guide to Event-Driven Methods*, Addison-Wesley, 1999.
(Zel98) M.V. Zelkowitz and D.R. Wallace, "Experimental Models for Validating Technology," *Computer*, vol. 31, iss. 5, 1998, pp. 23-31.

APPENDIX B. LIST OF STANDARDS

(IEEE610.12-90) IEEE Std 610.12-1990 (R2002), *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1990.

(IEEE12207.0-96) IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.

(ISO15939-02) ISO/IEC 15939:2002, *Software Engineering-Software Measurement Process*, ISO and IEC, 2002.

(PMI00) Project Management Institute Standards Committee, *A Guide to the Project Management Body of Knowledge (PMBOK)*, Project Management Institute, 2000.

CHAPTER 9

SOFTWARE ENGINEERING PROCESS

ACRONYMS	
CMMI	Capability Maturity Model Integration
EF	Experience Factory
FP	Function Point
HRM	Human Resources Management
IDEAL	Initiating-Diagnosing-Establishing-Acting-Leaning (model)
OMG	Object Management Group
QIP	Quality Improvement Paradigm
SCAMPI	CMM Based Appraisal for Process Improvement using the CMMI
SCE	Software Capability Evaluation
SEPG	Software Engineering Process Group

INTRODUCTION

The Software Engineering Process KA can be examined on two levels. The first level encompasses the technical and managerial activities within the software life cycle processes that are performed during software acquisition, development, maintenance, and retirement. The second is the meta-level, which is concerned with the definition, implementation, assessment, measurement, management, change, and improvement of the software life cycle processes themselves. The first level is covered by the other KAs in the Guide. This KA is concerned with the second.

The term "software engineering process" can be interpreted in different ways, and this may cause confusion.

- One meaning, where the word *the* is used, as in *the* software engineering process, could imply that there is only one right way of performing software engineering tasks. This meaning is avoided in the Guide, because no such process exists. Standards such as IEEE12207 speak of software engineering *processes*, meaning that there are many processes involved, such as Development Process or Configuration Management Process.
- A second meaning refers to the general discussion of processes related to software engineering. This is the meaning intended in the title of this KA, and the one most often intended in the KA description.
- Finally, a third meaning could signify the actual set of activities performed within an organization, which could be viewed as one process, especially from within the organization. This meaning is used in the KA in a very few instances.

This KA applies to any part of the management of software life cycle processes where procedural or technological change is being introduced for process or product improvement.

Software engineering process is relevant not only to large organizations. On the contrary, process-related activities can, and have been, performed successfully by small organizations, teams, and individuals.

The objective of managing software life cycle processes is to implement new or better processes in actual practices, be they individual, project, or organizational.

This KA does not explicitly address human resources management (HRM), for example, as embodied in the People CMM (Cur02) and systems engineering processes [ISO1528-028; IEEE 1220-98].

It should also be recognized that many software engineering process issues are closely related to other disciplines, such as management, albeit sometimes using a different terminology.

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING PROCESS

Figure 1 shows the breakdown of topics in this KA.

1. Process Implementation and Change

This subarea focuses on organizational change. It describes the infrastructure, activities, models, and practical considerations for process implementation and change.

Described here is the situation in which processes are deployed for the first time (for example, introducing an inspection process within a project or a method covering the complete life cycle), and where current processes are changed (for example, introducing a tool, or optimizing a procedure). This can also be termed *process evolution*. In both instances, existing practices have to be modified. If the modifications are extensive, then changes in the organizational culture may also be necessary.

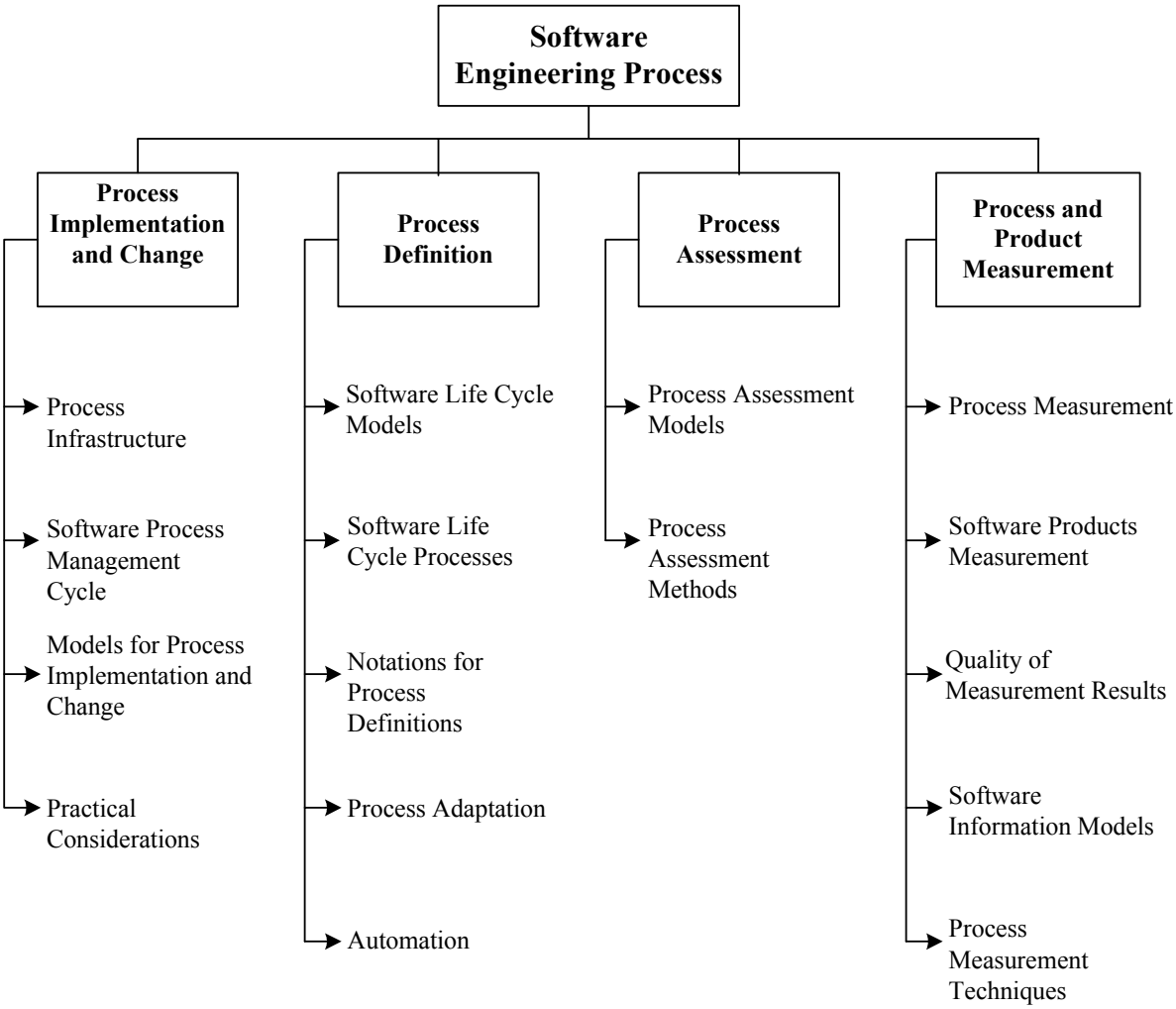


Figure 1 Breakdown of topics for the Software Engineering Process KA

1.1. Process Infrastructure

[IEEE12207.0-96; ISO15504-98; SEL96]

This topic includes the knowledge related to the software engineering process infrastructure.

To establish software life cycle processes, it is necessary to have an appropriate infrastructure in place, meaning that the resources must be available (competent staff, tools, and funding) and the responsibilities assigned. When these tasks have been completed, it is an indication of management's commitment to, and ownership of, the software engineering process effort. Various committees may have to be established, such as a steering committee to oversee the software engineering process effort.

A description of an infrastructure for process improvement in general is provided in [McF96]. Two main types of infrastructure are used in practice: the Software Engineering Process Group and the Experience Factory.

1.1.1. Software Engineering Process Group (SEPG)

The SEPG is intended to be the central focus of software engineering process improvement, and it has a number of responsibilities in terms of initiating and sustaining it. These are described in [Fow90].

1.1.2. Experience Factory (EF)

The concept of the EF separates the project organization (the software development organization, for example) from the improvement organization. The project organization focuses on the development and maintenance of software, while the EF is concerned with software engineering process improvement.

The EF is intended to institutionalize the collective learning of an organization by developing, updating, and delivering to the project organization *experience packages* (for example, guides, models, and training courses), also referred to as *process assets*. The project organization offers the EF their products, the plans used in their development, and the data gathered during development and operation. Examples of experience packages are presented in [Bas92].

1.2. Software Process Management Cycle

[Bas92; Fow90; IEEE12207.0-96; ISO15504-98; McF96; SEL96]

The management of software processes consists of four activities sequenced in an iterative cycle allowing for continuous feedback and improvement of the software process:

- The *Establish Process Infrastructure* activity consists of establishing commitment to process implementation and change (including obtaining management buy-in) and putting in place an appropriate infrastructure (resources and responsibilities) to make it happen.
- The goal of the *Planning* activity is to understand the current business objectives and process needs of the individual, project, or organization, to identify its strengths and weaknesses, and to make a plan for process implementation and change.
- The goal of *Process Implementation and Change* is to execute the plan, deploy new processes (which may involve, for example, the deployment of tools and training of staff), and/or change existing processes.
- *Process Evaluation* is concerned with finding out how well the implementation and change went, whether or not the expected benefits materialized. The results are then used as input for subsequent cycles.

1.3. Models For Process Implementation And Change

Two general models that have emerged for driving process implementation and change are the Quality Improvement Paradigm (QIP) [SEL96] and the IDEAL model [McF96]. The two paradigms are compared in [SEL96]. Evaluation of process implementation and change outcomes can be qualitative or quantitative.

1.4. Practical Considerations

Process implementation and change constitute an instance of organizational change. Most successful organizational change efforts treat the change as a project in its own right, with appropriate plans, monitoring, and review.

Guidelines about process implementation and change within software engineering organizations, including action planning, training, management sponsorship, commitment, and the selection of pilot projects, and which cover both processes and tools, are given in [Moi98; San98; Sti99]. Empirical studies on success factors for process change are reported in (ElE99a).

The role of change agents in this activity is discussed in (Hut94). Process implementation and change can also be seen as an instance of consulting (either internal or external).

One can also view organizational change from the perspective of technology transfer (Rog83). Software engineering articles which discuss technology transfer and the characteristics of recipients of new technology (which could include process-related technologies) are (Pfl99; Rag89).

There are two ways of approaching the evaluation of process implementation and change, either in terms of changes to the process itself or in terms of changes to the process outcomes (for example, measuring the return on investment from making the change). A pragmatic look at what can be achieved from such evaluation studies is given in (Her98).

Overviews of how to evaluate process implementation and change, and examples of studies that do so, can be found in [Gol99], (Kit98; Kra99; McG94).

2. Process Definition

A process definition can be a procedure, a policy, or a standard. Software life cycle processes are defined for a number of reasons, including increasing the quality of the product, facilitating human understanding and communication, supporting process improvement, supporting process management, providing automated process guidance, and providing automated execution support. The types of process definitions required will depend, at least partially, on the reason for the definition.

It should also be noted that the context of the project and organization will determine the type of process definition that is most useful. Important variables to consider include the nature of the work (for example, maintenance or development), the application domain, the life cycle model, and the maturity of the organization.

2.1. Software Life Cycle Models

[Pfl01:c2; IEEE12207.0-96]

Software life cycle models serve as a high-level definition of the phases that occur during development. They are not aimed at providing detailed definitions but at highlighting the key activities and their interdependencies. Examples of software life cycle models are the waterfall model, the

throwaway prototyping model, evolutionary development, incremental/iterative delivery, the spiral model, the reusable software model, and automated software synthesis. Comparisons of these models are provided in [Com97], (Dav88), and a method for selecting among many of them in (Ale91).

2.2. Software Life Cycle Processes

Definitions of software life cycle processes tend to be more detailed than software life cycle models. However, software life cycle processes do not attempt to order their processes in time. This means that, in principle, the software life cycle processes can be arranged to fit any of the software life cycle models. The main reference in this area is IEEE/EIA 12207.0: *Information Technology — Software Life Cycle Processes* [IEEE 12207.0-96].

The IEEE 1074:1997 standard on developing life cycle processes also provides a list of processes and activities for software development and maintenance [IEEE1074-97], as well as a list of life cycle activities which can be mapped into processes and organized in the same way as any of the software life cycle models. In addition, it identifies and links other IEEE software standards to these activities. In principle, IEEE Std 1074 can be used to build processes conforming to any of the life cycle models. Standards which focus on maintenance processes are IEEE Std 1219-1998 and ISO 14764: 1998 [IEEE 1219-98].

Other important standards providing process definitions include

- IEEE Std 1540: Software Risk Management (IEEE1540-01)
- IEEE Std 1517: Software Reuse Processes (IEEE 1517-99)
- ISO/IEC 15939: Software Measurement Process [ISO15939-02]. See also the Software Engineering Management KA for a detailed description of this process.

In some situations, software engineering processes must be defined taking into account the organizational processes for quality management. ISO 9001 [ISO9001-00] provides requirements for quality management processes, and ISO/IEC 90003 interprets those requirements for organizations developing software (ISO90003-04).

Some software life cycle processes emphasize rapid delivery and strong user participation, namely agile methods such as Extreme Programming [Bec99]. A form of the selection problem concerns the choice along the agile plan-driven method axis. A risk-based approach to making that decision is described in (Boe03a).

2.3. Notations for Process Definitions

Processes can be defined at different levels of abstraction (for example, generic definitions vs. adapted definitions, descriptive vs. prescriptive vs. proscriptive) [Pfl01]. Various elements of a process can be defined, for example, activities, products (artifacts), and resources. Detailed frameworks which structure the types of information required to define processes are described in (Mad94).

There are a number of notations being used to define processes (SPC92). A key difference between them is in the type of information the frameworks mentioned above define, capture, and use. The software engineer should be aware of the following approaches: data flow diagrams, in terms of process purpose and outcomes [ISO15504-98], as a list of processes decomposed into constituent activities and tasks defined in natural language [IEEE12207.0-96], Statecharts (Har98), ETVX (Rad85), Actor-Dependency modeling (Yu94), SADT notation (Mcg93), Petri nets (Ban95); IDEF0 (IEEE 1320.1-98), and rule-based (Bar95). More recently, a process modeling standard has been published by the OMG which is intended to harmonize modeling notations. This is termed the SPEM (Software Process Engineering Meta-Model) specification. [OMG02]

2.4. Process Adaptation

[IEEE 12207.0-96; ISO15504-98; Joh99]

It is important to note that predefined processes—even standardized ones—must be adapted to local needs, for example, organizational context, project size, regulatory requirements, industry practices, and corporate cultures. Some standards, such as IEEE/EIA 12207, contain mechanisms and recommendations for accomplishing the adaptation.

2.5. Automation

[Pfl01:c2]

Automated tools either support the execution of the process definitions or they provide guidance to humans performing the defined processes. In cases where process analysis is performed, some tools allow different types of simulations (for example, discrete event simulation).

In addition, there are tools which support each of the above process definition notations. Moreover, these tools can execute the process definitions to provide automated support to the actual processes, or to fully automate them in some instances. An overview of process-modeling tools can be found in [Fin94] and of process-centered environments in (Gar96). Work on applying the Internet to the provision of real-time process guidance is described in (Kel98).

3. Process Assessment

Process assessment is carried out using both an assessment model and an assessment method. In some instances, the term “appraisal” is used instead of assessment, and the term “capability evaluation” is used when the appraisal is for the purpose of awarding a contract.

3.1. Process Assessment Models

An assessment model captures what is recognized as good practices. These practices may pertain to technical software

engineering activities only, or may also refer to, for example, management, systems engineering, and human resources management activities as well.

ISO/IEC 15504 [ISO15504-98] defines an exemplar assessment model and conformance requirements on other assessment models. Specific assessment models available and in use are SW-CMM (SEI95), CMMI [SEI01], and Bootstrap [Sti99]. Many other capability and maturity models have been defined—for example, for design, documentation, and formal methods, to name a few. ISO 9001 is another common assessment model which has been applied by software organizations (ISO9001-00).

A maturity model for systems engineering has also been developed, which would be useful where a project or organization is involved in the development and maintenance of systems, including software (EIA/IS731-99).

The applicability of assessment models to small organizations is addressed in [Joh99; San98].

There are two general architectures for an assessment model that make different assumptions about the order in which processes must be assessed: continuous and staged (Pau94). They are very different, and should be evaluated by the organization considering them to determine which would be the most pertinent to their needs and objectives.

3.2. Process Assessment Methods

[Gol99]

In order to perform an assessment, a specific assessment method needs to be followed to produce a quantitative score which characterizes the capability of the process (or maturity of the organization).

The CBA-IPI assessment method, for example, focuses on process improvement (Dun96), and the SCE method focuses on evaluating the capability of suppliers (Bar95). Both of these were developed for the SW-CMM. Requirements on both types of methods which reflect what are believed to be good assessment practices are provided in [ISO15504-98], (Mas95). The SCAMPI methods are geared toward CMMI assessments [SEI01]. The activities performed during an assessment, the distribution of effort on these activities, as well as the atmosphere during an assessment are different when they are for improvement than when they are for a contract award.

There have been criticisms of process assessment models and methods, for example (Fay97; Gra98). Most of these criticisms have been concerned with the empirical evidence supporting the use of assessment models and methods. However, since the publication of these articles, there has been some systematic evidence supporting the efficacy of process assessments. (Cla97; Ele00; Ele00a; Kri99)

4. Process and Product Measurement

While the application of measurement to software engineering can be complex, particularly in terms of modeling and analysis methods, there are several aspects of software engineering measurement which are fundamental and which underlie many of the more advanced measurement and analysis processes. Furthermore, achievement of process and product improvement efforts can only be assessed if a set of baseline measures has been established.

Measurement can be performed to support the initiation of process implementation and change or to evaluate the consequences of process implementation and change, or it can be performed on the product itself.

Key terms on software measures and measurement methods have been defined in ISO/IEC 15939 on the basis of the ISO international vocabulary of metrology. ISO/IEC 15359 also provides a standard process for measuring both process and product characteristics. [VIM93]

Nevertheless, readers will encounter terminological differences in the literature; for example, the term “metric” is sometimes used in place of “measure.”

4.1. Process Measurement

[ISO15539-02]

The term “process measurement” as used here means that quantitative information about the process is collected, analyzed, and interpreted. Measurement is used to identify the strengths and weaknesses of processes and to evaluate processes after they have been implemented and/or changed.

Process measurement may serve other purposes as well. For example, process measurement is useful for managing a software engineering project. Here, the focus is on process measurement for the purpose of process implementation and change.

The path diagram in Figure 2 illustrates an important assumption made in most software engineering projects, which is that usually the process has an impact on project outcomes. The context affects the relationship between the process and process outcomes. This means that this process-to-process outcome relationship depends on the context.

Not every process will have a positive impact on all outcomes. For example, the introduction of software inspections may reduce testing effort and cost, but may increase elapsed time if each inspection introduces long delays due to the scheduling of large inspection meetings. (Vot93) Therefore, it is preferable to use multiple process outcome measures which are important to the organization’s business.

While some effort can be made to assess the utilization of tools and hardware, the primary resource that needs to be

managed in software engineering is personnel. As a result, the main measures of interest are those related to the productivity of teams or processes (for example, using a measure of function points produced per unit of person-effort) and their associated levels of experience in software engineering in general, and perhaps in particular technologies. [Fen98: c3, c11; Som05: c25]

Process outcomes could, for example, be product quality (faults per KLOC (Kilo Lines of Code) or per Function Point (FP)), maintainability (the effort to make a certain type of change), productivity (LOC (Lines of Code) or Function Points per person-month), time-to-market, or customer satisfaction (as measured through a customer survey). This relationship depends on the particular context (for example, size of the organization or size of the project).

In general, we are most concerned about process outcomes. However, in order to achieve the process outcomes that we desire (for example, better quality, better maintainability, greater customer satisfaction), we have to implement the appropriate process.

Of course, it is not only the process that has an impact on outcomes. Other factors, such as the capability of the staff and the tools that are used, play an important role. When evaluating the impact of a process change, for example, it is important to factor out these other influences. Furthermore, the extent to which the process is institutionalized (that is, process fidelity) is important, as it may explain why "good" processes do not always give the desired outcomes in a given situation.

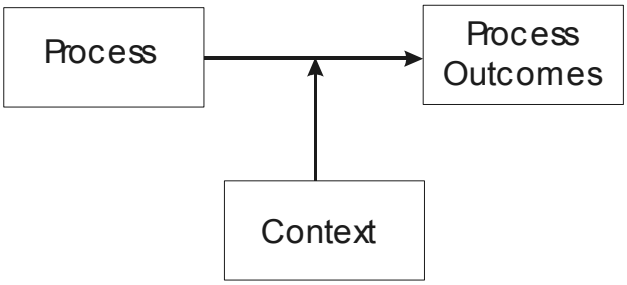


Figure 2 Path diagram showing the relationship between process and outcomes (results).

Software Product Measurement
[ISO9126-01]

Software product measurement includes, notably, the measurement of product size, product structure, and product quality.

4.1.1. Size measurement

Software product size is most often assessed by measures of length (for example, lines of source code in a module, pages in a software requirements specification document), or functionality (for example, function points in a specification). The principles of functional size measurement are provided in IEEE Std 14143.1. International standards for functional size measurement methods include ISO/IEC 19761, 20926, and 20968 [IEEE 14143.1-00; ISO19761-03; ISO20926-03; ISO20968-02].

