

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]

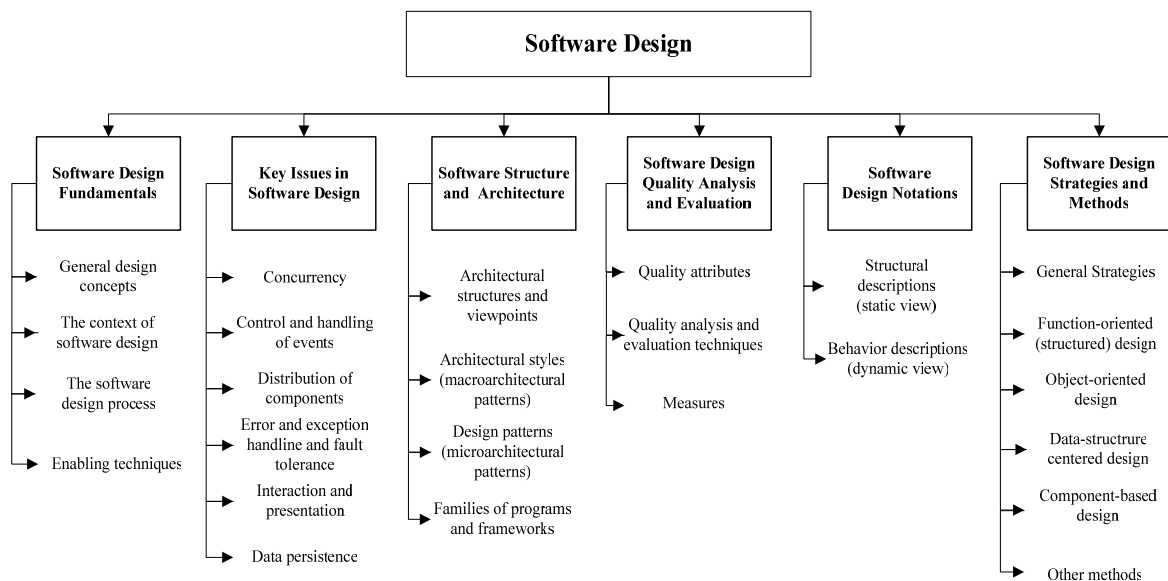


Figure 1 Breakdown of topics for the Software Design KA

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 *macroarchitecture*), other design patterns can be used to describe details at a lower, more local level (their *microarchitecture*). [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]

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 15026-1:1998 *Information Technology — System and Software Integrity Levels*, ISO and IEC, 1998.