4.1.2. Structure measurement

A diverse range of measures of software product structure may be applied to both high- and low-level design and code artifacts to reflect control flow (for example the cyclomatic number, code knots), data flow (for example, measures of slicing), nesting (for example, the nesting polynomial measure, the BAND measure), control structures (for example, the vector measure, the NPATH measure), and modular structure and interaction (for example, information flow, tree-based measures, coupling and cohesion). [Fen98: c8; Pre04: c15]

4.1.3. Quality measurement

As a multi-dimensional attribute, quality measurement is less straightforward to define than those above. Furthermore, some of the dimensions of quality are likely to require measurement in qualitative rather than quantitative form. A more detailed discussion of software quality measurement is provided in the Software Quality KA, topic 3.4. ISO models of software product quality and of related measurements are described in ISO 9126, parts 1 to 4 [ISO9126-01]. [Fen98: c9,c10; Pre04: c15; Som05: c24]

4.2. Quality Of Measurement Results

The quality of the measurement results (accuracy, reproducibility, repeatability, convertibility, random measurement errors) is essential for the measurement programs to provide effective and bounded results. Key characteristics of measurement results and related quality of measuring instruments have been defined in the ISO International vocabulary on metrology. [VIM93]

The theory of measurement establishes the foundation on which meaningful measurements can be made. The theory of measurement and scale types is discussed in [Kan02]. Measurement is defined in the theory as "the assignment of numbers to objects in a systematic way to represent properties of the object."

An appreciation of software measurement scales and the implications of each scale type in relation to the subsequent selection of data analysis methods is especially important. [Abr96; Fen98: c2; Pfl01: c11] Meaningful scales are related to a classification of scales. For those, measurement theory provides a succession of more and more constrained ways of assigning the measures. If the numbers assigned are merely to provide labels to classify the objects, they are called *nominal*. If they are assigned in a way that ranks the

objects (for example, good, better, best), they are called *ordinal*. If they deal with magnitudes of the property relative to a defined measurement unit, they are called *interval* (and the intervals are uniform between the numbers unless otherwise specified, and are therefore additive). Measurements are at the *ratio* level if they have an absolute zero point, so ratios of distances to the zero point are meaningful.

4.3. Software Information Models

As the data are collected and the measurement repository is populated, we become able to build models using both data and knowledge.

These models exist for the purposes of analysis, classification, and prediction. Such models need to be evaluated to ensure that their levels of accuracy are sufficient and that their limitations are known and understood. The refinement of models, which takes place both during and after projects are completed, is another important activity.

4.3.1. Model building

Model building includes both calibration and evaluation of the model. The goal-driven approach to measurement informs the model building process to the extent that models are constructed to answer relevant questions and achieve software improvement goals. This process is also influenced by the implied limitations of particular measurement scales in relation to the choice of analysis method. The models are calibrated (by using particularly relevant observations, for example, recent projects, projects using similar technology) and their effectiveness is evaluated (for example, by testing their performance on holdout samples). [Fen98: c4,c6,c13;Pfl01: c3,c11,c12; Som05: c25]

4.3.2. Model implementation

Model implementation includes both interpretation and refinement of models–the calibrated models are applied to the process, their outcomes are interpreted and evaluated in the context of the process/project, and the models are then refined where appropriate. [Fen98: c6; Pfl01: c3,c11,c12; Pre04: c22; Som05: c25]

4.4. *Process Measurement Techniques*

Measurement techniques may be used to analyze software engineering processes and to identify strengths and weaknesses. This can be performed to initiate process implementation and change, or to evaluate the consequences of process implementation and change.

The quality of measurement results, such as accuracy, repeatability, and reproducibility, are issues in the measurement of software engineering processes, since there are both instrument-based and judgmental measurements, as, for example, when assessors assign scores to a particular process. A discussion and method for achieving quality of measurement are presented in [Gol99].

Process measurement techniques have been classified into two general types: analytic and benchmarking. The two types of techniques can be used together since they are based on different types of information. (Car91)

4.4.1. Analytical techniques

The analytical techniques are characterized as relying on "quantitative evidence to determine where improvements are needed and whether an improvement initiative has been successful." The analytical type is exemplified by the Quality Improvement Paradigm (QIP) consisting of a cycle of understanding, assessing, and packaging [SEL96]. The techniques presented next are intended as other examples of analytical techniques, and reflect what is done in practice. [Fen98; Mus99], (Lyu96; Wei93; Zel98) Whether or not a specific organization uses all these techniques will depend, at least partially, on its maturity.

- *Experimental Studies*: Experimentation involves setting up controlled or quasi experiments in the organization to evaluate processes. (McG94) Usually, a new process is compared with the current process to determine whether or not the former has better process outcomes.

  Another type of experimental study is process simulation. This type of study can be used to analyze process behavior, explore process improvement potentials, predict process outcomes if the current process is changed in a certain way, and control process execution. Initial data about the performance of the current process need to be collected, however, as a basis for the simulation.

- *Process Definition Review* is a means by which a process definition (either a descriptive or a prescriptive one, or both) is reviewed, and deficiencies and potential process improvements identified. Typical examples of this are presented in (Ban95; Kel98). An easy operational way to analyze a process is to compare it to an existing standard (national, international, or professional body), such as IEEE/EIA 12207.0[IEEE12207.0-96]. With this approach, quantitative data are not collected on the process, or, if they are, they play a supportive role. The individuals performing the analysis of the process definition use their knowledge and capabilities to decide what process changes would potentially lead to desirable process outcomes. Observational studies can also provide useful feedback for identifying process improvements. (Agr99)

- *Orthogonal Defect Classification* is a technique which can be used to link faults found with potential causes. It relies on a mapping between fault types and fault triggers. (Chi92; Chi96) The IEEE Standard on the classification of faults (or anomalies) may be useful in this context (*IEEE Standard for the Classification of Software Anomalies* (IEEE1044-93).

- *Root Cause Analysis* is another common analytical technique which is used in practice. This involves tracing back from detected problems (for example, faults) to identify the process causes, with the aim of changing the process to avoid these problems in the future. Examples for different types of processes are described in (Col93; Ele97; Nak91).
- The Orthogonal Defect Classification technique described above can be used to find catagories in which many problems exist, at which point they can be analyzed. Orthogonal Defect Classification is thus a technique used to make a quantitative selection for where to apply Root Cause Analysis.
- *Statistical Process Control* is an effective way to identify stability, or the lack of it, in the process through the use of control charts and their interpretations. A good introduction to SPC in the context of software engineering is presented in (Flo99).
- The *Personal Software Process* defines a series of improvements to an individual's development practices in a specified order [Hum95]. It is 'bottom-up' in the sense that it stipulates personal data collection and improvements based on the data interpretations.

4.4.2. Benchmarking techniques

The second type of technique, benchmarking, "depends on identifying an 'excellent' organization in a field and documenting its practices and tools." Benchmarking assumes that if a less-proficient organization adopts the practices of the excellent organization, it will also become excellent. Benchmarking involves assessing the maturity of an organization or the capability of its processes. It is exemplified by the software process assessment work. A general introductory overview of process assessments and their application is provided in (Zah98).

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	[Abr96]	[Bas92]	[Bec99]	[Boe03]	[Com97]	[Fen98]	[Fin94]	[Fow90]	[Gol99]	[Joh99]	[McF96]	[Moi98]	[Mus99]	[OMG02]	[Pfl01]	[Pre04]	[San98]	[SEI01]	[SEL96]	[Som05]	[Sti99]
1. Process Implementation and Change																					
1.1 Process Infrastructure		*						*			*								*		
1.2 Software Process Management Cycle		*						*			*								*		
1.3 Models for Process Implementation and Change											*								*		
1.4 Practical Considerations									*			*					*				*
2. Process Definition																					
2.1Software Life Cycle Models					*										c2						
2.2 Software Life Cycle Processes			*	*																	
2.3 Notations for Process Definitions														*	c2						
2.4 Process Adaptation										*											
2.5 Automation							*								c2						
3. Process Assessment																					
3.1 Process Assessment Models										*							*	*			*
3.2 Process Assessment Methods									*									*			
4. Measurement																					
4.1 Process Measurement						c3,c11														c25	
4.2 Software Products Measurement						c8-c10										c15				c24	
4.3 Quality of Measurement Results	*					c2									c11						
4.4 Software Information Models																					
Model building						c4,c6,c13									c3,c11,c12					c25	
Model Implementation						c6									c3,c11,c12	c22			*	c25	
4.5 Process Measurement Techniques						*			*				*								

	ISO 9001	ISO 9000-3	ISO 9126	ISO 14764	ISO 15504	ISO 15288	ISO 15939	ISO 19761	ISO 20926	ISO 20968	ISO VIM	IEEE 1044	IEEE 1061	IEEE 1074	IEEE 1219	IEEE 1517	IEEE 1540	IEEE 12207	IEEE 14143.1
1. Process Implementation and Change																			
1.1 Process Infrastructure					*													*	
1.2 Software Process Management Cycle					*													*	
1.3 Models for Process Implementation and Change																			
1.4 Practical Considerations																			
2. Process Definition																			
2.1 Life Cycle Models																			
2.2 Software Life Cycle Processes	*	*		*			*							*	*	*	*	*	
2.3 Notations for Process Definitions					*														
2.4 Process Adaptation																		*	
2.5 Automation																			
3. Process Assessment																			
3.1 Process Assessment Models	*				*														
3.2 Process Assessment Methods					*														
4. Measurement																			
4.1 Process Measurement							*				*								
4.2 Software Products Measurement			*				*	*	*										*
4.3 Quality of Measurement Results											*								
4.4 Software Information Models																			
Model building																			
Model Implementation																			
4.5 Process Measurement Techniques												*						*	

RECOMMENDED REFERENCES

[Abr96] A. Abran and P.N. Robillard, "Function Points Analysis: An Empirical Study of its Measurement Processes," *IEEE Transactions on Software Engineering*, vol. 22, 1996, pp. 895-909.

[Bas92] V. Basili et al., "The Software Engineering Laboratory — An Operational Software Experience Factory," presented at the International Conference on Software Engineering, 1992.

[Bec99] K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

[Boe03] B. Boehm and R. Turner, "Using Risk to Balance Agile and Plan-Driven Methods," *Computer*, June 2003, pp. 57-66.

[Com97] E. Comer, "Alternative Software Life Cycle Models," presented at International Conference on Software Engineering, 1997.

[ElE99] K. El-Emam and N. Madhavji, *Elements of Software Process Assessment and Improvement*, IEEE Computer Society Press, 1999.

[Fen98] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, second ed., International Thomson Computer Press, 1998.

[Fin94] A. Finkelstein, J. Kramer, and B. Nuseibeh, "Software Process Modeling and Technology," *Research Studies Press Ltd.*, 1994.

[Fow90] P. Fowler and S. Rifkin, *Software Engineering Process Group Guide*, Software Engineering Institute, Technical Report CMU/SEI-90-TR-24, 1990, available at http://www.sei.cmu.edu/pub/documents/90.reports/pdf/tr24.90.pdf.

[Gol99] D. Goldenson et al., "Empirical Studies of Software Process Assessment Methods," presented at Elements of Software Process Assessment and Improvement, 1999.

[IEEE1074-97] IEEE Std 1074-1997, *IEEE Standard for Developing Software Life Cycle Processes*, IEEE, 1997.

[IEEE12207.0-96] IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.

[VIM93] ISO VIM, *International Vocabulary of Basic and General Terms in Metrology*, ISO, 1993.

[ISO9126-01] ISO/IEC 9126-1:2001, *Software Engineering - Product Quality-Part 1: Quality Model*, ISO and IEC, 2001.

[ISO15504-98] ISO/IEC TR 15504:1998, *Information Technology - Software Process Assessment (parts 1-9)*, ISO and IEC, 1998.

[ISO15939-02] ISO/IEC 15939:2002, *Software Engineering — Software Measurement Process*, ISO and IEC, 2002.

[Joh99] D. Johnson and J. Brodman, "Tailoring the CMM for Small Businesses, Small Organizations, and Small Projects," presented at Elements of Software Process Assessment and Improvement, 1999.

[McF96] B. McFeeley, *IDEAL: A User's Guide for Software Process Improvement*, Software Engineering Institute CMU/SEI-96-HB-001, 1996, available at http://www.sei.cmu.edu/pub/documents/96.reports/pdf/hb001.96.pdf.

[Moi98] D. Moitra, "Managing Change for Software Process Improvement Initiatives: A Practical Experience-Based Approach," *Software Process — Improvement and Practice*, vol. 4, iss. 4, 1998, pp. 199-207.

[Mus99] J. Musa, *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*, McGraw-Hill, 1999.

[OMG02] Object Management Group, "Software Process Engineering Metamodel Specification," 2002, available at http://www.omg.org/docs/formal/02-11-14.pdf.

[Pfl01] S.L. Pfleeger, *Software Engineering: Theory and Practice*, second ed., Prentice Hall, 2001.

[Pre04] R.S. Pressman, *Software Engineering: A Practitioner's Approach*, sixth ed., McGraw-Hill, 2004.

[San98] M. Sanders, "The SPIRE Handbook: Better, Faster, Cheaper Software Development in Small Organisations," *European Commission*, 1998.

[SEI01] Software Engineering Institute, "Capability Maturity Model Integration, v1.1," 2001, available at http://www.sei.cmu.edu/cmmi/cmmi.html.

[SEL96] Software Engineering Laboratory, *Software Process Improvement Guidebook*, NASA/GSFC, Technical Report SEL-95-102, April 1996, available at http://sel.gsfc.nasa.gov/website/documents/online-doc/95-102.pdf.

[Som05] I. Sommerville, *Software Engineering*, seventh ed., Addison-Wesley, 2005.

[Sti99] H. Stienen, "Software Process Assessment and Improvement: 5 Years of Experiences with Bootstrap," *Elements of Software Process Assessment and Improvement*, K. El-Emam and N. Madhavji, eds., IEEE Computer Society Press, 1999.

Appendix A. List of Further Readings

(Agr99) W. Agresti, "The Role of Design and Analysis in Process Improvement," presented at Elements of Software Process Assessment and Improvement, 1999.

(Ale91) L. Alexander and A. Davis, "Criteria for Selecting Software Process Models," presented at COMPSAC '91, 1991.

(Ban95) S. Bandinelli et al., "Modeling and Improving an Industrial Software Process," *IEEE Transactions on Software Engineering*, vol. 21, iss. 5, 1995, pp. 440-454.

(Bar95) N. Barghouti et al., "Two Case Studies in Modeling Real, Corporate Processes," *Software Process — Improvement and Practice*, Pilot Issue, 1995, pp. 17-32.

(Boe03a) B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley, 2003.

(Bur99) I. Burnstein et al., "A Testing Maturity Model for Software Test Process Assessment and Improvement," *Software Quality Professional*, vol. 1, iss. 4, 1999, pp. 8-21.

(Chi92) R. Chillarege et al., "Orthogonal Defect Classification - A Concept for In-Process Measurement," *IEEE Transactions on Software Engineering*, vol. 18, iss. 11, 1992, pp. 943-956.

(Chi96) R. Chillarege, "Orthogonal Defect Classification," *Handbook of Software Reliability Engineering*, M. Lyu, ed., IEEE Computer Society Press, 1996.

(Col93) J. Collofello and B. Gosalia, "An Application of Causal Analysis to the Software Production Process," *Software Practice and Experience*, vol. 23, iss. 10, 1993, pp. 1095-1105.

(Cur02) B. Curtis, W. Hefley, and S. Miller, *The People Capability Maturity Model: Guidelines for Improving the Workforce*, Addison-Wesley, 2002.

(Dav88) A. Davis, E. Bersoff, and E. Comer, "A Strategy for Comparing Alternative Software Development Life Cycle Models," *IEEE Transactions on Software Engineering*, vol. 14, iss. 10, 1988, pp. 1453-1461.

(Dun96) D. Dunnaway and S. Masters, "CMM-Based Appraisal for Internal Process Improvement (CBA IPI): Method Description," Software Engineering Institute CMU/SEI-96-TR-007, 1996, available at http://www.sei.cmu.edu/pub/documents/96.reports/pdf/tr007. 96.pdf.

(EIA/IS731-99) EIA, "EIA/IS 731 Systems Engineering Capability Model," 1999, available at http://www.geia.org/eoc/G47/index.html.

(ElE-97) K. El-Emam, D. Holtje, and N. Madhavji, "Causal Analysis of the Requirements Change Process for a Large System," presented at Proceedings of the International Conference on Software Maintenance, 1997.

(ElE-99a) K. El-Emam, B. Smith, and P. Fusaro, "Success Factors and Barriers in Software Process Improvement: An Empirical Study," *Better Software Practice for Business Benefit: Principles and Experiences*, R. Messnarz and C. Tully, eds., IEEE Computer Society Press, 1999.

(ElE-00a) K. El-Emam and A. Birk, "Validating the ISO/IEC 15504 Measures of Software Development Process Capability," *Journal of Systems and Software*, vol. 51, iss. 2, 2000, pp. 119-149.

(ElE-00b) K. El-Emam and A. Birk, "Validating the ISO/IEC 15504 Measures of Software Requirements Analysis Process Capability," *IEEE Transactions on Software Engineering*, vol. 26, iss. 6, June 2000, pp. 541-566.

(Fay97) M. Fayad and M. Laitinen, "Process Assessment: Considered Wasteful," *Communications of the ACM*, vol. 40, iss. 11, November 1997.

(Flo99) W. Florac and A. Carleton, *Measuring the Software Process: Statistical Process Control for Software Process Improvement*, Addison-Wesley, 1999.

(Gar96) P. Garg and M. Jazayeri, "Process-Centered Software Engineering Environments: A Grand Tour," *Software Process*, A. Fuggetta and A. Wolf, eds., John Wiley & Sons, 1996.

(Gra97) R. Grady, *Successful Software Process Improvement*, Prentice Hall, 1997.

(Gra88) E. Gray and W. Smith, "On the Limitations of Software Process Assessment and the Recognition of a Required Re-Orientation for Global Process Improvement," *Software Quality Journal*, vol. 7, 1998, pp. 21-34.

(Har98) D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The Statemate Approach*, McGraw-Hill, 1998.

(Her98) J. Herbsleb, "Hard Problems and Hard Science: On the Practical Limits of Experimentation," *IEEE TCSE Software Process Newsletter*, vol. 11, 1998, pp. 18-21, available at http://www.seg.iit.nrc.ca/SPN.

(Hum95) W. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley, 1995.

(Hum99) W. Humphrey, *An Introduction to the Team Software Process*, Addison-Wesley, 1999.

(Hut94) D. Hutton, *The Change Agent's Handbook: A Survival Guide for Quality Improvement Champions*, Irwin, 1994.

(Kan02) S.H. Kan, *Metrics and Models in Software Quality Engineering*, second ed., Addison-Wesley, 2002.

(Kel98) M. Kellner et al., "Process Guides: Effective Guidance for Process Participants," presented at the 5th International Conference on the Software Process, 1998.

(Kit98) B. Kitchenham, "Selecting Projects for Technology Evaluation," *IEEE TCSE Software Process Newsletter*, iss. 11, 1998, pp. 3-6, available at http://www.seg.iit.nrc.ca/SPN.

(Kra99) H. Krasner, "The Payoff for Software Process Improvement: What It Is and How to Get It," presented at

Elements of Software Process Assessment and Improvement, 1999.

(Kri99) M.S. Krishnan and M. Kellner, "Measuring Process Consistency: Implications for Reducing Software Defects," *IEEE Transactions on Software Engineering*, vol. 25, iss. 6, November/December 1999, pp. 800-815.

(Lyu96) M.R. Lyu, *Handbook of Software Reliability Engineering*, Mc-Graw-Hill/IEEE, 1996.

(Mad94) N. Madhavji et al., "Elicit: A Method for Eliciting Process Models," presented at Proceedings of the Third International Conference on the Software Process, 1994.

(Mas95) S. Masters and C. Bothwell, "CMM Appraisal Framework - Version 1.0," Software Engineering Institute CMU/SEI-TR-95-001, 1995, available at http://www.sei.cmu.edu/pub/documents/95.reports/pdf/tr001.95.pdf.

(McG94) F. McGarry et al., "Software Process Improvement in the NASA Software Engineering Laboratory," Software Engineering Institute CMU/SEI-94-TR-22, 1994, available at http://www.sei.cmu.edu/pub/documents/94.reports/pdf/ tr22.94.pdf.

(McG01) J. McGarry et al., *Practical Software Measurement: Objective Information for Decision Makers*, Addison-Wesley, 2001.

(Mcg93) C. McGowan and S. Bohner, "Model Based Process Assessments," presented at International Conference on Software Engineering, 1993.

(Nak91) T. Nakajo and H. Kume, "A Case History Analysis of Software Error Cause-Effect Relationship," *IEEE Transactions on Software Engineering*, vol. 17, iss. 8, 1991.

(Pau94) M. Paulk and M. Konrad, "Measuring Process Capability Versus Organizational Process Maturity," presented at 4th International Conference on Software Quality, 1994.

(Pfl99) S.L. Pfleeger, "Understanding and Improving Technology Transfer in Software Engineering," *Journal of Systems and Software*, vol. 47, 1999, pp. 111-124.

(Pfl01) S.L. Pfleeger, *Software Engineering: Theory and Practice*, second ed., Prentice Hall, 2001.

(Rad85) R. Radice et al., "A Programming Process Architecture," *IBM Systems Journal*, vol. 24, iss. 2, 1985, pp. 79-90.

(Rag89) S. Raghavan and D. Chand, "Diffusing Software-Engineering Methods," *IEEE Software*, July 1989, pp. 81-90.

(Rog83) E. Rogers, *Diffusion of Innovations*, Free Press, 1983.

(Sch99) E. Schein, *Process Consultation Revisited: Building the Helping Relationship*, Addison-Wesley, 1999.

(SEI95) Software Engineering Institute, *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, 1995.

(SEL96) Software Engineering Laboratory, *Software Process Improvement Guidebook*, Software Engineering Laboratory, NASA/GSFC, Technical Report SEL-95-102, April 1996, available at http://sel.gsfc.nasa.gov/website/documents/online-doc/95-102.pdf

(SPC92) Software Productivity Consortium, *Process Definition and Modeling Guidebook*, Software Productivity Consortium, SPC-92041-CMC, 1992.

(Som97) I. Sommerville and P. Sawyer, *Requirements Engineering: A Good Practice Guide*, John Wiley & Sons, 1997.

(Vot93) L. Votta, "Does Every Inspection Need a Meeting?" *ACM Software Engineering Notes*, vol. 18, iss. 5, 1993, pp. 107-114.

(Wei93) G.M. Weinberg, "Quality Software Management," *First-Order Measurement (Ch. 8, Measuring Cost and Value)*, vol. 2, 1993.

(Yu94) E. Yu and J. Mylopolous, "Understanding 'Why' in Software Process Modeling, Analysis, and Design," presented at 16th International Conference on Software Engineering, 1994

(Zah98) S. Zahran, *Software Process Improvement: Practical Guidelines for Business Success*, Addison-Wesley, 1998.

(Zel98) M. V. Zelkowitz and D. R. Wallace, "Experimental Models for Validating Technology," *Computer*, vol. 31, iss. 5, 1998, pp. 23-31.

Appendix B. List of Standards

(IEEE1044-93) IEEE Std 1044-1993 (R2002), *IEEE Standard for the Classification of Software Anomalies*, IEEE, 1993.

(IEEE1061-98) IEEE Std 1061-1998, *IEEE Standard for a Software Quality Metrics Methodology*, IEEE, 1998.

(IEEE1074-97) IEEE Std 1074-1997, *IEEE Standard for Developing Software Life Cycle Processes*, IEEE, 1997.

(IEEE1219-98) IEEE Std 1219-1998, *IEEE Standard for Software Maintenance*, IEEE, 1998.

(IEEE1220-98) IEEE Std 1220-1998, *IEEE Standard for the Application and Management of the Systems Engineering Process*, IEEE, 1998.

(IEEE1517-99) IEEE Std 1517-1999, *IEEE Standard for Information Technology-Software Life Cycle Processes-Reuse Processes*, IEEE, 1999.

(IEEE1540-01) IEEE Std 1540-2001//ISO/IEC16085:2003, *IEEE Standard for Software Life Cycle Processes-Risk Management*, IEEE, 2001.

(IEEE12207.0-96) IEEE/EIA 12207.0-1996//ISO/ IEC12207:1995, *Industry Implementation of Int. Std ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.

(IEEE12207.1-96) IEEE/EIA 12207.1-1996, *Industry Implementation of Int. Std ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes - Life Cycle Data*, IEEE, 1996.

(IEEE12207.2-97) IEEE/EIA 12207.2-1997, *Industry Implementation of Int. Std ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes - Implementation Considerations*, IEEE, 1997.

(IEEE14143.1-00) IEEE Std 14143.1-2000//ISO/ IEC14143-1:1998, *Information Technology-Software Measurement-Functional Size Measurement-Part 1: Definitions of Concepts*, IEEE, 2000.

(ISO9001-00) ISO 9001:2000, *Quality Management Systems-Requirements*, ISO, 1994.

(ISO9126-01) ISO/IEC 9126-1:2001, *Software Engineering-Product Quality-Part 1: Quality Model*, ISO and IEC, 2001.

(ISO14674-99) ISO/IEC 14674:1999, *Information Technology - Software Maintenance*, ISO and IEC, 1999.

(ISO15288-02) ISO/IEC 15288:2002, *Systems Engineering-System Life Cycle Process*, ISO and IEC, 2002.

(ISO15504-98) ISO/IEC TR 15504:1998, *Information Technology - Software Process Assessment (parts 1-9)*, ISO and IEC, 1998.

(ISO15939-02) ISO/IEC 15939:2002, *Software Engineering-Software Measurement Process*, ISO and IEC, 2002.

(ISO19761-03) ISO/IEC 19761:2003, *Software Engineering-Cosmic FPP-A Functional Size Measurement Method*, ISO and IEC, 2003.

(ISO20926-03) ISO/IEC 20926:2003, *Software Engineering-IFPUG 4.1 Unadjusted Functional Size Measurement Method-Counting Practices Manual*, ISO and IEC, 2003.

(ISO20968-02) ISO/IEC 20968:2002, *Software Engineering-MK II Function Point Analysis - Counting Practices Manual*, ISO and IEC, 2002.

(ISO90003-04) ISO/IEC 90003:2004, *Software and Systems Engineering - Guidelines for the Application of ISO9001:2000 to Computer Software*, ISO and IEC, 2004.

(VIM93) ISO VIM, *International Vocabulary of Basic and General Terms in Metrology*, ISO, 1993.

CHAPTER 10

SOFTWARE ENGINEERING TOOLS AND METHODS

ACRONYM

CASE	Computer Assisted Software Engineering

INTRODUCTION

Software development tools are the computer-based tools that are intended to assist the software life cycle processes. Tools allow repetitive, well-defined actions to be automated, reducing the cognitive load on the software engineer who is then free to concentrate on the creative aspects of the process. Tools are often designed to support particular software engineering methods, reducing any administrative load associated with applying the method manually. Like software engineering methods, they are intended to make software engineering more systematic, and they vary in scope from supporting individual tasks to encompassing the complete life cycle.

Software engineering methods impose structure on the software engineering activity with the goal of making the activity systematic and ultimately more likely to be successful. Methods usually provide a notation and vocabulary, procedures for performing identifiable tasks, and guidelines for checking both the process and the product. They vary widely in scope, from a single life cycle phase to the complete life cycle. The emphasis in this KA is on software engineering methods encompassing multiple life cycle phases, since phase-specific methods are covered by other KAs.

While there are detailed manuals on specific tools and numerous research papers on innovative tools, generic technical writings on software engineering tools are relatively scarce. One difficulty is the high rate of change in software tools in general. Specific details alter regularly, making it difficult to provide concrete, up-to-date examples.

The Software Engineering Tools and Methods KA covers the complete life cycle processes, and is therefore related to every KA in the Guide.

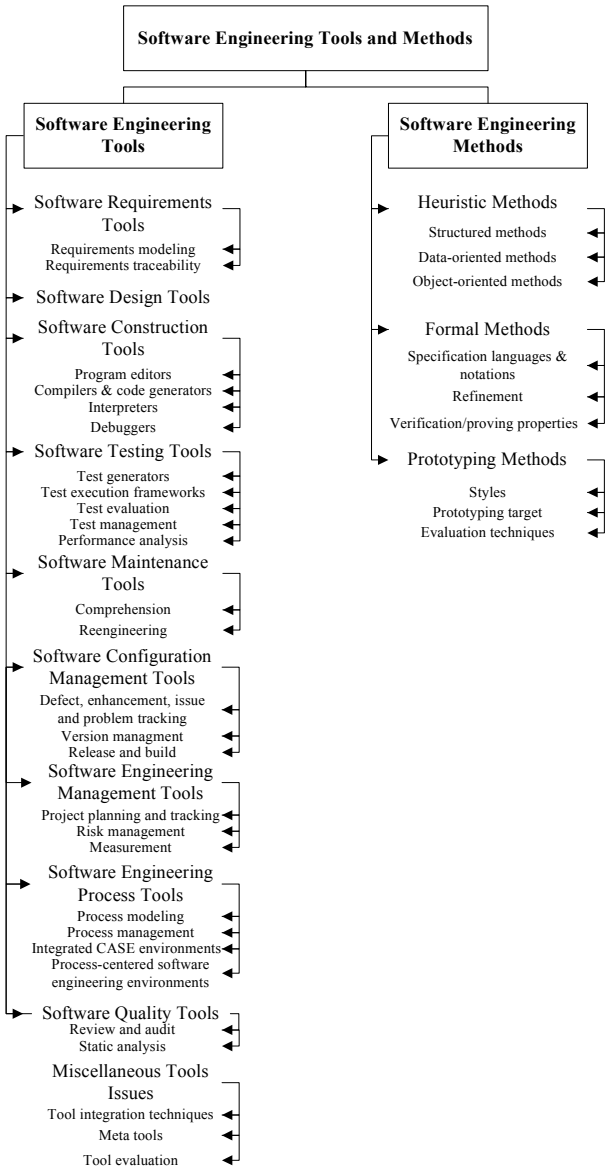


Figure 1 Breakdown of topics in the Software Engineering Tools and Methods KA

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING TOOLS AND METHODS

1. Software Engineering Tools

The first five topics in the *Software Engineering Tools* subarea correspond to the first five KAs of the Guide (Software Requirements, Software Design, Software Construction, Software Testing, and Software Maintenance). The next four topics correspond to the remaining KAs (Software Configuration Management, Software Engineering Management, Software Engineering Process, and Software Quality). An additional topic is provided, Miscellaneous, addressing areas such as tool integration techniques which are potentially applicable to all classes of tools.

1.1. Software Requirements Tools

[Dor97, Dor02]

Tools for dealing with software requirements have been classified into two categories: modeling and traceability tools.

- Requirements modeling tools. These tools are used for eliciting, analyzing, specifying, and validating software requirements
- Requirement traceability tools. [Dor02] These tools are becoming increasingly important as the complexity of software grows. Since they are also relevant in other life cycle processes, they are presented separately from the requirements modeling tools.

1.2. Software Design Tools

[Dor02]

This topic covers tools for creating and checking software designs. There are a variety of such tools, with much of this variety being a consequence of the diversity of software design notations and methods. In spite of this variety, no compelling divisions for this topic have been found.

1.3. Software Construction Tools

[Dor02, Rei96]

This topic covers software construction tools. These tools are used to produce and translate program representation (for instance, source code) which is sufficiently detailed and explicit to enable machine execution.

- Program editors. These tools are used for the creation and modification of programs, and possibly the documents associated with them. They can be general-purpose text or document editors, or they can be specialized for a target language.
- Compilers and code generators. Traditionally, compilers have been noninteractive translators of source code, but there has been a trend to integrate compilers and program editors to provide integrated programming environments. This topic also covers preprocessors, linker/loaders, and code generators.
- Interpreters. These tools provide software execution through emulation. They can support software construction activities by providing a more controllable and observable environment for program execution.
- Debuggers. These tools are considered a separate category since they support the software construction process, but they are different from program editors and compilers.

1.4. Software Testing Tools

[Dor02, Pfl01, Rei96]

- Test generators. These tools assist in the development of test cases.
- Test execution frameworks. These tools enable the execution of test cases in a controlled environment where the behavior of the object under test is observed.
- Test evaluation tools. These tools support the assessment of the results of test execution, helping to determine whether or not the observed behavior conforms to the expected behavior.
- Test management tools. These tools provide support for all aspects of the software testing process.
- Performance analysis tools. [Rei96] These tools are used for measuring and analyzing software performance, which is a specialized form of testing where the goal is to assess performance behavior rather than functional behavior (correctness).

1.5. Software Maintenance Tools

[Dor02, Pfl01]

This topic encompasses tools which are particularly important in software maintenance where existing software is being modified. Two categories are identified: comprehension tools and reengineering tools.

- Comprehension tools. [Re196] These tools assist in the human comprehension of programs. Examples include visualization tools such as animators and program slicers.
- Reengineering tools. In the Software Maintenance KA, reengineering is defined as the examination and alteration of the subject software to reconstitute it in a new form, and includes the subsequent implementation of the new form. Reengineering tools support that activity.

Reverse engineering tools assist the process by working backwards from an existing product to create artifacts such as specification and design descriptions, which then can be transformed to generate a new product from an old one.

1.6. Software Configuration Management Tools

[Dor02, Rei96, Som05]

Tools for configuration management have been divided into three categories: tracking, version management, and release tools.

- Defect, enhancement, issue, and problem-tracking tools. These tools are used in connection with the problem-tracking issues associated with a particular software product.
- Version management tools. These tools are involved in the management of multiple versions of a product.
- Release and build tools. These tools are used to manage the tasks of software release and build. The category includes installation tools which have become widely used for configuring the installation of software products.

Additional information is given in the Software Configuration Management KA, topic 1.3 *Planning for SCM*.

1.7. Software Engineering Management Tools

[Dor02]

Software engineering management tools are subdivided into three categories: project planning and tracking, risk management, and measurement.

- Project planning and tracking tools. These tools are used in software project effort measurement and cost estimation, as well as project scheduling.
- Risk management tools. These tools are used in identifying, estimating, and monitoring risks.
- Measurement tools. The measurement tools assist in performing the activities related to the software measurement program.

1.8. Software Engineering Process Tools

[Dor02, Som05]

Software engineering process tools are divided into modeling tools, management tools, and software development environments.

- Process modeling tools. [Pfl01] These tools are used to model and investigate software engineering processes.
- Process management tools. These tools provide support for software engineering management.
- Integrated CASE environments. [Rei96, Som05] (ECMA55-93, ECMA69-94, IEEE1209-92, IEEE1348-95, Mul96) Integrated computer-aided software engineering tools or environments covering multiple phases of the software engineering life cycle belong in this subtopic. Such tools perform multiple functions and hence potentially interact with the software life cycle process being executed.
- Process-centered software engineering environments. [Rei96] (Gar96) These environments explicitly incorporate information on the software life cycle processes and guide and monitor the user according to the defined process.

1.9. Software Quality Tools

[Dor02]

Quality tools are divided into two categories: inspection and analysis tools.

- Review and audit tools. These tools are used to support reviews and audits.
- Static analysis tools. [Cla96, Pfl01, Rei96] These tools are used to analyze software artifacts, such as syntactic and semantic analyzers, as well as data, control flow, and dependency analyzers. Such tools are intended for checking software artifacts for conformance or for verifying desired properties.

1.10. Miscellaneous Tool Issues

[Dor02]

This topic covers issues applicable to all classes of tools. Three categories have been identified: tool integration techniques, meta-tools, and tool evaluation.

- Tool integration techniques. [Pfl01, Rei96, Som01] (Bro94) Tool integration is important for making individual tools cooperate. This category potentially overlaps with the integrated CASE environments category where integration techniques are applied; however, it was felt that it is sufficiently distinct to merit a category of its own. Typical kinds of tool integration are platform, presentation, process, data, and control.
- Meta-tools. Meta-tools generate other tools; compiler-compilers are the classic example.
- Tool evaluation. [Pfl01] (IEEE1209-92, IEEE1348-95, Mos92, Val97) Because of the continuous evolution of software engineering tools, tool evaluation is an essential topic.

2. Software Engineering Methods

The *Software Engineering Methods* subarea is divided into three topics: *heuristic methods* dealing with informal approaches, *formal methods* dealing with mathematically based approaches, and *prototyping methods* dealing with software engineering approaches based on various forms of prototyping. These three topics are not disjoint; rather they represent distinct concerns. For example, an object-oriented method may incorporate formal techniques and rely on prototyping for verification and validation. Like software engineering tools, methodologies continuously evolve. Consequently, the KA description avoids as far as possible naming particular methodologies.

2.1. *Heuristic methods*

[Was96]

This topic contains four categories: *structured, data-oriented, object-oriented, and domain-specific*. The domain-specific category includes specialized methods for developing systems which involve real-time, safety, or security aspects.

- Structured methods. [Dor02, Pfl01, Pre04, Som05] The system is built from a functional viewpoint, starting with a high-level view and progressively refining this into a more detailed design.
- Data-oriented methods. [Dor02, Pre04] Here, the starting points are the data structures that a program manipulates rather than the function it performs.
- Object-oriented methods. [Dor02, Pfl01, Pre04, Som05] The system is viewed as a collection of objects rather than functions.

2.2. *Formal Methods*

[Dor02, Pre04, Som05]

This subsection deals with mathematically based software engineering methods, and is subdivided according to the various aspects of formal methods.

- Specification languages and notations. [Cla96, Pfl01, Pre01] This topic concerns the specification notation or language used. Specification languages can be classified as model-oriented, property-oriented, or behavior-oriented.
- Refinement. [Pre04] This topic deals with how the method refines (or transforms) the specification into a form which is closer to the desired final form of an executable program.
- Verification/proving properties. [Cla96, Pfl01, Som05] This topic covers the verification properties that are specific to the formal approach, including both theorem proving and model checking.

2.3. *Prototyping Methods*

[Pre04, Som05, Was96]

This subsection covers methods involving software prototyping and is subdivided into prototyping styles, targets, and evaluation techniques.

- Prototyping styles. [Dor02, Pfl01, Pre04] (Pom96) The prototyping styles topic identifies the various approaches: throwaway, evolutionary, and executable specification.
- Prototyping target. [Dor97] (Pom96) Examples of the targets of a prototyping method may be requirements, architectural design, or the user interface.
- Prototyping evaluation techniques. This topic covers the ways in which the results of a prototype exercise are used.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	[Cla96]	[Dor02] {Dor97}	[Pfl01] {PFL98}	[Pre04]	[Rei96]	[Som05]	[Was96]
1. Software Tools							
1.1 Software Requirements Tools		{c4s1} ,v2c8s4					
Requirement modeling tools							
Requirements traceability tools		v1c4s2					
1.2 Software Design Tools		v2c8s4					
1.3 Software Construction Tools		v2c8s4			c112s2		
Program editors							
Compilers and code generators							
Interpreters							
Debuggers							
1.4 Software Testing Tools		v2c8s4	C8s7,c9s7		c112s3		
Test generators							
Test execution frameworks							
Test evaluation tools							
Test management tools							
Performance analysis tools					c112s5		
1.5 Software Maintenance Tools		v2c8s4	c11s5				
Comprehension tools					c112s5		
Reengineering tools							
1.6 Software Configuration Management Tools		v2c8s4	c11s5		c112s3	c29	
Defect, enhancement, issue, and problem-tracking tools							
Version management tools							
Release and build tools							

	[Cla96]	[Dor02] {Dor97}	[Pfl01] {PFL98}	[Pre04]	[Rei96]	[Som05]	[Was96]
1.7 Software Engineering Management Tools		v2c8s4					
Project planning and tracking tools							
Risk management tools							
Measurement tools							
1.8 Software Engineering Process Tools		v2c8s4					
Process modeling tools			c2s3, 2s4				
Process management tools							
Integrated CASE environments					c112s3, c112s4	c3	
Process-centered software engineering environments					c112s5		
1.9 Software Quality Tools		v2c8s4					
Review and audit tools							
Static analysis tools	*		C8s7		c112s5		
1.10 Miscellaneous Tool Issues		v2c8s4					
Tool integration techniques			c1s8		c112s4		*
Meta-tools							
Tool evaluation			C9s10				
2. Development Methods							
2.1 Heuristic Methods							*
Structured methods		v1c5s1, v1c6s3	c4s5	c7-c9		c15	
Data-oriented methods		v1c5s1, v1c6s3		c7-c9			
Object-oriented methods		v1c6s2, v1c6s3	c4s4, c6, c8s5	c7-c9		c12	
2.2 *Formal Methods*		v1c6s5		c28		c9	
Specification languages and notation	*		c4s5				
Refinement							
Verification/proving properties	*		c5s7, c8s3				
2.3 *Prototyping Methods*						c8	*
Styles		v1c4s4	c4s6, c5s6				
Prototyping target		v1c4s4					
Evaluation techniques							

RECOMMENDED REFERENCES FOR SOFTWARE ENGINEERING TOOLS AND METHODS

[Cla96] E.M. Clarke et al., "Formal Methods: State of the Art and Future Directions," *ACM Computer Surveys*, vol. 28, iss. 4, 1996, pp. 626-643.

[Dor97] M. Christensen, M. Dorfman and R.H. Thayer, eds., *Software Engineering*, IEEE Computer Society Press, 1997.

[Dor02] M. Christensen, M. Dorfman and R.H. Thayer, eds., *Software Engineering*, Vol. 1 & Vol. 2, IEEE Computer Society Press, 2002.

[Pfl01] S.L. Pfleeger, *Software Engineering: Theory and Practice*, second ed., Prentice Hall, 2001.

[Pre04] R.S. Pressman, *Software Engineering: A Practitioner's Approach*, sixth ed., McGraw-Hill, 2004.

[Rei96] S.P. Reiss, *Software Tools and Environments in The Computer Science and Engineering Handbook*, CRC Press, 1996.

[Som05] I. Sommerville, *Software Engineering*, seventh ed., Addison-Wesley, 2005.

[Was96] A.I. Wasserman, "Toward a Discipline of Software Engineering," *IEEE Software*, vol. 13, iss. 6, November 1996, pp. 23-31.

APPENDIX A. LIST OF FURTHER READINGS

(Ber93) E.V. Berard, *Essays on Object-Oriented Software Engineering*, Prentice Hall, 1993.

(Bis92) W. Bischofberger and G. Pomberger, *Prototyping-Oriented Software Development: Concepts and Tools*, Springer-Verlag, 1992.

(Bro94) A.W. Brown et al., *Principles of CASE Tool Integration*, Oxford University Press, 1994.

(Car95) D.J. Carney and A.W. Brown, "On the Necessary Conditions for the Composition of Integrated Software Engineering Environments," presented at Advances in Computers, 1995.

(Col94) D. Coleman et al., *Object-Oriented Development: The Fusion Method*, Prentice Hall, 1994.

(Cra95) D. Craigen, S. Gerhart, and T. Ralston, "Formal Methods Reality Check: Industrial Usage," *IEEE Transactions on Software Engineering*, vol. 21, iss. 2, February 1995, pp. 90-98.

(Fin00) A. Finkelstein, ed., *The Future of Software Engineering*, ACM, 2000.

(Gar96) P.K. Garg and M. Jazayeri, *Process-Centered Software Engineering Environments*, IEEE Computer Society Press, 1996.

(Har00) W. Harrison, H. Ossher, and P. Tarr, "Software Engineering Tools and Environments: A Roadmap," 2000.

(Jar98) S. Jarzabek and R. Huang, "The Case for User-Centered CASE Tools," *Communications of the ACM*, vol. 41, iss. 8, August 1998, pp. 93-99.

(Kit95) B. Kitchenham, L. Pickard, and S.L. Pfleeger, "Case Studies for Method and Tool Evaluation," *IEEE Software*, vol. 12, iss. 4, July 1995, pp. 52-62.

(Lam00) A. v. Lamsweerde, "Formal Specification: A Roadmap," *The Future of Software Engineering*, A. Finkelstein, ed., ACM, 2000, pp. 149-159.

(Mey97) B. Meyer, *Object-Oriented Software Construction*, second ed., Prentice Hall, 1997.

(Moo98) J.W. Moore, *Software Engineering Standards, A User's Roadmap*, IEEE Computer Society Press, 1998.

(Mos92) V. Mosley, "How to Assess Tools Efficiently and Quantitatively," *IEEE Software*, vol. 9, iss. 3, May 1992, pp. 29-32.

(Mül96) H.A. Muller, R.J. Norman, and J. Slonim, eds., "Computer Aided Software Engineering," special issue of *Automated Software Engineering*, vol. 3, iss. 3/4, Kluwer, 1996.

(Mül00) H. Müller et al., "Reverse Engineering: A Roadmap," *The Future of Software Engineering*, A. Finkelstein, ed., ACM, 2000, pp. 49-60.

(Pom96) G. Pomberger and G. Blaschek, *Object-Orientation and Prototyping in Software Engineering*: Prentice Hall, 1996.

(Pos96) R.M. Poston, *Automating Specification-based Software Testing*, IEEE Press, 1996.

(Ric92) C. Rich and R.C. Waters, "Knowledge Intensive Software Engineering Tools," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, iss. 5, October 1992, pp. 424-430.

(Son92) X. Song and L.J. Osterweil, "Towards Objective, Systematic Design-Method Comparisons," *IEEE Software*, vol. 9, iss. 3, May 1992, pp. 43-53.

(Tuc96) A.B. Tucker, *The Computer Science and Engineering Handbook*, CRC Press, 1996.

(Val97) L.A. Valaer and R.C.B. II, "Choosing a User Interface Development Tool," *IEEE Software*, vol. 14, iss. 4, 1997, pp. 29-39.

(Vin90) W.G. Vincenti, *What Engineers Know and How They Know It — Analytical Studies from Aeronautical History*, John Hopkins University Press, 1990.

(Wie98) R. Wieringa, "A Survey of Structured and Object-Oriented Software Specification Methods and Techniques," *ACM Computing Surveys*, vol. 30, iss. 4, 1998, pp. 459-527.

APPENDIX B. LIST OF STANDARDS

(ECMA55-93) ECMA, *TR/55 Reference Model for Frameworks of Software Engineering Environments*, third ed., 1993.

(ECMA69-94) ECMA, *TR/69 Reference Model for Project Support Environments*, 1994.

(IEEE1175.1-02) IEEE Std 1175.1-2002, *IEEE Guide for CASE Tool Interconnections—Classification and Description*, IEEE Press, 2002.

(IEEE1209-92) IEEE Std 1209-1992, *Recommended Practice for the Evaluation and Selection of CASE Tools, (ISO/IEC 14102, 1995)*, IEEE Press, 1992.

(IEEE1348-95) IEEE Std 1348-1995, *Recommended Practice for the Adoption of CASE Tools, (ISO/IEC 14471)*, IEEE Press, 1995.

(IEEE12207.0-96) IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology—Software Life Cycle Processes*, IEEE Press, 1996.

CHAPTER 11

SOFTWARE QUALITY

ACRONYMS

CMMI	Capability Maturity Model Integrated
COTS	Commercial Off-the-Shelf Software
PDCA	Plan, Do, Check, Act
SQA	Software Quality Assurance
SQM	Software Quality Management
TQM	Total Quality Management
V&V	Verification and Validation

INTRODUCTION

What is software quality, and why is it so important that it be pervasive in the SWEBOK Guide? Over the years, authors and organizations have defined the term "quality" differently. To Phil Crosby (Cro79), it was "conformance to user requirements." Watts Humphrey (Hum89) refers to it as "achieving excellent levels of fitness for use," while IBM coined the phrase "market-driven quality," which is based on achieving total customer satisfaction. The Baldrige criteria for organizational quality (NIST03) use a similar phrase, "customer-driven quality," and include customer satisfaction as a major consideration. More recently, quality has been defined in (ISO9001-00) as "the degree to which a set of inherent *characteristics* fulfills *requirements.*"

This chapter deals with software quality considerations which transcend the life cycle processes. Software quality is a ubiquitous concern in software engineering, and so it is also considered in many of the KAs. In summary, the SWEBOK Guide describes a number of ways of achieving software quality. In particular, this KA will cover *static techniques*, those which do not require the execution of the software being evaluated, while *dynamic techniques* are covered in the Software Testing KA.

BREAKDOWN OF SOFTWARE QUALITY TOPICS

1. Software Quality Fundamentals

Agreement on quality requirements, as well as clear communication to the software engineer on what constitutes quality, require that the many aspects of quality be formally defined and discussed.

A software engineer should understand the underlying meanings of quality concepts and characteristics and their value to the software under development or to maintenance. The important concept is that the software requirements define the required quality characteristics of the software and influence the measurement methods and acceptance criteria for assessing these characteristics.

1.1. Software Engineering Culture and Ethics

Software engineers are expected to share a commitment to software quality as part of their culture. A healthy software engineering culture is described in [Wie96].

Ethics can play a significant role in software quality, the culture, and the attitudes of software engineers. The IEEE Computer Society and the ACM [IEEE99] have developed a code of ethics and professional practice based on eight principles to help software engineers reinforce attitudes related to quality and to the independence of their work.

1.2. Value and Costs of Quality

[Boe78; NIST03; Pre04; Wei93]

The notion of "quality" is not as simple as it may seem. For any engineered product, there are many desired qualities relevant to a particular perspective of the product, to be discussed and determined at the time that the product requirements are set down. Quality characteristics may be required or not, or may be required to a greater or lesser degree, and trade-offs may be made among them. [Pfl01]

The cost of quality can be differentiated into prevention cost, appraisal cost, internal failure cost, and external failure cost. [Hou99]

A motivation behind a software project is the desire to create software that has value, and this value may or may not be quantified as a cost. The customer will have some maximum cost in mind, in return for which it is expected that the basic purpose of the software will be fulfilled. The customer may also have some expectation as to the quality of the software. Sometimes customers may not have thought through the quality issues or their related costs. Is the characteristic merely decorative, or is it essential to the software? If the answer lies somewhere in between, as is almost always the case, it is a matter of making the customer a part of the decision process and fully aware of both costs and benefits. Ideally, most of these decisions will be made in the software requirements process (see the Software Requirements KA), but these issues may arise throughout the software life cycle. There is no definite rule as to how these decisions should be made, but the software engineer should be able to present quality alternatives and their costs. A discussion concerning cost and the value of quality requirements can be found in [Jon96:c5; Wei96:c11].

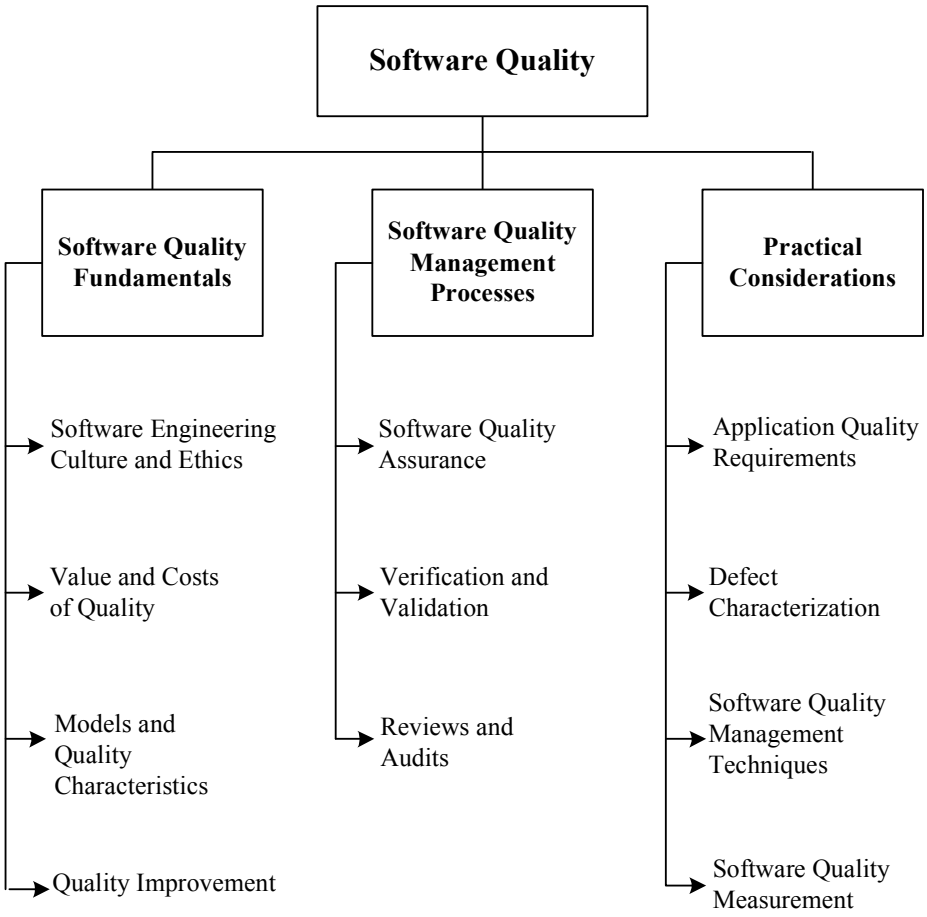


Figure 1 Breakdown of topics for the Software Quality KA

1.3. Models and Quality Characteristics

[Dac01; Kia95; Lap91; Lew92; Mus99; NIST; Pre01; Rak97; Sei02; Wal96]

Terminology for software quality characteristics differs from one taxonomy (or model of software quality) to another, each model perhaps having a different number of hierarchical levels and a different total number of characteristics. Various authors have produced models of software quality characteristics or attributes which can be useful for discussing, planning, and rating the quality of software products. [Boe78; McC77] ISO/IEC has defined three related models of software product quality (internal quality, external quality, and quality in use) (ISO9126-01) and a set of related parts (ISO14598-98).

1.3.1. Software engineering process quality

Software quality management and software engineering process quality have a direct bearing on the quality of the software product.

Models and criteria which evaluate the capabilities of software organizations are primarily project organization and management considerations, and, as such, are covered in the Software Engineering Management and Software Engineering Process KAs.

Of course, it is not possible to completely distinguish the quality of the process from the quality of the product.

Process quality, discussed in the Software Engineering Process KA of this *Guide*, influences the quality characteristics of software products, which in turn affect quality-in-use as perceived by the customer.

Two important quality standards are TickIT [Llo03] and one which has an impact on software quality, the ISO9001-00 standard**,** along with its guidelines for application to software [ISO90003-04].

Another industry standard on software quality is CMMI [SEI02], also discussed in the Software Engineering Process KA. CMMI intends to provide guidance for improving processes. Specific process areas related to quality

management are (a) process and product quality assurance, (b) process verification, and (c) process validation. CMMI classifies reviews and audits as methods of verification, and not as specific processes like (IEEE12207.0-96).

There was initially some debate over whether ISO9001 or CMMI should be used by software engineers to ensure quality. This debate is widely published, and, as a result, the position has been taken that the two are complementary and that having ISO9001 certification can help greatly in achieving the higher maturity levels of the CMMI. [Dac01]

1.3.2. Software product quality

The software engineer needs, first of all, to determine the real purpose of the software. In this regard, it is of prime importance to keep in mind that the customer's requirements come first and that they include quality requirements, not just functional requirements. Thus, the software engineer has a responsibility to elicit quality requirements which may not be explicit at the outset and to discuss their importance as well as the level of difficulty in attaining them. All processes associated with software quality (for example, building, checking, and improving quality) will be designed with these requirements in mind, and they carry additional costs.

Standard (ISO9126-01) defines, for two of its three models of quality, the related quality characteristics and sub-characteristics, and measures which are useful for assessing software product quality. (Sur03)

The meaning of the term "product" is extended to include any artifact which is the output of any process used to build the final software product. Examples of a product include, but are not limited to, an entire system requirements specification, a software requirements specification for a software component of a system, a design module, code, test documentation, or reports produced as a result of quality analysis tasks. While most treatments of quality are described in terms of the final software and system performance, sound engineering practice requires that intermediate products relevant to quality be evaluated throughout the software engineering process.

1.4. Quality Improvement

[NIST03; Pre04; Wei96]

The quality of software products can be improved through an iterative process of continuous improvement which requires management control, coordination, and feedback from many concurrent processes: (1) the software life cycle processes, (2) the process of error/defect detection, removal, and prevention, and (3) the quality improvement process. (Kin92)

The theory and concepts behind quality improvement, such as *building in quality* through the prevention and early detection of errors, continuous improvement, and customer focus, are pertinent to software engineering. These concepts are based on the work of experts in quality who have stated that the quality of a product is directly linked to the quality of the process used to create it. (Cro79, Dem86, Jur89)

Approaches such as the Total Quality Management (TQM) process of *Plan, Do, Check*, and *Act* (PDCA) are tools by which quality objectives can be met. Management sponsorship supports process and product evaluations and the resulting findings. Then, an improvement program is developed identifying detailed actions and improvement projects to be addressed in a feasible time frame. Management support implies that each improvement project has enough resources to achieve the goal defined for it. Management sponsorship must be solicited frequently by implementing proactive communication activities. The involvement of work groups, as well as middle-management support and resources allocated at project level, are discussed in the Software Engineering Process KA.

2. Software Quality Management Processes

Software quality management (SQM) applies to all perspectives of software processes, products, and resources. It defines processes, process owners, and requirements for those processes, measurements of the process and its outputs, and feedback channels. (Art93)

Software quality management processes consist of many activities. Some may find defects directly, while others indicate where further examination may be valuable. The latter are also referred to as direct-defect-finding activities. Many activities often serve as both.

Planning for software quality involves:

(1) Defining the required product in terms of its quality characteristics (described in more detail in, for instance, the Software Engineering Management KA).

(2) Planning the processes to achieve the required product (described in, for instance, the Software Design and the Software Construction KAs).

These aspects differ from, for instance, the planning SQM processes themselves, which assess planned quality characteristics versus actual implementation of those plans. **The software quality management processes must address how well software products will, or do, satisfy customer and stakeholder requirements, provide value to the customers and other stakeholders, and provide the software quality needed to meet software requirements.**

SQM can be used to evaluate the intermediate products as well as the final product.

Some of the specific SQM processes are defined in standard (IEEE12207.0-96):

- Quality assurance process
- Verification process
- Validation process
- Review process

- Audit process

These processes encourage quality and also find possible problems. But they differ somewhat in their emphasis.

SQM processes help ensure better software quality in a given project. They also provide, as a by-product, general information to management, including an indication of the quality of the entire software engineering process. The Software Engineering Process and Software Engineering Management KAs discuss quality programs for the organization developing the software. SQM can provide relevant feedback for these areas.

SQM processes consist of tasks and techniques to indicate how software plans (for example, management, development, configuration management) are being implemented and how well the intermediate and final products are meeting their specified requirements. Results from these tasks are assembled in reports for management before corrective action is taken. The management of an SQM process is tasked with ensuring that the results of these reports are accurate.

As described in this KA, SQM processes are closely related; they can overlap and are sometimes even combined. They seem largely reactive in nature because they address the processes as practiced and the products as produced; but they have a major role at the planning stage in being proactive in terms of the processes and procedures needed to attain the quality characteristics and degree of quality needed by the stakeholders in the software.

Risk management can also play an important role in delivering quality software. Incorporating disciplined risk analysis and management techniques into the software life cycle processes can increase the potential for producing a quality product (Cha89). Refer to the Software Engineering Management KA for related material on risk management.

2.1. Software Quality Assurance

[Ack02; Ebe94; Fre98; Gra92; Hor03; Pfl01; Pre04; Rak97; Sch99; Som05; Voa99; Wal89; Wal96]

SQA processes provide assurance that the software products and processes in the project life cycle conform to their specified requirements by planning, enacting, and performing a set of activities to provide adequate confidence that quality is being built into the software. This means ensuring that the problem is clearly and adequately stated and that the solution's requirements are properly defined and expressed. SQA seeks to maintain the quality throughout the development and maintenance of the product by the execution of a variety of activities at each stage which can result in early identification of problems, an almost inevitable feature of any complex activity. The role of SQA with respect to process is to ensure that planned processes are appropriate and later implemented according to plan, and that relevant measurement processes are provided to the appropriate organization.

The SQA plan defines the means that will be used to ensure that software developed for a specific product satisfies the user's requirements and is of the highest quality possible within project constraints. In order to do so, it must first ensure that the quality target is clearly defined and understood. It must consider management, development, and maintenance plans for the software. Refer to standard (IEEE730-98) for details.

The specific quality activities and tasks are laid out, with their costs and resource requirements, their overall management objectives, and their schedule in relation to those objectives in the software engineering management, development, or maintenance plans. The SQA plan should be consistent with the software configuration management plan (refer to the Software Configuration Management KA). The SQA plan identifies documents, standards, practices, and conventions governing the project and how they will be checked and monitored to ensure adequacy and compliance. The SQA plan also identifies measures, statistical techniques, procedures for problem reporting and corrective action, resources such as tools, techniques, and methodologies, security for physical media, training, and SQA reporting and documentation. Moreover, the SQA plan addresses the software quality assurance activities of any other type of activity described in the software plans, such as procurement of supplier software to the project or commercial off-the-shelf software (COTS) installation, and service after delivery of the software. It can also contain acceptance criteria as well as reporting and management activities which are critical to software quality.

2.2. Verification & Validation

[Fre98; Hor03; Pfl01; Pre04; Som05; Wal89; Wal96]

For purposes of brevity, Verification and Validation (V&V) are treated as a single topic in this *Guide* rather than as two separate topics as in the standard (IEEE12207.0-96). "Software V&V is a disciplined approach to assessing software products throughout the product life cycle. A V&V effort strives to ensure that quality is built into the software and that the software satisfies user requirements" (IEEE1059-93).

V&V addresses software product quality directly and uses testing techniques which can locate defects so that they can be addressed. It also assesses the intermediate products, however, and, in this capacity, the intermediate steps of the software life cycle processes.

The V&V process determines whether or not products of a given development or maintenance activity conform to the requirement of that activity, and whether or not the final software product fulfills its intended purpose and meets user requirements. Verification is an attempt to ensure that the product is built correctly, in the sense that the output products of an activity meet the specifications

imposed on them in previous activities. Validation is an attempt to ensure that the right product is built, that is, the product fulfills its specific intended purpose. Both the verification process and the validation process begin early in the development or maintenance phase. They provide an examination of key product features in relation both to the product's immediate predecessor and to the specifications it must meet.

The purpose of planning V&V is to ensure that each resource, role, and responsibility is clearly assigned. The resulting V&V plan documents and describes the various resources and their roles and activities, as well as the techniques and tools to be used. An understanding of the different purposes of each V&V activity will help in the careful planning of the techniques and resources needed to fulfill their purposes. Standards (IEEE1012-98:s7 and IEEE1059-93: Appendix A) specify what ordinarily goes into a V&V plan.

The plan also addresses the management, communication, policies, and procedures of the V&V activities and their interaction, as well as defect reporting and documentation requirements.

2.3. Reviews and Audits

For purposes of brevity, reviews and audits are treated as a single topic in this *Guide*, rather than as two separate topics as in (IEEE12207.0-96). The review and audit process is broadly defined in (IEEE12207.0-96) and in more detail in (IEEE1028-97). Five types of reviews or audits are presented in the IEEE1028-97 standard:

- Management reviews
- Technical reviews
- Inspections
- Walk-throughs
- Audits

2.3.1. Management reviews

"The purpose of a management review is to monitor progress, determine the status of plans and schedules, confirm requirements and their system allocation, or evaluate the effectiveness of management approaches used to achieve fitness for purpose" [IEEE1028-97]. They support decisions about changes and corrective actions that are required during a software project. Management reviews determine the adequacy of plans, schedules, and requirements and monitor their progress or inconsistencies. These reviews may be performed on products such as audit reports, progress reports, V&V reports, and plans of many types, including risk management, project management, software configuration management, software safety, and risk assessment, among others. Refer to the Software Engineering Management and to the Software Configuration Management KAs for related material.

2.3.2. Technical reviews

[Fre98; Hor03; Lew92; Pfl01; Pre04;

Som05; Voa99; Wal89; Wal96]

"The purpose of a technical review is to evaluate a software product to determine its suitability for its intended use. The objective is to identify discrepancies from approved specifications and standards. The results should provide management with evidence confirming (or not) that the product meets the specifications and adheres to standards, and that changes are controlled" (IEEE1028-97).

Specific roles must be established in a technical review: a decision-maker, a review leader, a recorder, and technical staff to support the review activities. A technical review requires that mandatory inputs be in place in order to proceed:

- Statement of objectives
- A specific software product
- The specific project management plan
- The issues list associated with this product
- The technical review procedure

The team follows the review procedure. A technically qualified individual presents an overview of the product, and the examination is conducted during one or more meetings. The technical review is completed once all the activities listed in the examination have been completed.

2.3.3. Inspections

[Ack02; Fre98; Gil93; Rad02; Rak97]

"The purpose of an inspection is to detect and identify software product anomalies" (IEEE1028-97). Two important differentiators of inspections as opposed to reviews are as follows:

1. An individual holding a management position over any member of the inspection team shall not participate in the inspection.

2. An inspection is to be led by an impartial facilitator who is trained in inspection techniques.

Software inspections always involve the author of an intermediate or final product, while other reviews might not. Inspections also include an inspection leader, a recorder, a reader, and a few (2 to 5) inspectors. The members of an inspection team may possess different expertise, such as domain expertise, design method expertise, or language expertise. Inspections are usually conducted on one relatively small section of the product at a time. Each team member must examine the software product and other review inputs prior to the review

meeting, perhaps by applying an analytical technique (refer to section 3.3.3) to a small section of the product, or to the entire product with a focus only on one aspect, for example, interfaces. Any anomaly found is documented and sent to the inspection leader. During the inspection, the inspection leader conducts the session and verifies that everyone has prepared for the inspection. A checklist, with anomalies and questions germane to the issues of interest, is a common tool used in inspections. The resulting list often classifies the anomalies (refer to IEEE1044-93 for details) and is reviewed for completeness and accuracy by the team. The inspection exit decision must correspond to one of the following three criteria:

1. Accept with no or at most minor reworking
2. Accept with rework verification
3. Reinspect

Inspection meetings typically last a few hours, whereas technical reviews and audits are usually broader in scope and take longer.

2.3.4. Walk-throughs

[Fre98; Hor03; Pfl01; Pre04; Som05;

Wal89; Wal96]

"The purpose of a walk-through is to evaluate a software product. A walk-through may be conducted for the purpose of educating an audience regarding a software product." (IEEE1028-97) The major objectives are to [IEEE1028-97]:

- Find anomalies
- Improve the software product
- Consider alternative implementations
- Evaluate conformance to standards and specifications

The walk-through is similar to an inspection but is typically conducted less formally. The walk-through is primarily organized by the software engineer to give his teammates the opportunity to review his work, as an assurance technique.

2.3.5. Audits

[Fre98; Hor03; Pfl01; Pre01; Som05;

Voa99; Wal89; Wal96]

"The purpose of a software audit is to provide an independent evaluation of the conformance of software products and processes to applicable regulations, standards, guidelines, plans, and procedures" [IEEE1028-97]. The audit is a formally organized activity, with participants having specific roles, such as lead auditor, another auditor, a recorder, or an initiator, and includes a representative of the audited organization. The audit will identify instances of nonconformance and produce a report requiring the team to take corrective action.

While there may be many formal names for reviews and audits such as those identified in the standard (IEEE1028-97), the important point is that they can occur on almost any product at any stage of the development or maintenance process.

3. Practical Considerations

3.1. Software Quality Requirements

[Hor03; Lew92; Rak97; Sch99; Wal89; Wal96]

3.1.1. Influence factors

Various factors influence planning, management, and selection of SQM activities and techniques, including:

- The domain of the system in which the software will reside (safety-critical, mission-critical, business-critical)
- System and software requirements
- The commercial (external) or standard (internal) components to be used in the system
- The specific software engineering standards applicable
- The methods and software tools to be used for development and maintenance and for quality evaluation and improvement
- The budget, staff, project organization, plans, and scheduling of all the processes
- The intended users and use of the system
- The integrity level of the system

Information on these factors influences how the SQM processes are organized and documented, how specific SQM activities are selected, what resources are needed, and which will impose bounds on the efforts.

3.1.2. Dependability

In cases where system failure may have extremely severe consequences, overall dependability (hardware, software, and human) is the main quality requirement over and above basic functionality. Software dependability includes such characteristics as fault tolerance, safety, security, and usability. Reliability is also a criterion which can be defined in terms of dependability (ISO9126).

The body of literature for systems must be highly dependable ("high confidence" or "high integrity systems"). Terminology for traditional mechanical and electrical systems which may not include software has been imported for discussing threats or hazards, risks, system integrity, and related concepts, and may be found in the references cited for this section.

3.1.3. Integrity levels of software

The integrity level is determined based on the possible consequences of failure of the software and the probability of failure. For software in which safety or security is important, techniques such as hazard analysis for safety or threat analysis for security may be used to develop a planning activity which would identify where potential trouble spots lie. The failure history of similar software may also help in identifying which techniques will be most useful in detecting faults and assessing quality. Integrity levels (for example, gradation of integrity) are proposed in (IEEE1012-98).

3.2. Defect Characterization

[Fri95; Hor03; Lew92; Rub94; Wak99; Wal89]

SQM processes find defects. Characterizing those defects leads to an understanding of the product, facilitates corrections to the process or the product, and informs project management or the customer of the status of the process or product. Many defect (fault) taxonomies exist, and, while attempts have been made to gain consensus on a fault and failure taxonomy, the literature indicates that there are quite a few in use [Bei90, Chi96, Gra92], IEEE1044-93) Defect (anomaly) characterization is also used in audits and reviews, with the review leader often presenting a list of anomalies provided by team members for consideration at a review meeting.

As new design methods and languages evolve, along with advances in overall software technologies, new classes of defects appear, and a great deal of effort is required to interpret previously defined classes. When tracking defects, the software engineer is interested in not only the number of defects but also the types. Information alone, without some classification, is not really of any use in identifying the underlying causes of the defects, since specific types of problems need to be grouped together in order for determinations to be made about them. The point is to establish a defect taxonomy that is meaningful to the organization and to the software engineers.

SQM discovers information at all stages of software development and maintenance. Typically, where the word "defect" is used, it refers to a "fault" as defined below. However, different cultures and standards may use somewhat different meanings for these terms, which have led to attempts to define them. Partial definitions taken from standard (IEEE610.12-90) are:

- *Error*: "A difference…between a computed result and the correct result"
- *Fault*: "An incorrect step, process, or data definition in a computer program"
- *Failure*: "The [incorrect] result of a fault"
- *Mistake*: "A human action that produces an incorrect result"

Failures found in testing as a result of software faults are included as defects in the discussion in this section. Reliability models are built from failure data collected during software testing or from software in service, and thus can be used to predict future failures and to assist in decisions on when to stop testing. [Mus89]

One probable action resulting from SQM findings is to remove the defects from the product under examination. Other actions enable the achievement of full value from the findings of SQM activities. These actions include analyzing and summarizing the findings, and using measurement techniques to improve the product and the process as well as to track the defects and their removal. Process improvement is primarily discussed in the Software Engineering Process KA, with the SQM process being a source of information.

Data on the inadequacies and defects found during the implementation of SQM techniques may be lost unless they are recorded. For some techniques (for example, technical reviews, audits, inspections), recorders are present to set down such information, along with issues and decisions. When automated tools are used, the tool output may provide the defect information. Data about defects may be collected and recorded on an SCR (software change request) form and may subsequently be entered into some type of database, either manually or automatically, from an analysis tool. Reports about defects are provided to the management of the organization.

3.3. *Software Quality Management Techniques*

[Bas94; Bei90; Con86; Chi96; Fen97; Fri95; Lev95; Mus89; Pen93; Sch99; Wak99; Wei93; Zel98]

SQM techniques can be categorized in many ways: static, people-intensive, analytical, dynamic.

3.3.1. Static techniques

Static techniques involve examination of the project documentation and software, and other information about the software products, without executing them. These techniques may include people-intensive activities (as defined in 3.3.2) or analytical activities (as defined in 3.3.3) conducted by individuals, with or without the assistance of automated tools.

3.3.2. People-intensive techniques

The setting for people-intensive techniques, including reviews and audits, may vary from a formal meeting to an informal gathering or a desk-check situation, but (usually, at least) two or more people are involved. Preparation ahead of time may be necessary. Resources other than the items under examination may include checklists and results from analytical techniques and testing. These activities are discussed in (IEEE1028-97) on reviews and audits. [Fre98, Hor03] and [Jon96, Rak97]

3.3.3. Analytical techniques

A software engineer generally applies analytical techniques. Sometimes several software engineers use the same technique, but each applies it to different parts of the product. Some techniques are tool-driven; others are manual. Some may find defects directly, but they are typically used to support other techniques. Some also include various assessments as part of overall quality analysis. Examples of such techniques include complexity analysis, control flow analysis, and algorithmic analysis.

Each type of analysis has a specific purpose, and not all types are applied to every project. An example of a support technique is complexity analysis, which is useful for determining whether or not the design or implementation is too complex to develop correctly, to test, or to maintain. The results of a complexity analysis may also be used in developing test cases. Defect-finding techniques, such as control flow analysis, may also be used to support another activity. For software with many algorithms, algorithmic analysis is important, especially when an incorrect algorithm could cause a catastrophic result. There are too many analytical techniques to list them all here. The list and references provided may offer insights into the selection of a technique, as well as suggestions for further reading.

Other, more formal, types of analytical techniques are known as formal methods. They are used to verify software requirements and designs. Proof of correctness applies to critical parts of software. They have mostly been used in the verification of crucial parts of critical systems, such as specific security and safety requirements. (Nas97)

3.3.4. Dynamic techniques

Different kinds of dynamic techniques are performed throughout the development and maintenance of software. Generally, these are testing techniques, but techniques such as simulation, model checking, and symbolic execution may be considered dynamic. Code reading is considered a static technique, but experienced software engineers may execute the code as they read through it. In this sense, code reading may also qualify as a dynamic technique. This discrepancy in categorizing indicates that people with different roles in the organization may consider and apply these techniques differently.

Some testing may thus be performed in the development process, SQA process, or V&V process, again depending on project organization. Because SQM plans address testing, this section includes some comments on testing. The Software Testing KA provides discussion and technical references to theory, techniques for testing, and automation.

3.3.5. Testing

The assurance processes described in SQA and V&V examine every output relative to the software requirement specification to ensure the output's traceability, consistency, completeness, correctness, and performance. This confirmation also includes the outputs of the development and maintenance processes, collecting, analyzing, and measuring the results. SQA ensures that appropriate types of tests are planned, developed, and implemented, and V&V develops test plans, strategies, cases, and procedures.

Testing is discussed in detail in the Software Testing KA. Two types of testing may fall under the headings SQA and V&V, because of their responsibility for the quality of the materials used in the project:

- Evaluation and test of tools to be used on the project (IEEE1462-98)
- Conformance test (or review of conformance test) of components and COTS products to be used in the product; there now exists a standard for software packages (IEEE1465-98)

Sometimes an independent V&V organization may be asked to monitor the test process and sometimes to witness the actual execution to ensure that it is conducted in accordance with specified procedures. Again, V&V may be called upon to evaluate the testing itself: adequacy of plans and procedures, and adequacy and accuracy of results.

Another type of testing that may fall under the heading of V&V organization is third-party testing. The third party is not the developer, nor is in any way associated with the development of the product. Instead, the third party is an independent facility, usually accredited by some body of authority. Their purpose is to test a product for conformance to a specific set of requirements.

3.4. Software Quality Measurement

[Gra92]

The models of software product quality often include measures to determine the degree of each quality characteristic attained by the product.

If they are selected properly, measures can support software quality (among other aspects of the software life cycle processes) in multiple ways. They can help in the management decision-making process. They can find problematic areas and bottlenecks in the software process; and they can help the software engineers assess the quality of their work for SQA purposes and for longer-term process quality improvement.

With the increasing sophistication of software, questions of quality go beyond whether or not the software works to how well it achieves measurable quality goals.

There are a few more topics where measurement supports SQM directly. These include assistance in deciding when to stop testing. For this, reliability models and benchmarks, both using fault and failure data, are useful.

The cost of SQM processes is an issue which is almost always raised in deciding how a project should be organized. Often, generic models of cost are used, which are based on when a defect is found and how much effort it takes to fix the defect relative to finding the defect earlier in the development process. Project data may give a better picture of cost. Discussion on this topic can be found in [Rak97: pp. 39-50]. Related information can be found in the Software Engineering Process and Software Engineering Management KAs.

Finally, the SQM reports themselves provide valuable information not only on these processes, but also on how all the software life cycle processes can be improved. Discussions on these topics are found in [McC04] and (IEEE1012-98).

While the measures for quality characteristics and product features may be useful in themselves (for example, the number of defective requirements or the proportion of defective requirements), mathematical and graphical techniques can be applied to aid in the interpretation of the measures. These fit into the following categories and are discussed in [Fen97, Jon96, Kan02, Lyu96, Mus99].

- Statistically based (for example, Pareto analysis, run charts, scatter plots, normal distribution)
- Statistical tests (for example, the binomial test, chi-squared test)
- Trend analysis
- Prediction (for example, reliability models)

The statistically based techniques and tests often provide a snapshot of the more troublesome areas of the software product under examination. The resulting charts and graphs are visualization aids which the decision-makers can use to focus resources where they appear most needed. Results from trend analysis may indicate that a schedule has not been respected, such as in testing, or that certain classes of faults will become more intense unless some corrective action is taken in development. The predictive techniques assist in planning test time and in predicting failure. More discussion on measurement in general appears in the Software Engineering Process and Software Engineering Management KAs. More specific information on testing measurement is presented in the Software Testing KA.

References [Fen97, Jon96, Kan02, Pfl01] provide discussion on defect analysis, which consists of measuring defect occurrences and then applying statistical methods to understanding the types of defects that occur most frequently, that is, answering questions in order to assess their density. They also aid in understanding the trends and how well detection techniques are working, and how well the development and maintenance processes are progressing. Measurement of test coverage helps to estimate how much test effort remains to be done, and to predict possible remaining defects. From these measurement methods, defect profiles can be developed for a specific application domain. Then, for the next software system within that organization, the profiles can be used to guide the SQM processes, that is, to expend the effort where the problems are most likely to occur. Similarly, benchmarks, or defect counts typical of that domain, may serve as one aid in determining when the product is ready for delivery.

Discussion on using data from SQM to improve development and maintenance processes appears in the Software Engineering Management and the Software Engineering Process KAs.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	[Boe78]	[Dac01]	[Hou99]	[IEEE99]	[ISO9001-00]	[ISO90003-04]	[Jon96]	[Kia95]	[Lap91]	[Lew92]	[Llo03]	[McC77]	[Mus99]	[NIST03]	[Pfl01]	[Pre04]	[Rak97]	[Sei02]	[Wal96]	[Wei93]	[Wei96]
1. Software Quality Fundamental																					
1.1 Software Engineering Culture and Ethics				*																	*
1.2 Value and Cost of Quality	*		*				*							*	*	*				*	*
1.3 Models and Quality Characteristics	*	*			*	*		*	*	*	*	*	*	*		*	*	*	*		
1.4 Software Quality Improvement														*		*					*

	[Ack02]	[Ebe94]	[Fre98]	[Gil93]	[Gra92]	[Hor03]	[Lew92]	[Pfl01]	[Pre04]	[Rad02]	[Rak97]	[Sch99]	[Som05]	[Voa99]	[Wal89]	[Wal96]
2. Software Quality Management Processes																
2.1 Software Quality Assurance	*	*	*		*	*		*	*		*	*	*	*	*	*
2.2 Verification and Validation			*			*		*	*				*		*	*
2.3 Reviews and Audits	*		*	*		*	*	*	*	*	*		*	*	*	*

	[Bas84]	[Bei90]	[Con86]	[Chi96]	[Fen97]	[Fre98]	[Fri95]	[Gra92]	[Hor03]	[Jon96]	[Kan02]	[Lev95]	[Lew92]	[Lyu96]	[McC04]	[Mus89]	[Mus99]	[Pen93]	[Pfl01]	[Rak97]	[Rub94]	[Sch99]	[Wak99]	[Wal89]	[Wal96]	[Wei93]	[Zel98]
3. Software Quality Practical Considerations																											
3.1 Software Quality Requirements									*				*							*		*		*	*		
3.2 Defect Characterization		*		*			*	*	*				*			*					*		*	*			
3.3 SQM Techniques	*	*	*	*	*	*	*		*	*		*				*		*		*		*	*			*	*
3.4 Software Quality Measurement					*			*		*	*			*	*		*		*	*							

RECOMMENDED REFERENCES FOR SOFTWARE QUALITY

[Ack02] F.A. Ackerman, "Software Inspections and the Cost Effective Production of Reliable Software," *Software Engineering, Volume 2: The Supporting Processes*, Richard H. Thayer and Mark Christensen, eds., Wiley-IEEE Computer Society Press, 2002.

[Bas84] V.R. Basili and D.M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, vol. SE-10, iss. 6, November 1984, pp. 728-738.

[Bei90] B. Beizer, *Software Testing Techniques*, International Thomson Press, 1990.

[Boe78] B.W. Boehm et al., "Characteristics of Software Quality," *TRW Series on Software Technologies*, vol. 1, 1978.

[Chi96] R. Chillarege, "Orthogonal Defect Classification," *Handbook of Software Reliability Engineering*, M. Lyu, ed., IEEE Computer Society Press, 1996.

[Con86] S.D. Conte, H.E. Dunsmore, and V.Y. Shen, *Software Engineering Metrics and Models*: The Benjamin Cummings Publishing Company, 1986.

[Dac01] G. Dache, "IT Companies will gain competitive advantage by integrating CMM with ISO9001," *Quality System Update*, vol. 11, iss. 11, November 2001.

[Ebe94] R.G. Ebenau and S. Strauss, *Software Inspection Process*, McGraw-Hill, 1994.

[Fen98] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, second ed., International Thomson Computer Press, 1998.

[Fre98] D.P. Freedman and G.M. Weinberg, *Handbook of Walkthroughs, Inspections, and Technical Reviews*, Little, Brown and Company, 1998.

[Fri95] M.A. Friedman and J.M. Voas, *Software Assessment: Reliability, Safety Testability*, John Wiley & Sons, 1995.

[Gil93] T. Gilb and D. Graham, *Software Inspections*, Addison-Wesley, 1993.

[Gra92] R.B. Grady, *Practical Software Metrics for Project Management and Process Management*, Prentice Hall, 1992.

[Hor03] J. W. Horch, *Practical Guide to Software Quality Management*, Artech House Publishers, 2003.

[Hou99] D. Houston, "Software Quality Professional," *ASQC*, vol. 1, iss. 2, 1999.

[IEEE-CS-99] IEEE-CS-1999, "Software Engineering Code of Ethics and Professional Practice," IEEE-CS/ACM, 1999, available at http://www.computer.org/certification/ethics.htm.

[ISO9001-00] ISO 9001:2000, *Quality Management Systems — Requirements*, ISO, 2000.

[ISO90003-04] ISO/IEC 90003:2004, *Software and Systems Engineering-Guidelines for the Application of ISO9001:2000 to Computer Software*, ISO and IEC, 2004.

[Jon96] C. Jones and J. Capers, *Applied Software Measurement: Assuring Productivity and Quality*, second ed., McGraw-Hill, 1996.

[Kan02] S.H. Kan, *Metrics and Models in Software Quality Engineering*, second ed., Addison-Wesley, 2002.

[Kia95] D. Kiang, "Harmonization of International Software Standards on Integrity and Dependability," *Proc. IEEE International Software Engineering Standards Symposium*, IEEE Computer Society Press, 1995.

[Lap91] J.C. Laprie, *Dependability: Basic Concepts and Terminology in English, French, German, Italian and Japanese, IFIP WG 10.4*, Springer-Verlag, 1991.

[Lev95] N.G. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley, 1995.

[Lew92] R.O. Lewis, *Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software*, John Wiley & Sons, 1992.

[Llo03] Lloyd's Register, "TickIT Guide," iss. 5, 2003, available at http://www.tickit.org.

[Lyu96] M.R. Lyu, *Handbook of Software Reliability Engineering*: McGraw-Hill/IEEE, 1996.

[Mcc77] J.A. McCall, "Factors in Software Quality — General Electric," n77C1502, June 1977.

[McC04] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, second ed., 2004.

[Mus89] J.D. Musa and A.F. Ackerman, "Quantifying Software Validation: When to Stop Testing?" *IEEE Software*, vol. 6, iss. 3, May 1989, pp. 19-27.

[Mus99] J. Musa, *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*: McGraw Hill, 1999.

[NIST03] National Institute of Standards and Technology, "Baldrige National Quality Program," available at http://www.quality.nist.gov.

[Pen93] W.W. Peng and D.R. Wallace, "Software Error Analysis," National Institute of Standards and Technology, Gaithersburg, NIST SP 500-209, December 1993, available at http://hissa.nist.gov/SWERROR/.

[Pfl01] S.L. Pfleeger, *Software Engineering: Theory and Practice*, second ed., Prentice Hall, 2001.

[Pre04] R.S. Pressman, *Software Engineering: A Practitioner's Approach*, sixth ed., McGraw-Hill, 2004.

[Rad02] R. Radice, *High Quality Low Cost Software Inspections*, Paradoxicon, 2002, p. 479.

[Rak97] S.R. Rakitin, *Software Verification and Validation: A Practitioner's Guide*, Artech House, 1997.

[Rub94] J. Rubin, *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*, John Wiley & Sons, 1994.

[Sch99] G.C. Schulmeyer and J.I. McManus, *Handbook of Software Quality Assurance*, third ed., Prentice Hall, 1999.

[SEI02] "Capability Maturity Model Integration for Software Engineering (CMMI)," CMU/SEI-2002-TR-028, ESC-TR-2002-028, Software Engineering Institute, Carnegie Mellon University, 2002.

[Som05] I. Sommerville, *Software Engineering*, seventh ed., Addison-Wesley, 2005.

[Voa99] J. Voas, "Certifying Software For High Assurance Environments," *IEEE Software*, vol. 16, iss. 4, July-August 1999, pp. 48-54.

[Wak99] S. Wakid, D.R. Kuhn, and D.R. Wallace, "Toward Credible IT Testing and Certification," *IEEE Software*, July/August 1999, pp. 39-47.

[Wal89] D.R. Wallace and R.U. Fujii, "Software Verification and Validation: An Overview," *IEEE Software*, vol. 6, iss. 3, May 1989, pp. 10-17.

[Wal96] D.R. Wallace, L. Ippolito, and B. Cuthill, "Reference Information for the Software Verification and Validation Process," NIST SP 500-234, NIST, April 1996, available at http://hissa.nist.gov/VV234/.

[Wei93] G.M. Weinberg, "Measuring Cost and Value," *Quality Software Management: First-Order Measurement*, vol. 2, chap. 8, Dorset House, 1993.

[Wie96] K. Wiegers, *Creating a Software Engineering Culture*, Dorset House, 1996.

[Zel98] M.V. Zelkowitz and D.R. Wallace, "Experimental Models for Validating Technology," *Computer*, vol. 31, iss. 5, 1998, pp. 23-31.

APPENDIX A. LIST OF FURTHER READINGS

(Abr96) A. Abran and P.N. Robillard, "Function Points Analysis: An Empirical Study of Its Measurement Processes," presented at IEEE Transactions on Software Engineering, 1996. **//journal or conference?//**

(Art93) L.J. Arthur, *Improving Software Quality: An Insider's Guide to TQM*, John Wiley & Sons, 1993.

(Bev97) N. Bevan, "Quality and Usability: A New Framework," *Achieving Software Product Quality*, E. v. Veenendaal and J. McMullan, eds., Uitgeverij Tutein Nolthenius, 1997.

(Cha89) R.N. Charette, *Software Engineering Risk Analysis and Management,* McGraw-Hill, 1989.

(Cro79) P.B. Crosby, *Quality Is Free*, McGraw-Hill, 1979.

(Dem86) W.E. Deming, *Out of the Crisis*: MIT Press, 1986.

(Dod00) Department of Defense and US Army, "Practical Software and Systems Measurement: A Foundation for Objective Project Management, Version 4.0b," October 2000, available at http://www.psmsc.com.

(Hum89) W. Humphrey, "Managing the Software Process," Chap. 8, 10, 16, Addison-Wesley, 1989.

(Hya96) L.E. Hyatt and L. Rosenberg, "A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality," presented at 8th Annual Software Technology Conference, 1996.

(Inc94) D. Ince, *ISO 9001 and Software Quality Assurance*, McGraw-Hill, 1994.

(Jur89) J.M. Juran, *Juran on Leadership for Quality*, The Free Press, 1989.

(Kin92) M.R. Kindl, "Software Quality and Testing: What DoD Can Learn from Commercial Practices," U.S. Army Institute for Research in Management Information, Communications and Computer Sciences, Georgia Institute of Technology, August 1992.

(NAS97) NASA, "Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, Volume II: A Practitioner's Companion," 1997, available at http://eis.jpl.nasa.gov/quality/Formal_Methods/.

(Pal97) J.D. Palmer, "Traceability," *Software Engineering*, M. Dorfman and R. Thayer, eds., 1997, pp. 266-276.

(Ros98) L. Rosenberg, "Applying and Interpreting Object-Oriented Metrics," presented at Software Technology Conference, 1998, available at http://satc.gsfc.nasa.gov/support/index. html.

(Sur03) W. Suryn, A. Abran, and A. April, "ISO/IEC SQuaRE. The Second Generation of Standards for Software Product Quality," presented at IASTED 2003, 2003.

(Vin90) W.G. Vincenti, *What Engineers Know and How They Know It — Analytical Studies from Aeronautical History*, John Hopkins University Press, 1990.

APPENDIX B. LIST OF STANDARDS

(FIPS140.1-94) FIPS 140-1, *Security Requirements for Cryptographic Modules*, 1994.

(IEC61508-98) IEC 61508, *Functional Safety — Safety-Related Systems Parts 1, 2, 3*, IEEE, 1998.

(IEEE610.12-90) IEEE Std 610.12-1990 (R2002), *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1990.

(IEEE730-02) IEEE Std 730-2002, *IEEE Standard for Software Quality Assurance Plans*, IEEE, 2002.

(IEEE982.1-88) IEEE Std 982.1-1988, *IEEE Standard Dictionary of Measures to Produce Reliable Software*, 1988.

(IEEE1008-87) IEEE Std 1008-1987 (R2003), *IEEE Standard for Software Unit Testing*, IEEE, 1987.

(IEEE1012-98) IEEE Std 1012-1998, *Software Verification and Validation*, IEEE, 1998.

(IEEE1028-97) IEEE Std 1028-1997 (R2002), *IEEE Standard for Software Reviews*, IEEE, 1997.

(IEEE1044-93) IEEE Std 1044-1993 (R2002), *IEEE Standard for the Classification of Software Anomalies*, IEEE, 1993.

(IEEE1059-93) IEEE Std 1059-1993, *IEEE Guide for Software Verification and Validation Plans*, IEEE, 1993.

(IEEE1061-98) IEEE Std 1061-1998, *IEEE Standard for a Software Quality Metrics Methodology*, IEEE, 1998.

(IEEE1228-94) IEEE Std 1228-1994, *Software Safety Plans*, IEEE, 1994.

(IEEE1462-98) IEEE Std 1462-1998//ISO/IEC14102, *Information Technology — Guideline for the Evaluation and Selection of CASE Tools*.

(IEEE1465-98) IEEE Std 1465-1998//ISO/IEC12119:1994, *IEEE Standard Adoption of International Standard IDO/IEC12119:1994(E), Information Technology-Software Packages — Quality Requirements and Testing*, IEEE, 1998.

(IEEE12207.0-96) IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.

(ISO9001-00) ISO 9001:2000, *Quality Management Systems — Requirements*, ISO, 2000.

(ISO9126-01) ISO/IEC 9126-1:2001, *Software Engineering — Product Quality, Part 1: Quality Model*, ISO and IEC, 2001.

(ISO14598-98) ISO/IEC 14598:1998, *Software Product Evaluation*, ISO and IEC, 1998.

(ISO15026-98) ISO/IEC 15026:1998, *Information Technology — System and Software Integrity Levels*, ISO and IEC, 1998.

(ISO15504-98) ISO/IEC TR 15504-1998, *Information Technology — Software Process Assessment (parts 1-9)*, ISO and IEC, 1998.

(ISO15939-00) ISO/IEC 15939:2000, *Information Technology — Software Measurement Process*, ISO and IEC, 2000.

(ISO90003-04) ISO/IEC 90003:2004, *Software and Systems Engineering — Guidelines for the Application of ISO9001:2000 to Computer Software*, ISO and IEC, 2004.

CHAPTER 12

RELATED DISCIPLINES OF SOFTWARE ENGINEERING

INTRODUCTION

In order to circumscribe software engineering, it is necessary to identify the disciplines with which software engineering shares a common boundary. This chapter identifies, in alphabetical order, these Related Disciplines. Of course, the Related Disciplines also share many common boundaries between themselves.

Using a consensus-based recognized source, this chapter identifies for each Related Discipline:

- An informative definition (when feasible)
- A list of knowledge areas

Figure 1 gives a graphical representation of these Related Disciplines.

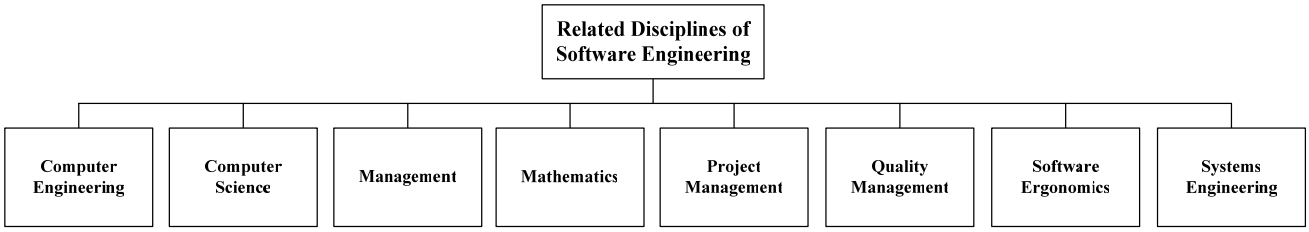


Figure 1 Related Disciplines of Software Engineering

LIST OF RELATED DISCIPLINES AND THEIR KNOWLEDGE AREAS

Computer Engineering

The draft report of the volume on computer engineering of the Computing Curricula 2001 project (CC2001)[1] states that "computer engineering embodies the science and technology of design, construction, implementation and maintenance of software and hardware components of modern computing systems and computer-controlled equipment."

This report identifies the following Knowledge Areas (known as areas in the report) for computer engineering:

- Algorithms and Complexity
- Computer Architecture and Organization
- Computer Systems Engineering
- Circuits and Systems
- Digital Logic
- Discrete Structures
- Digital Signal Processing
- Distributed Systems
- Electronics
- Embedded Systems
- Human-Computer Interaction
- Information Management
- Intelligent Systems
- Computer Networks
- Operating Systems
- Programming Fundamentals
- Probability and Statistics
- Social and Professional Issues
- Software Engineering
- Test and Verification
- VLSI/ASIC Design

[1] http://www.eng.auburn.edu/ece/CCCE/Iron_Man_Draft_October_2003.pdf

Computer Science

The final report of the volume on computer science of the Computing Curricula 2001 project (CC2001)[2] identifies the following list of knowledge areas (identified as areas in the report) for computer science:

- Discrete Structures
- Programming Fundamentals
- Algorithms and Complexity
- Architecture and Organization
- Operating Systems
- Net-Centric Computing
- Programming Languages
- Human-Computer Interaction
- Graphics and Visual Computing
- Intelligent Systems
- Information Management
- Social and Professional Issues
- Software Engineering
- Computational Science and Numerical Methods

Management

The European MBA Guidelines defined by the European association of national accreditation bodies (EQUAL)[3] states that the Master of Business Administration degree should include coverage of and instruction in

1) Accounting
- Finance
- Marketing and Sales
- Operations Management
- Information Systems Management
- Law
- Human Resource Management
- Economics
- Quantitative Analysis
- Business Policy and Strategy

Mathematics

Two sources are selected to identify the list of knowledge areas for mathematics. The report titled "Accreditation Criteria and Procedures"[4] of the Canadian Engineering Accreditation Board identifies that appropriate elements of the following areas should be present in an undergraduate engineering curriculum:

- Linear Algebra
- Differential and Integral Calculus
- Differential Equations
- Probability
- Statistics
- Numerical analysis
- Discrete Mathematics

A more focused list of mathematical topics (called units and topics in the report) that underpin software engineering can be found in the draft report of the volume on software engineering of the Computing Curricula 2001 project (CC2001).[5]

Project Management

Project management is defined in the 2000 Edition of *A Guide to the Project Management Body of Knowledge* (PMBOK® Guide[6]) published by the Project Management Institute and adopted as IEEE Std 1490-2003, as "the application of knowledge, skills, tools, and techniques to project activities to meet project requirements."

The Knowledge Areas identified in the PMBOK Guide for project management are

- Project Integration Management
- Project Scope Management
- Project Time Management
- Project Cost Management
- Project Quality Management
- Project Human Resource Management
- Project Communications Management
- Project Risk Management
- Project Procurement Management

Quality Management

Quality management is defined in ISO 9000-2000 as "coordinated activities to direct and control an organization with regard to quality." The three selected reference on quality management are

- ISO 9000:2000 Quality management systems -- Fundamentals and vocabulary
- ISO 9001:2000 Quality management systems – Requirements
- ISO 9004:2000 Quality management systems -- Guidelines for performance improvements

The American Society for Quality identifies the following Knowledge Areas (first-level breakdown topics in their

[2] http://www.computer.org/education/cc2001/final/cc2001.pdf
[3] http://www.efmd.be/
[4] http://www.ccpe.ca/e/files/report_ceab.pdf
[5] http://sites.computer.org/ccse/volume/FirstDraft.pdf
[6] PMBOK is a registered trademark in the United States and other nations.

outline) in their Body of Knowledge for certification as a Quality Engineer:[7]

2) Management and Leadership in Quality Engineering

- Quality Systems Development, Implementation And Verification
- Planning, Controlling, and Assuring Product and Process Quality
- Reliability and Risk Management
- Problem Solving and Quality Improvement
- Quantitative Methods

Software Ergonomics

The field of ergonomics is defined by ISO Technical Committee 159 on Ergonomics as follows: "Ergonomics or (human factors) is the scientific discipline concerned with the understanding of the interactions among human and other elements of a system, and the profession that applies theory, principles, data and methods to design in order to optimize human well-being and overall system performance."[8]

A list of Knowledge Areas for ergonomics as it applies to software is proposed below:[9]

- Cognition
- Cognitive AI I: Reasoning
- Machine Learning and Grammar Induction
- Formal Methods in Cognitive Science: Language
- Formal Methods in Cognitive Science: Reasoning
- Formal Methods in Cognitive Science:
  --Cognitive Architecture
- Cognitive AI II: Learning
- Foundations of Cognitive Science
- Information Extraction from Speech and Text
- Lexical Processing
- Computational Language Acquisition
- The Nature of HCI
  --(Meta-)Models of HCI
- Use and Context of Computers
  --Human Social Organization and Work
  --Application Areas
- Human-Machine Fit and Adaptation
- Human Characteristics
  --Human Information Processing
  --Language, Communication, Interaction
  --Ergonomics
- Computer System and Interface Architecture
  --Input and Output Devices
  --Dialogue Techniques
  --Dialogue Genre
  --Computer Graphics
- Dialogue Architecture
- Development Process
  --Design Approaches
  --Implementation Techniques
  --Evaluation Techniques
  --Example Systems and Case Studies

A more focused list of topics on human-computer interface design (called units and topics in the report) for software engineering curriculum purposes can be found in the draft report of the volume on software engineering of the Computing Curricula 2001 project (CC2001).[10]

Systems Engineering

The International Council on Systems Engineering (INCOSE)[11] states that "Systems Engineering is an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem: operations performance, test, manufacturing, cost and schedule, training and support and disposal."

Systems engineering integrates all the disciplines and specialty groups into a team effort forming a structured development process that proceeds from concept to production to operation. Systems engineering considers both the business and the technical needs of all customers with the goal of providing a quality product that meets user needs.

The International Council on Systems Engineering (INCOSE, www.incose.org) is working on a Guide to the Systems Engineering Body of Knowledge. Preliminary versions include the following first-level competency areas:

1. Business Processes and Operational Assessment (BPOA)

[7] http://isotc.iso.ch/livelink/livelink.exe/fetch/2000/2122/687806/ISO_TC_159__Ergonomics_.pdf?nodeid=1162319&vernum=0http://www.asq.org/cert/types/cqe/bok.html

[8] http://isotc.iso.ch/livelink/livelink.exe/fetch/2000/2122/687806/ISO_TC_159_Ergonomics_.pdf?nodeid=1162319&vernum=0

[9] This list was compiled for the 2001 edition of the SWEBOK Guide from the list of courses offered at the John Hopkins University Department of Cognitive Sciences and from the ACM SIGCHI Curricula for Human-Computer Interaction.
The list was then refined by three experts in the field: two from Université du Québec à Montréal and W. W. McMillan, from Eastern Michigan University. They were asked to indicate which of these topics should be known by a software engineer. The topics that were rejected by two of the three respondents were removed from the original list.

[10] http://sites.computer.org/ccse/volume/FirstDraft.pdf

[11] www.incose.org

2. System/Solution/Test Architecture (SSTA)
3. Life Cycle Cost & Cost-Benefit Analysis (LCC & CBA)
4. Serviceability / Logistics (S/L)
5. Modeling, Simulation, & Analysis (MS&A)
6. Management: Risk, Configuration, Baseline (Mgt)

APPENDIX A

KNOWLEDGE AREA DESCRIPTION SPECIFICATIONS FOR THE IRONMAN VERSION OF THE GUIDE TO THE SOFTWARE ENGINEERING BODY OF KNOWLEDGE

INTRODUCTION

This document presents version 1.9 of the specifications provided by the Editorial Team to the Knowledge Area Specialist regarding the Knowledge Area Descriptions of the Guide to the Software Engineering Body of Knowledge (Ironman Version).

This document begins by presenting specifications on the contents of the Knowledge Area Description. Criteria and requirements are defined for proposed breakdowns of topics, for the rationale underlying these breakdowns and the succinct description of topics, for selecting reference materials, and for identifying relevant Knowledge Areas of Related Disciplines. Important input documents are also identified and their role within the project is explained. Non-content issues such as submission format and style guidelines are also discussed.

CRITERIA AND REQUIREMENTS FOR PROPOSING THE BREAKDOWN(S) OF TOPICS WITHIN A KNOWLEDGE AREA

The following requirements and criteria should be used when proposing a breakdown of topics within a given Knowledge Area:

a) Associate editors are expected to propose one or possibly two complementary breakdowns that are specific to their Knowledge Area. The topics found in all breakdowns within a given Knowledge Area must be identical.

b) These breakdowns of topics are expected to be "reasonable," not "perfect." The Guide to the Software Engineering Body of Knowledge is definitely viewed as a multiphase effort, and many iterations within each phase as well as multiple phases will be necessary to continuously improve these breakdowns.

c) The proposed breakdown of topics within a Knowledge Area must decompose the subset of the Software Engineering Body of Knowledge that is "generally accepted." See below a more detailed discussion on this.

d) The proposed breakdown of topics within a Knowledge Area must not presume specific application domains, business needs, sizes of organizations, organizational structures, management philosophies, software life cycle models, software technologies, or software development methods.

e) The proposed breakdown of topics must, as much as possible, be compatible with the various schools of thought within software engineering.

f) The proposed breakdown of topics within Knowledge Areas must be compatible with the breakdown of software engineering generally found in industry and in the software engineering literature and standards.

g) The proposed breakdown of topics is expected to be as inclusive as possible. It is deemed better to suggest too many topics and have them abandoned later than to do the reverse.

h) The Knowledge Area Associate Editors are expected to adopt the position that even though the following "themes" are common across all Knowledge Areas, they are also an integral part of all Knowledge Areas and therefore must be incorporated into the proposed breakdown of topics of each Knowledge Area. These common themes are quality (in general) and measurement.

   Please note that the issue of how to properly handle these "cross-running" or "orthogonal topics" and whether or not they should be handled in a different manner has not been completely resolved yet.

i) The proposed breakdowns should be at most two or three levels deep. Even though no upper or lower limit is imposed on the number of topics within each Knowledge Area, Knowledge Area Associate Editors are expected to propose a reasonable and manageable number of topics per Knowledge Area. Emphasis should also be put on the selection of the topics themselves rather than on their organization in an appropriate hierarchy.

   Proposed topic names must be significant enough to be meaningful even when cited outside the Guide to the Software Engineering Body of Knowledge.

j) The description of a Knowledge Area will include a chart (in tree form) describing the knowledge breakdown.

CRITERIA AND REQUIREMENTS FOR DESCRIBING TOPICS

a) Topics need only to be sufficiently described so the reader can select the appropriate reference material according to his/her needs.

CRITERIA AND REQUIREMENTS FOR SELECTING REFERENCE MATERIAL

a) Specific reference material must be identified for each topic. Each reference material can of course cover multiple topics.

b) Proposed reference material can be book chapters, refereed journal papers, refereed conference papers, refereed technical or industrial reports, or any other type of recognized artifact such as web documents. They must be generally available and must not be confidential in nature. Reference should be as precise as possible by identifying what specific chapter or section is relevant.

c) Proposed reference material must be in English.

d) A reasonable amount of reference material must be selected for each Knowledge Area. The following guidelines should be used in determining how much is reasonable:

- If the reference material were written in a coherent manner that followed the proposed breakdown of topics and in a uniform style (for example in a new book based on the proposed Knowledge Area description), an average target for the number of pages would be 500. However, this target may not be attainable when selecting existing reference material due to differences in style and overlap and redundancy between the selected reference materials.
- The amount of reference material would be reasonable if it consisted of the study material on this Knowledge Area of a software engineering licensing exam that a graduate would pass after completing four years of work experience.
- The Guide to the Software Engineering Body of Knowledge is intended by definition to be selective in its choice of topics and associated reference material. The list of reference material for each Knowledge Area should be viewed and will be presented as an "informed and reasonable selection" rather than as a definitive list.
- Additional reference material can be included in a "Further Readings" list. These further readings still must be related to the topics in the breakdown. They must also discuss generally accepted knowledge. There should not be a matrix between the reference material listed in Further Readings and the individual topics.

e) If deemed feasible and cost-effective by the IEEE Computer Society, selected reference material will be published on the Guide to the Software Engineering Body of Knowledge web site. To facilitate this task, preference should be given to reference material for which the copyrights already belong to the IEEE Computer Society. This should however not be seen as a constraint or an obligation.

f) A matrix of reference material versus topics must be provided.

CRITERIA AND REQUIREMENTS FOR IDENTIFYING KNOWLEDGE AREAS OF THE RELATED DISCIPLINES

Knowledge Area Associate Editors are expected to identify in a separate section which Knowledge Areas of the Related Disciplines are sufficiently relevant to the Software Engineering Knowledge Area that has been assigned to them be expected knowledge by a graduate plus four years of experience.

This information will be particularly useful to and will engage much dialogue between the Guide to the Software Engineering Body of Knowledge initiative and our sister initiatives responsible for defining a common software engineering curricula and standard performance norms for software engineers.

The list of Knowledge Areas of Related Disciplines can be found in the Proposed Baseline List of Related Disciplines. If deemed necessary and if accompanied by a justification, Knowledge Area Specialists can also propose additional Related Disciplines not already included or identified in the Proposed Baseline List of Related Disciplines. (Please note that a classification of the topics from the Related Disciplines has been produced but will be published on the web site at a latter date in a separate working document. Please contact the editorial team for more information).

COMMON TABLE OF CONTENTS

Knowledge Area descriptions should use the following table of contents:

- Introduction
- Breakdown of topics of the Knowledge Area (for clarity purposes, we believe this section should be placed in front and not in an appendix at the end of the document. Also, it should be accompanied by a figure describing the breakdown)
- Matrix of topics vs. Reference material
- Recommended references for the Knowledge Area being described (please do not mix them with references used to write the Knowledge Area description)
- List of Further Readings

WHAT DO WE MEAN BY "GENERALLY ACCEPTED KNOWLEDGE"?

The software engineering body of knowledge is an all-inclusive term that describes the sum of knowledge within the profession of software engineering. However, the Guide to the Software Engineering Body of Knowledge seeks to identify and describe that subset of the body of knowledge that is generally accepted or, in other words, the core body of knowledge. To better illustrate what "generally accepted knowledge" is relative to other types of knowledge, Figure 1 proposes a draft three-category schema for classifying knowledge.

The Project Management Institute in its Guide to the Project Management Body of Knowledge[1] defines "generally accepted" knowledge for project management in the following manner:

"'Generally accepted' means that the knowledge and practices described are applicable to most projects most of the time, and that there is widespread consensus about their value and usefulness. 'Generally accepted' does not mean that the knowledge and practices described are or should be applied uniformly on all projects; the project management team is always responsible for determining what is appropriate for any given project."

The Guide to the Project Management Body of Knowledge is now an IEEE Standard.

At the Mont Tremblant kick-off meeting in 1998, the Industrial Advisory Board better defined "generally accepted" as knowledge to be included in the study material of a software engineering licensing exam that a graduate would pass after completing four years of work experience. These two definitions should be seen as complementary.

Knowledge Area Associate Editors are also expected to be somewhat forward looking in their interpretation by taking into consideration not only what is "generally accepted" today and but what they expect will be "generally accepted" in a 3- to 5-year timeframe.

[1] See "A Guide to the Project Management Body of Knowledge," Project Management Institute, Newton Square, PA 1996, 2000; available from www.pmi.org.

Figure 1 Categories of knowledge

LENGTH OF KNOWLEDGE AREA DESCRIPTION

Knowledge Area Descriptions are currently expected to be roughly in the 10-page range using the format of the International Conference on Software Engineering format as defined below. This includes text, references, appendices, tables, etc. This, of course, does not include the reference materials themselves. This limit should, however, not be seen as a constraint or an obligation.

ROLE OF EDITORIAL TEAM

Alain Abran and James W. Moore are the Executive Editors and are responsible for maintaining good relations with the IEEE Computer Society, the Industrial Advisory Board, the Executive Change Control Board, and the Panel of Experts as well as for the overall strategy, approach, organization, and funding of the project.

Pierre Bourque and Robert Dupuis are the Editors and are responsible for the coordination, operation, and logistics of this project. More specifically, the Editors are responsible for developing the project plan and the Knowledge Area description specification, coordinating Knowledge Area Associate Editors and their contribution, recruiting the reviewers and the review captains, as well as coordinating the various review cycles.

The Editors are therefore responsible for the coherence of the entire Guide and for identifying and establishing links between the Knowledge Areas. The Editors and the Knowledge Area Associate Editors will negotiate the resolution of gaps and overlaps between Knowledge Areas.

IMPORTANT RELATED DOCUMENTS (IN ALPHABETICAL ORDER OF FIRST AUTHOR)

1. P. Bourque, R. Dupuis, A. Abran, J. W. Moore, L. Tripp, and D. Frailey, "A Baseline List of Knowledge Areas for the Stone Man Version of the Guide to the Software Engineering Body of Knowledge,"

Université du Québec à Montréal, Montréal, February 1999.

Based on the Straw Man version, on the discussions held and the expectations stated at the kick-off meeting of the Industrial Advisory Board, on other body-of-knowledge proposals, and on criteria defined in this document, this document proposes a baseline list of ten Knowledge Areas for the Trial Version of the Guide to the Software Engineering Body of Knowledge. This baseline may of course evolve as work progresses and issues are identified during the course of the project.

This document is available at www.swebok.org.

2. P. Bourque, R. Dupuis, A. Abran, J. W. Moore, and L. Tripp**,** "A Proposed Baseline List of Related Disciplines for the Stone Man Version of the Guide to the Software Engineering Body of Knowledge," Université du Québec à Montréal, Montréal, February 1999.

Based on the Straw Man version, on the discussions held and the expectations stated at the kick-off meeting of the Industrial Advisory Board, and on subsequent work, this document proposes a baseline list of Related Disciplines and Knowledge Areas within these Related Disciplines. This document has been submitted to and discussed with the Industrial Advisory Board, and a recognized list of Knowledge Areas still has to be identified for certain Related Disciplines. Associate editors will be informed of the evolution of this document.

The current version is available at www.swebok.org.

3. P. Bourque, R. Dupuis, A. Abran, J. W. Moore, L. Tripp, K. Shyne, B. Pflug, M. Maya, and G. Tremblay, *Guide to the Software Engineering Body of Knowledge - A Straw Man Version*, technical report, Université du Québec à Montréal, Montréal, September 1998.

This report is the basis for the entire project. It defines general project strategy, rationale, and underlying principles and proposes an initial list of Knowledge Areas and Related Disciplines.

This report is available at www.swebok.org.

4. J. W. Moore, *Software Engineering Standards, A User's Road Map*, IEEE Computer Society Press, 1998.

This book describes the scope, roles, uses, and development trends of the most widely used software engineering standards. It concentrates on important software engineering activities — quality and project management, system engineering, dependability, and safety. The analysis and regrouping of the standard collections exposes you to key relationships between standards.

Even though the Guide to the Software Engineering Body of Knowledge is not a software engineering standards development project per se, special care will be taken throughout the project regarding the compatibility of the Guide with the current IEEE and ISO Software Engineering Standards Collection.

5. *IEEE Standard Glossary of Software Engineering Terminology*, std 610.12-1990, IEEE, 1990.

The hierarchy of references for terminology is *Merriam Webster's Collegiate Dictionary* (10th ed.), IEEE std 610.12, and new proposed definitions if required.

6. *Information Technology – Software Life Cycle Processes*, International std ISO/IEC 12207:1995(E), 1995.

This standard is considered the key standard regarding the definition of life cycle process and has been adopted by the two main standardization bodies in software engineering: ISO/IEC JTC1 SC7 and the IEEE Computer Society Software Engineering Standards Committee. It also has been designated as the pivotal standard around which the Software Engineering Standards Committee (SESC) is currently harmonizing its entire collection of standards. This standard was a key input to the Straw Man version.

Even though we do not intend that the Guide to the Software Engineering Body of Knowledge be fully 12207-compliant, this standard remains a key input to the Stone Man version and special care will be taken throughout the project regarding the compatibility of the Guide with the 12207 standard.

7. *Merriam Webster's Collegiate Dictionary* (10th ed.).

See note for std IEEE 610.12.

STYLE AND TECHNICAL GUIDELINES

Knowledge Area Descriptions should conform to the International Conference on Software Engineering Proceedings format (templates are available at http://sunset.usc.edu/icse99/cfp /technical_papers.html).

Knowledge Area Descriptions are expected to follow the IEEE Computer Society Style Guide. See http://www.computer.org/author/style/cs-style.htm.

Microsoft Word is the preferred submission format. Please contact the Editorial Team if this is not feasible for you.

OTHER DETAILED GUIDELINES

When referencing the guide, we recommend that you use the full title "Guide to the SWEBOK" instead of only "SWEBOK."

For the purpose of simplicity, we recommend that Knowledge Area Associate Editors avoid footnotes. Instead, they should try to include their content in the main text.

We recommend using in the text explicit references to standards, as opposed to simply inserting numbers referencing items in the bibliography. We believe it allows

the reader to be better exposed to the source and scope of a standard.

The text accompanying figures and tables should be self-explanatory or have enough related text. This would ensure that the reader knows what the figures and tables mean.

Make sure you use current information about references (versions, titles, etc.).

To make sure that some information contained in the Guide to the SWEBOK does not become rapidly obsolete, please avoid directly naming tools and products. Instead, try to name their functions. The list of tools and products can always be put in an appendix.

You are expected to spell out all acronyms used and to use all appropriate copyrights, service marks, etc.

The Knowledge Area Descriptions should always be written in third person.

EDITING

The Editorial Team and professional editors will edit Knowledge Area Descriptions. Editing includes copy editing (grammar, punctuation, and capitalization), style editing (conformance to the Computer Society magazines' house style), and content editing (flow, meaning, clarity, directness, and organization). The final editing will be a collaborative process in which the Editorial Team and the authors work together to achieve a concise, well-worded, and useful Knowledge Area Description.

RELEASE OF COPYRIGHT

All intellectual properties associated with the Guide to the Software Engineering Body of Knowledge will remain with the IEEE Computer Society. Knowledge Area Associate Editors were asked to sign a copyright release form.

It is also understood that the Guide to the Software Engineering Body of Knowledge will be put in the public domain by the IEEE Computer Society, free of charge through web technology or by other means.

For more information, see http://www.computer.org/copyright.htm.

APPENDIX B

EVOLUTION OF THE GUIDE TO THE SOFTWARE ENGINEERING BODY OF KNOWLEDGE

INTRODUCTION

Although the 2004 Guide to the Software Engineering Body of Knowledge is a milestone in reaching a broad agreement on the content of the software engineering discipline, it is not the end of the process. The 2004 Guide is simply the current edition of a guide that will continue evolving to meet the needs of the software engineering community. Planning for evolution is not yet complete, but a tentative outline of the process is provided in this section. As of this writing, this process has been endorsed by the project's Industrial Advisory Board and briefed to the Board of Governors of the IEEE Computer Society, but is not yet either funded or implemented.

STAKEHOLDERS

Widespread adoption of the SWEBOK Guide has produced a substantial community of stakeholders in addition to the Computer Society itself. There are a number of projects—both inside and outside the Computer Society—that are coordinating their content with the content of the SWEBOK Guide. (More about that in a moment.) Several corporations, including some of the members of the project's Industrial Advisory Board, have adopted the Guide for use in their internal programs for education and training. In a broader sense, the software engineering practitioner community, professional development community, and education community pay attention to the SWEBOK Guide to help define the scope of their efforts. A notable stakeholder group is the holders of the IEEE Computer Society's certification—Certified Software Development Professional—because the scope of the CSDP examination is largely aligned with the scope of the SWEBOK Guide.

The IEEE Computer Society and other organizations are now conducting a number of projects that depend on the evolution of the SWEBOK Guide:

- The CSDP examination, initially developed in parallel with the SWEBOK Guide, will evolve to a close match to the Guide—both in scope[1] and reference material.
- The Computer Society's Distance Learning curriculum for software engineers will have the same scope as the SWEBOK Guide. An initial overview course is already available.
- Although the goals of undergraduate education differ somewhat from those of professional development, the joint ACM/IEEE-CS project to develop an undergraduate software engineering curriculum is largely reconciled with the scope of the SWEBOK Guide.
- The IEEE-CS Software Engineering Standards Committee (SESC) has organized its collection by the Knowledge Areas of the SWEBOK Guide, and the IEEE Standards Association has already published a CD-ROM collected edition of software engineering standards that reflects that organization.
- ISO/IEC JTC1/SC7, the international standards organization for software and systems engineering, is adopting the SWEBOK Guide as ISO/IEC Technical Report 19759 and harmonizing its collection with that of IEEE.
- The IEEE Computer Society Press, in cooperation with SESC, is developing a book series based on software engineering standards and the SWEBOK Guide.
- The Computer Society's Software Engineering Portal ("SE Online"), currently in planning, will be organized by the Knowledge Areas of the SWEBOK Guide.
- The Trial Use Version of the SWEBOK Guide was translated into Japanese. It is anticipated that the 2004 Version will also be translated into Japanese, Chinese, and possibly other languages.

THE EVOLUTION PROCESS

Obviously, a product with this much uptake must be evolved in an open, consultative, deliberate, and transparent fashion so that other projects can successfully coordinate efforts. The currently planned strategy is to evolve the SWEBOK Guide using a "time-boxed" approach. The time-box approach is selected because it allows the SWEBOK Guide and coordinating projects to perform revision in anticipation of a fixed date for convergence. The initial time box is currently planned to be four years in duration.

At the beginning of the time box, in consultation with coordinating projects, an overall plan for the four-year revision would be determined. During the first year, structural changes to the SWEBOK Guide (e.g., changes in number or scope of Knowledge Areas) would be determined. During the second and third years, the selection and treatment of topics within the Knowledge Areas would be revised. During the fourth year, the text of the Knowledge Area descriptions would be revised and up-to-date references would be selected.

The overall project would be managed by a Computer Society committee of volunteers and representatives of

[1] The CSDP adds one Knowledge Area, Business Practices and Engineering Economics, to the ten Knowledge Areas covered by the SWEBOK Guide.

coordinating projects. The committee would be responsible to set overall plans, coordinate with stakeholders, and recommend approval of the final revision. The committee would be advised by a SWEBOK Advisory Committee (SWAC) composed of organizational adopters of the SWEBOK Guide. The SWAC would also be the focus for obtaining corporate financial support for the evolution of the SWEBOK Guide. Past corporate financial support has allowed us to make the SWEBOK Guide available for free on a Web site. Future support will allow us to continue the practice for future editions.

Notionally, each of the four years would include a cycle of workshop, drafting, balloting, and ballot resolution. A yearly cycle might involve the following activities:

- A workshop, organized as a part of a major conference, would specify issues for treatment during the coming year, prioritize the issues, recommend approaches for dealing with them, and nominate drafters to implement the approaches.
- Each drafter would write or modify a Knowledge Area description using the approach recommended by the workshop and available references. In the final year of the cycle, drafters would recommend specific up-to-date references for citation in the SWEBOK Guide. Drafters would also be responsible for modifying their drafts in response to comments from balloters.
- Each annual cycle would include balloting on the revisions to the Knowledge Area descriptions. Balloters would review the drafted Knowledge Area descriptions and the recommended references, provide comments, and vote approval on the revisions. Balloting would be open to members of the Computer Society and other qualified participants. (Nonmembers would have to pay a fee to defray the expense of balloting.) Holders of the CSDP would be particularly welcome as members of the balloting group or as volunteers in other roles.
- A Ballot Resolution Committee would be selected by the Managing Committee to serve as intermediaries between the drafters and the balloters. Its job is to determine consensus for changes requested by the balloting group and to ensure that the drafters implement the needed changes. In some cases, the Ballot Resolution Committee may phrase questions for the balloting group and use their answers to guide the revision of the draft. Each year's goal is to achieve consensus among the balloting group on the new and revised draft Knowledge Areas and to gain a vote of approval from the balloters. Although the SWEBOK Guide would not be changed until the end of the time box, the approved material from each year's cycle will be made freely available.

At the conclusion of the time box, the completed product, SWEBOK Guide 2008, would be reviewed and approved by the Computer Society Board of Governors for publication. If continuing corporate financial support can be obtained, the product would be made freely available on a Web site.

ANTICIPATED CHANGES

It is important to note that the SWEBOK Guide is inherently a conservative document for several reasons. First, it limits itself to knowledge characteristic of software engineering; so information from related disciplines—even disciplines applied by software engineers—is omitted. Second, it is developed and approved by a consensus process, so it can only record information for which broad agreement can be obtained. Third, knowledge regarded as specialized to specific domains is excluded. Finally and most importantly, the Guide records only the knowledge which is "generally accepted." Even current and valid techniques may need some time to gain general acceptance within the community.

This conservative approach is apparent in the current SWEBOK Guide. After six years of work, it still has the same ten Knowledge Areas. One might ask if that selection of Knowledge Areas will ever be changed. The plan for evolution includes some criteria for adding a Knowledge Area or changing the scope of a Knowledge Area. In principle, the candidate must be widely recognized inside and outside the software engineering community as representing a distinct area of knowledge and the generally accepted knowledge within the proposed area must be sufficiently detailed and complete to merit treatment similar to those currently in the SWEBOK Guide. In operational terms, it must be possible to cleanly decouple the proposed Knowledge Area from the existing ones, and that decoupling must add significant value to the overall taxonomy of knowledge provided by the Guide. However, simply being a "cross-cutting" topic is not justification for separate treatment because separation, in many cases, simply compounds the problem of topic overlap. In general, growth in the total number of Knowledge Areas is to be avoided when it complicates the efforts of readers to find desired information.

Adding a topic to a Knowledge Area is easier. In principle, it must be mature (or, at least, rapidly reaching maturity) and generally accepted.[2] Evidence for general acceptance can be found in many places, including software engineering curricula, software engineering standards, and widely used textbooks. Of course, topics must be suitable to the SWEBOK Guide's design point of a bachelor's degree plus four years of experience.[3]

---

[2] For the definition of "generally accepted," we use IEEE Std 1490-1998, Adoption of PMI Standard—A Guide to the Project Management Body of Knowledge: "Generally accepted means that the knowledge and practices described are applicable to most projects most of the time, and that there is widespread consensus about their value and usefulness. It does not mean that the knowledge and practices should be applied uniformly to all projects without considering whether they are appropriate."

[3] Of course, this particular specification is stated in terms relevant to the US. In other countries, it might be stated differently.

That design point raises the issue of the volume of material referenced by the SWEBOK Guide. The total amount of material should be consistent with the design point of a bachelor's degree plus four years of experience. Currently, the editorial team estimates an appropriate amount to be 5000 pages of textbook material. During the evolution of the Guide, it will be necessary to manage the lists of cited material so that references are currently accessible, provide appropriate coverage of the Knowledge Areas, and total to a reasonable amount of material.

A final topic is the role to be played by users of the SWEBOK Guide in its evolution. The Editorial Team believes that continual public comment is the fuel that will drive the evolution of the SWEBOK Guide. Public comments will raise issues for treatment by the annual workshop, hence setting the agenda for revision of the SWEBOK Guide. We hope to provide a public, online forum for comment by any member of the software engineering community and to serve as a focal point for adoption activities.

APPENDIX C

ALLOCATION OF IEEE AND ISO SOFTWARE ENGINEERING STANDARDS TO SWEBOK KNOWLEDGE AREAS

This table lists software engineering standards produced by IEEE and ISO/IEC JTC1/SC7, as well as a few selected standards from other sources. For each standard, its number and title is provided. In addition, the "Description" column provides material excerpted from the standard's abstract or other introductory material. Each of the standards is mapped to the Knowledge Areas of the SWEBOK Guide. In a few cases—like vocabulary standards—the standard applies equally to all KAs; in such cases, an X appears in every column. In most cases, each standard has a forced allocation to a single primary Knowledge Area; this allocation is marked with a "P". In many cases, there are secondary allocations to other KAs; these are marked with an "S". The list is ordered by standard number, regardless of its category (IEEE, ISO, etc.).

Standard Number	Standard Name	Description	Software Requirements	Software Design	Software Construction	Software Testing	Software Maintenance	Software Configuration Management	Software Engineering Management	Software Engineering Process	Software Engineering Tools and Methods	Software Quality
IEEE Std 610.12-1990 (R2002)	IEEE Standard Glossary of Software Engineering Terminology	This standard is a glossary of software engineering terminology.	X	X	X	X	X	X	X	X	X	X
IEEE Std 730-2002	IEEE Standard for Software Quality Assurance Plans	This standard specifies the format and content of software quality assurance plans.						S	S			P
IEEE Std 828-1998	IEEE Standard for Software Configuration Management Plans	This standard specifies the content of a software configuration management plan along with requirements for specific activities.						P	S			
IEEE Std 829-1998	IEEE Standard for Software Test Documentation	This standard describes the form and content of a basic set of documentation for planning, executing, and reporting software testing.			S	P						S
IEEE Std 830-1998	IEEE Recommended Practice for Software Requirements Specifications	This document recommends the content and characteristics of a software requirements specification. Sample outlines are provided.	P									
IEEE Std 982.1-1988	IEEE Standard Dictionary of Measures to Produce Reliable Software	This standard provides a set of measures for evaluating the reliability of a software product and for obtaining early forecasts of the reliability of a product under development.		S	S	S			S			P

Standard Number	Standard Name	Description	Software Requirements	Software Design	Software Construction	Software Testing	Software Maintenance	Software Configuration Management	Software Engineering Management	Software Engineering Process	Software Engineering Tools and Methods	Software Quality
IEEE Std 1008-1987 (R2003)	IEEE Standard for Software Unit Testing	This standard describes a sound approach to software unit testing and the concepts and assumptions on which it is based. It also provides guidance and resource information.			S	P				S		S
IEEE Std 1012-1998 and 1012a-1998	IEEE Standard for Software Verification and Validation	This standard describes software verification and validation processes that are used to determine if software products of an activity meet the requirements of the activity and to determine if software satisfies the user's needs for the intended usage. The scope includes analysis, evaluation, review, inspection, assessment, and testing of both products and processes.										P
IEEE Std 1016-1998	IEEE Recommended Practice for Software Design Descriptions	This document recommends content and organization of a software design description.		P								
IEEE Std 1028-1997 (R2002)	IEEE Standard for Software Reviews	This standard defines five types of software reviews and procedures for their execution. Review types include management reviews, technical reviews, inspections, walk-throughs, and audits.	S	S	S			S	S			P
IEEE Std 1044-1993 (R2002)	IEEE Standard Classification for Software Anomalies	This standard provides a uniform approach to the classification of anomalies found in software and its documentation. It includes helpful lists of anomaly classifications and related data.				S			S	S		P
IEEE Std 1045-1992 (R2002)	IEEE Standard for Software Productivity Metrics	This standard provides a consistent terminology for software productivity measures and defines a consistent way to measure the elements that go into computing software productivity.							P	S		
IEEE Std 1058-1998	IEEE Standard for Software Project Management Plans	This standard describes the format and contents of a software project management plan.							P			
IEEE Std 1061-1998	IEEE Standard for a Software Quality Metrics Methodology	This standard describes a methodology—spanning the entire life cycle—for establishing quality requirements and identifying, implementing, and validating the corresponding measures.			S		S		S	S		P
IEEE Std 1062, 1998 Edition	IEEE Recommended Practice for Software Acquisition	This document recommends a set of useful practices that can be selected and applied during software acquisition. It is primarily suited to acquisitions that include development or modification rather than off-the-shelf purchase.	S						P			

Standard Number	Standard Name	Description	Software Requirements	Software Design	Software Construction	Software Testing	Software Maintenance	Software Configuration Management	Software Engineering Management	Software Engineering Process	Software Engineering Tools and Methods	Software Quality
IEEE Std 1063-2001	IEEE Standard for Software User Documentation	This standard provides minimum requirements for the structure, content, and format of user documentation--both printed and electronic.			P							S
IEEE Std 1074-1997	IEEE Standard for Developing Software Life Cycle Processes	This standard describes an approach for the definition of software life cycle processes.								P		
IEEE Std 1175.1-2002	IEEE Guide for CASE Tool Interconnections — Classification and Description	This standard is the first of a planned series of standards on the integration of CASE tools into a productive software engineering environment. This part describes fundamental concepts and introduces the remaining (planned) parts.									P	
IEEE Std 1219-1998	IEEE Standard for Software Maintenance	This standard describes a process for software maintenance and its management.					P			S		
IEEE Std 1220-1998	IEEE Standard for the Application and Management of the Systems Engineering Process	This standard describes the systems engineering activities and process required throughout a system's life cycle to develop systems meeting customer needs, requirements, and constraints.								P		
IEEE Std 1228-1994	IEEE Standard for Software Safety Plans	This standard describes the minimum content of a plan for the software aspects of development, procurement, maintenance, and retirement of a safety-critical system.	S			S			S			P
IEEE Std 1233, 1998 Edition	IEEE Guide for Developing System Requirements Specifications	This document provides guidance on the development of a system requirements specification, covering the identification, organization, presentation, and modification of requirements. It also provides guidance on the characteristics and qualities of requirements.	**P**									
IEEE Std 1320.1-1998	IEEE Standard for Functional Modeling Language — Syntax and Semantics for IDEF0	This standard defines the IDEF0 modeling language used to represent decisions, actions, and activities of an organization or system. IDEF0 may be used to define requirements in terms of functions to be performed by a desired system.	S	S						S	P	

Standard Number	Standard Name	Description	Software Requirements	Software Design	Software Construction	Software Testing	Software Maintenance	Software Configuration Management	Software Engineering Management	Software Engineering Process	Software Engineering Tools and Methods	Software Quality
IEEE Std 1320.2-1998	IEEE Standard for Conceptual Modeling — Language Syntax and Semantics for IDEF1X 97 (IDEFobject)	This standard defines two conceptual modeling languages, collectively called IDEF1X97 (IDEFObject). The languages support the implementation of relational databases, object databases, and object models.	S	S							P	
IEEE Std 1362-1998	IEEE Guide for Information Technology — System Definition — Concept of Operations (ConOps) Document	This document provides guidance on the format and content of a Concept of Operations (ConOps) document, describing characteristics of a proposed system from the users' viewpoint.	**P**									
IEEE Std 1420.1-1995, 1420.1a-1996, and 1420.1b-1999 (R2002)	IEEE Standard for Information Technology — Software Reuse — Data Model for Reuse Library Interoperability	This standard and its supplements describe information that software reuse libraries should be able to exchange in order to interchange assets.									P	
IEEE Std 1462-1998 // ISO/IEC 14102:1995	IEEE Standard — Adoption of International Standard ISO/IEC 14102: 1995 — Information Technology — Guideline for the Evaluation and Selection of CASE tools	This standard provides guidelines for the evaluation and selection of CASE tools.									P	

Standard Number	Standard Name	Description	Software Requirements	Software Design	Software Construction	Software Testing	Software Maintenance	Software Configuration Management	Software Engineering Management	Software Engineering Process	Software Engineering Tools and Methods	Software Quality
IEEE Std 1465-1998 // ISO/IEC 12119	IEEE Standard, Adoption of International Standard ISO/IEC 12119:1994(E), Information Technology — Software packages — Quality requirements and testing	This standard describes quality requirements specifically suitable for software packages and guidance on testing the package against those requirements.	S									P
IEEE Std 1471-2000	IEEE Recommended Practice for Architectural Description of Software Intensive Systems	This document recommends a conceptual framework and content for the architectural description of software-intensive systems.	S	S							P	
IEEE Std 1490-1998	IEEE Guide — Adoption of PMI Standard — A Guide to the Project Management Body of Knowledge	This document is the IEEE adoption of a Project Management Body of Knowledge defined by the Project Management Institute. It identifies and describes generally accepted knowledge regarding project management.							P			
IEEE Std 1517-1999	IEEE Standard for Information Technology — Software Life Cycle Processes — Reuse Processes	This standard provides life cycle processes for systematic software reuse. The processes are suitable for use with IEEE/EIA 12207.			S					P		
IEEE Std 1540-2001 // ISO/IEC 16085:2003	IEEE Standard for Software Life Cycle Processes — Risk Management	This standard provides a life cycle process for software risk management. The process is suitable for use with IEEE/EIA 12207.							S	P		

Standard Number	Standard Name	Description	Software Requirements	Software Design	Software Construction	Software Testing	Software Maintenance	Software Configuration Management	Software Engineering Management	Software Engineering Process	Software Engineering Tools and Methods	Software Quality
IEEE Std 2001-2002	IEEE Recommended Practice for the Internet — Web Site Engineering, Web Site Management, and Web Site Life Cycle	This document recommends practices for engineering World Wide Web pages for use in intranet and extranet environments.										P
ISO 9001:2000	Quality Management Systems — Requirements	This standard specifies the requirements for an organizational quality management system aiming to provide products meeting requirements and enhance customer satisfaction.								S		P
ISO/IEC 9126-1:2001	Software Engineering — Product Quality — Part 1: Quality Model	This standard provides a model for software product quality covering internal quality, external quality, and quality in use. The model is in the form of a taxonomy of defined characteristics which software may exhibit.	P	S	S	S						
IEEE/EIA 12207.0-1996 // ISO/IEC 12207:1995	Industry Implementation of International Standard ISO/IEC 12207:1995, Standard for Information Technology — Software Life Cycle Processes	This standard provides a framework of processes used across the entire life cycle of software.	X	X	X	X	X	X	X	P	X	X
IEEE/EIA 12207.1-1996	Industry Implementation of International Standard ISO/IEC 12207:1995, Standard for Information Technology — Software Life Cycle Processes — Life Cycle Data	This document provides guidance on recording data resulting from the life cycle processes of IEEE/EIA 12207.0.	X	X	X	X	X	X	X	P	X	X

Standard Number	Standard Name	Description	Software Requirements	Software Design	Software Construction	Software Testing	Software Maintenance	Software Configuration Management	Software Engineering Management	Software Engineering Process	Software Engineering Tools and Methods	Software Quality
IEEE/EIA 12207.2-1997	Industry Implementation of International Standard ISO/IEC 12207:1995, Standard for Information Technology — Software Life Cycle Processes — Implementation Considerations	This document provides additional guidance for the implementation of the life cycle processes of IEEE/EIA 12207.0.	X	X	X	X	X	X	X	P	X	X
IEEE Std 14143.1-2000 // ISO/IEC 14143-1:1998	IEEE Adoption of ISO/IEC 14143-1:1998 — Information Technology— Software Measurement — Functional Size Measurement — Part 1: Definition of Concepts	This standard describes the fundamental concepts of a class of measures collectively known as functional size.	P				S		S			S
ISO/IEC TR 14471:1999	Information technology — Software engineering — Guidelines for the adoption of CASE tools	This document provides guidance in establishing processes and activities that may be applied in the adoption of CASE technology.									P	
ISO/IEC 14598 (Six parts)	Information technology — Software product evaluation	The ISO/IEC 14598 series gives an overview of software product evaluation processes and provides guidance and requirements for evaluation at the organizational or project level. The standards provide methods for measurement, assessment and evaluation.										P
ISO/IEC 14764:1999	Information Technology — Software Maintenance	This standard elaborates on the maintenance process provided in ISO/IEC 12207. It provides guidance in implementing the requirements of that process.					P					

Standard Number	Standard Name	Description	Software Requirements	Software Design	Software Construction	Software Testing	Software Maintenance	Software Configuration Management	Software Engineering Management	Software Engineering Process	Software Engineering Tools and Methods	Software Quality
ISO/IEC 15026:1998	Information Technology — System and Software Integrity Levels	This International Standard introduces the concepts of software integrity levels and software integrity requirements. It defines the concepts associated with integrity levels, defines the processes for determining integrity levels and software integrity requirements, and places requirements on each process.	S	S								P
ISO/IEC TR 15271:1998	Information technology — Guide for ISO/IEC 12207 (Software Life Cycle Processes)	This document is a guide to the use of ISO/IEC 12207.								P		
ISO/IEC 15288:2002	Systems Engineering — System Life Cycle Processes	This standard provides a framework of processes used across the entire life cycle of human-made systems.								P		
ISO/IEC TR 15504 (9 parts) and Draft IS 15504 (5 parts)	Software Engineering — Process Assessment	This technical report (now being revised as a standard) provides requirements on methods for performing process assessment as a basis for process improvement or capability determination.								P		
ISO/IEC 15939:2002	Software Engineering — Software Measurement Process	This standard provides a life cycle process for software measurement. The process is suitable for use with IEEE/EIA 12207.							S	P		S
ISO/IEC 19761:2003	Software engineering — COSMIC-FFP — A functional size measurement method	This standard describes the COSMIC-FFP Functional Size Measurement Method, a functional size measurement method conforming to the requirements of ISO/IEC 14143-1.	P				S		S			S
ISO/IEC 20926:2003	Software engineering — IFPUG 4.1 Unadjusted functional size measurement method — Counting practices manual	This standard describes IFPUG 4.1 Unadjusted Function Point Counting, a functional size measurement method conforming to the requirements of ISO/IEC 14143-1.	P				S		S			S

Standard Number	Standard Name	Description	Software Requirements	Software Design	Software Construction	Software Testing	Software Maintenance	Software Configuration Management	Software Engineering Management	Software Engineering Process	Software Engineering Tools and Methods	Software Quality
ISO/IEC 20968:2002	Software engineering — Mk II Function Point Analysis — Counting Practices Manual	This standard describes Mk II Function Point Analysis, a functional size measurement method conforming to the requirements of ISO/IEC 14143-1.	P				S		S			S
ISO/IEC 90003	Software and Systems Engineering — Guidelines for the Application of ISO 9001:2000 to Computer Software	This standard provides guidance for organizations in the application of ISO 9001:2000 to the acquisition, supply, development, operation, and maintenance of computer software.								S		P

APPENDIX D

CLASSIFICATION OF TOPICS ACCORDING TO BLOOM'S TAXONOMY

INTRODUCTION

Bloom's taxonomy[1] is a well-known and widely used classification of cognitive educational goals. In order to help audiences who wish to use the Guide as a tool in defining course material, university curricula, university program accreditation criteria, job descriptions, role descriptions within a software engineering process definition, professional development paths and professional training programs, and other needs, Bloom's taxonomy levels for SWEBOK Guide topics are proposed in this appendix for a software engineering graduate with four years of experience. A software engineering graduate with four years of experience is in essence the "target" of the SWEBOK Guide as defined by what is meant by generally accepted knowledge (See Introduction of the SWEBOK Guide).

Since this Appendix only pertains to what can be considered as "generally accepted" knowledge, it is very important to remember that a software engineer must know substantially more than this "category" of knowledge. In addition to "generally accepted" knowledge, a software engineering graduate with four years of knowledge must possess some elements from the Related Disciplines as well as certain elements of specialized knowledge, advanced knowledge, and possibly even research knowledge (see Introduction of the SWEBOK Guide).

The following assumptions were made when specifying the proposed taxonomy levels:

- The evaluations are proposed for a "generalist" software engineer and not a software engineer working in a specialized group such as a software configuration management team, for instance. Obviously, such a software engineer would require or would attain much higher taxonomy levels in the specialty area of their group;
- A software engineer with four years of experience is still at the beginning of their career and would be assigned relatively few management duties, or at least not for major endeavors. "Management-related topics" are therefore not given priority in the proposed evaluations. For the same reason, taxonomy levels tend to be lower for "early life cycle topics" such as those related to software requirements than for more technically-oriented topics such as those within software design, software construction or software testing.
- So the evaluations can be adapted for more senior software engineers or software engineers specializing in certain knowledge areas, no topic is given a taxonomy level higher than Analysis. This is consistent with the approach taken in the Software Engineering Education Body of Knowledge (SEEK) where no topic is assigned a taxonomy level higher than Application.[2] The purpose of SEEK is to define a software engineering education body of knowledge appropriate for guiding the development of undergraduate software engineering curricula. Though distinct notably in terms of scope, SEEK and the SWEBOK Guide are closely related.[3]

Bloom's Taxonomy of the Cognitive Domain proposed in 1956 contains six levels. Table 1[4] presents these levels and keywords often associated with each level.

[1] B. Bloom (ed.), *Taxonomy of Educational Objectives: The Classification of Educational Goals*, Mackay, 1956.

[2] See Joint Task Force on Computing Curricula – IEEE Computer Society / Association for Computing Machinery, Computing Curricula – Software Engineering Volume – Public Draft 1 – Computing Curriculum Software Engineering, 2003; http://sites.computer.org/ccse/.

[3] See P Bourque, F. Robert, J.-M. Lavoie, A. Lee, S. Trudel, T. Lethbridge, "Guide to the Software Engineering Body of Knowledge (SWEBOK) and the Software Engineering Education Body of Knowledge (SEEK) – A Preliminary Mapping," in *Proc. 10th Intern. Workshop Software Technology and Engineering Practice Conference* (STEP 2002), 2002, pp. 8-35.

[4] Table adapted from http://www.nwlink.com/~donclark/hrd/bloom.html.

Table 1 Bloom's Taxonomy

Bloom's Taxonomy Level	Associated Keywords
Knowledge: Recall data	Defines, describes, identifies, knows, labels, lists, matches, names, outlines, recalls, recognizes, reproduces, selects, states
Comprehension: Understand the meaning, translation, interpolation, and interpretation of instructions and problems; state a problem in one's own words.	Comprehends, converts, defends, distinguishes, estimates, explains, extends, generalizes, gives examples, infers, interprets, paraphrases, predicts, rewrites, summarizes, translates
Application: Use a concept in a new situation or use an abstraction unprompted; apply what was learned in the classroom to novel situations in the workplace	Applies, changes, computes, constructs, demonstrates, discovers, manipulates, modifies, operates, predicts, prepares, produces, relates, shows, solves, uses
Analysis: Separate material or concepts into component parts so that its organizational structure may be understood; distinguish between facts and inferences	Analyzes, breaks down, compares, contrasts, diagrams, deconstructs, differentiates, discriminates, distinguishes, identifies, illustrates, infers, outlines, relates, selects, separates
Synthesis: Build a structure or pattern from diverse elements; put parts together to form a whole, with emphasis on creating a new meaning or structure	Categorizes, combines, compiles, composes, creates, devises, designs, explains, generates, modifies, organizes, plans, rearranges, reconstructs, relates, reorganizes, revises, rewrites, summarizes, tells, writes
Evaluation: Make judgments about the value of ideas or materials	Appraises, compares, concludes, contrasts, criticizes, critiques, defends, describes, discriminates, evaluates, explains, interprets, justifies, relates, summarizes, supports

The breakdown of topics in the tables does not match perfectly the breakdown in the Knowledge Areas. The evaluation for this Appendix was prepared while some comments were still coming in.

Finally, please bear in mind that the evaluations of this Appendix should definitely only be seen as a proposal to be further developed and validated.

SOFTWARE REQUIREMENTS[5]

Breakdown of Topics	Taxonomy Level
1. Software requirements fundamentals	
Definition of software requirement	C
Product and process requirements	C
Functional and non-functional requirements	C
Emergent properties	C
Quantifiable requirements	C
System requirements and software requirements	C
2. Requirements process	
Process models	C
Process actors	C
Process support and management	C
Process quality and improvement	C
3. Requirements elicitation	
Requirements sources	C
Elicitation techniques	AP
4. Requirements analysis	
Requirements classification	AP
Conceptual modeling	AN
Architectural design and requirements allocation	AN
Requirements negotiation	AP
5. Requirements specification	
System definition document	C
System requirements specification	C
Software requirements specification	AP
6. Requirements validation	
Requirements reviews	AP
Prototyping	AP
Model validation	C
Acceptance tests	AP
7. Practical considerations	
Iterative nature of requirements process	C
Change management	AP
Requirements attributes	C
Requirements tracing	AP
Measuring requirements	AP

[5] K: Knowledge, C: Comprehension, AP: Application, AN: Analysis, E: Evaluation, S: Synthesis

SOFTWARE DESIGN

Breakdown of Topics	Taxonomy Level
1. Software design fundamentals	
General design concepts	C
Context of software design	C
Software design process	C
Enabling techniques	AN
2. Key issues in software design	
Concurrency	AP
Control and handling of events	AP
Distribution of components	AP
Error and exception handling and fault tolerance	AP
Interaction and presentation	AP
Data persistence	AP
3. Software structure and architecture	
Architectural structures and viewpoints	AP
Architectural styles (macroarchitectural patterns)	AN
Design patterns (microarchitectural patterns)	AN
Families of programs and frameworks	C
4. Software design quality analysis and evaluation	
Quality attributes	C
Quality analysis and evaluation techniques	AN
Measures	C
5. Software design notations	
Structural descriptions (static)	AP
Behavioral descriptions (dynamic)	AP
6. Software design strategies and methods	
General strategies	AN
Function-oriented (structured) design	AP
Object-oriented design	AN
Data-structure centered design	C
Component-based design (CBD)	C
Other methods	C

SOFTWARE CONSTRUCTION

Breakdown of Topics	Taxonomy Level
1. Software construction fundamentals	
Minimizing complexity	AN
Anticipating change	AN
Constructing for verification	AN
Standards in construction	AP
2. Managing construction	
Construction methods	C
Construction planning	AP
Construction measurement	AP
3. Practical considerations	
Construction design	AN
Construction languages	AP
Coding	AN
Construction testing	AP
Construction quality	AN
Integration	AP

SOFTWARE TESTING

Breakdown of Topics	Taxonomy Level
1. Software testing fundamentals	
Testing-related terminology	C
Key issues	AP
Relationships of testing to other activities	C
2. Test levels	
The target of the tests	AP
Objectives of testing	AP
3. Test techniques	
Based on tester's intuition and experience	AP
Specification-based	AP
Code-based	AP
Fault-based	AP
Usage-based	AP
Based on nature of application	AP
Selecting and combining techniques	AP
4. Test-related measures	
Evaluation of the program under test	AN
Evaluation of the tests performed	AN
5. Test process	
Management concerns	C
Test activities	AP

SOFTWARE MAINTENANCE

	Taxonomy Level
1. Software maintenance fundamentals	
Definitions and terminology	C
Nature of maintenance	C
Need for maintenance	C
Majority of maintenance costs	C
Evolution of software	C
Categories of maintenance	AP
2. Key issues in software maintenance	
Technical	
Limited understanding	C
Testing	AP
Impact analysis	AN
Maintainability	AN
Management issues	
Alignment with organizational issues	C
Staffing	C
Process issues	C
Organizational	C
Maintenance cost estimation	
Cost estimation	AP
Parametric models	C
Experience	AP
Software maintenance measurement	AP
3. Maintenance process	
Maintenance process models	C
Maintenance activities	
Unique activities	AP
Supporting activities	AP
4. Techniques for maintenance	
Program comprehension	AN
Reengineering	C
Reverse engineering	C

SOFTWARE CONFIGURATION MANAGEMENT

Breakdown of Topics	Taxonomy Level
1. Management of the SCM process	
Organizational context for SCM	C
Constraints and guidance for SCM	C
Planning for SCM	
SCM organization and responsibilities	AP
SCM resources and schedules	AP
Tool selection and implementation	AP
Vendor/subcontractor control	C
Interface control	C
Software configuration management plan	C
Surveillance of software configuration management	
SCM measures and measurement	AP
In-process audits of SCM	C
2. Software configuration identification	
Identifying items to be controlled	
Software configuration	AP
Software configuration items	AP
Software configuration item relationships	AP
Software versions	AP
Baseline	AP
Acquiring software configuration items	AP
Software library	C
3. Software configuration control	
Requesting, evaluating and approving software changes	
Software configuration control board	AP
Software change request process	AP
Implementing software changes	AP
Deviations & waivers	C
4. Software configuration status accounting	
Software configuration status information	C
Software configuration status reporting	AP
5. Software configuration auditing	
Software functional configuration audit	C
Software physical configuration audit	C
In-Process audits of a software baseline	C
6. Software release management and delivery	
Software building	AP
Software release management	C

SOFTWARE ENGINEERING MANAGEMENT

	Taxonomy Level
1. Initiation and scope definition	
Determination and negotiation of requirements	AP
Feasibility analysis	AP
Process for requirements review/revision	C
2. Software project planning	
Process planning	C
Determine deliverables	AP
Effort, schedule, and cost estimation	AP
Resource allocation	AP
Risk management	AP
Quality management	AP
Plan management	C
3. Software project enactment	
Implementation of plans	AP
Supplier contract management	C
Implementation of measurement process	AP
Monitor process	AN
Control process	AP
Reporting	AP
4. Review and evaluation	
Determining satisfaction of requirements	AP
Reviewing and evaluating performance	AP
5. Closure	
Determining closure	AP
Closure activities	AP
6. Software engineering measurement	
Establish and sustain measurement commitment	C
Plan the measurement process	C
Perform the measurement process	C
Evaluate measurement	C

SOFTWARE ENGINEERING PROCESS

	Taxonomy Level
1. Process implementation and change	
Process infrastructure	
Software engineering process group	C
Experience factory	C
Activities	AP
Models for process implementation and change	K
Practical considerations	C
2. Process definition	
Life cycle models	AP
Software life cycle processes	C
Notations for process definitions	C
Process adaptation	C
Automation	C
3. Process assessment	
Process assessment models	C
Process assessment methods	C
4. Product and process measurement	
Software process measurement	AP
Software product measurement	AP
Size measurement	AP
Structure measurement	AP
Quality measurement	AP
Quality of measurement results	AN
Software information models	
Model building	AP
Model implementation	AP
Measurement techniques	
Analytical techniques	AP
Benchmarking techniques	C

SOFTWARE ENGINEERING TOOLS AND METHODS

Breakdown of Topics	Taxonomy Level
1. Software tools	
Software requirements tools	AP
Software design tools	AP
Software construction tools	AP
Software testing tools	AP
Software maintenance tools	AP
Software engineering process tools	AP
Software quality tools	AP
Software configuration management tools	AP
Software engineering management tools	AP
Miscellaneous tool issues	AP
2. Software engineering methods	
Heuristic methods	AP
Formal methods and notations	C
Prototyping methods	AP
Miscellaneous method issues	C

SOFTWARE QUALITY

	Taxonomy Level
1. Software quality fundamentals	
Software engineering culture and ethics	AN
Value and costs of quality	AN
Quality models and characteristics	
Software process quality	AN
Software product quality	AN
Quality improvement	AP
2. Software quality management processes	
Software quality assurance	AP
Verification and validation	AP
Reviews and audits	
Inspections	AP
Peer reviews	AP
Walkthroughs	AP
Testing	AP
Audits	C
3. Practical considerations	
Application quality requirements	
Criticality of systems	C
Dependability	C
Integrity levels of software	C
Defect characterization	AP
Software quality management techniques	
Static techniques	AP
People-intensive techniques	AP
Analytic techniques	AP
Dynamic techniques	AP
Software quality measurement	AP