CHAPTER 3
SOFTWARE DESIGN

Guy Tremblay
Département d'informatique
Université du Québec à Montréal
C.P. 8888, Succ. Centre-Ville
Montréal, Québec, Canada, H3C 3P8
tremblay.guy@uqam.ca

Table of Contents

1. INTRODUCTION

This chapter presents a description of the Software Design knowledge area for the Guide to the SWEBOK (Stone Man version). First, a general definition of the knowledge area is given. A breakdown of topics is then presented for the knowledge area along with brief descriptions of the various topics. These topic descriptions are also accompanied by references to material that provide more detailed presentation and coverage of these topics. The recommended references are then briefly described, followed by a number of suggestions for further readings.

It is important to stress that various constraints had to be satisfied by the resulting Knowledge Area (KA) description to satisfy the requirements set forth for these descriptions (see Appendix A of the whole Guide to the SWEBOK). Among the major constraints were that the KA description had to describe "generally accepted" knowledge not specific to any application domains or development methods and had to be compatible with typical breakdowns found in the literature. For those interested, Section 4 presents a more detailed Breakdown Rationale explaining how the various requirements for the KA description were met. A final note concerning the requirements was that the KA description had to suggest a list of "Recommended references" with a reasonably *limited* number of entries. Satisfying this requirement meant, sadly, that not all interesting references could be included in the recommended references list, thus the list of further readings.

2. DEFINITION OF SOFTWARE DESIGN

According to the IEEE definition [IEE90], 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". Viewed as a process, software design is the activity, within the software development life cycle, where software requirements are analyzed in order to produce a description of the internal structure and organization of the system that will serve as the basis for its construction. More precisely, a software design (the result) must describe the architecture of the system, that is, how the system is decomposed and organized into components and must describe the interfaces between these components. It must also describe these components into a level of detail suitable for allowing their construction.

In a classical software development life cycle such as ISO/IEC 12207 Software life cycle processes [ISO95b], software design consist of two activities that fit between software requirements analysis and software coding and testing: i) software architectural design – sometimes called top-level design, where the top-level structure and organization of the system is described and the various components are identified; ii) software detailed design – where each component is sufficiently described to allow for its coding.

Software design plays an important role in the development of a software system in that it allows the developer to produce various models that form a kind of blueprint of the solution to be implemented. These models can be analyzed and evaluated to determine if they will allow the various requirements to be fulfilled. Various alternative solutions and trade-offs can also be examined and evaluated. Finally, the resulting models can be used to plan the subsequent development activities, in addition to being used as input and starting point of the coding and testing activities.

Concerning the scope of the Software Design KA, it is important to note that not all topics containing the word "design" in their names will be discussed in the present KA description. In the terminology of DeMarco [DeM99], the present KA is concerned mainly with D-design (Decomposition design), as discussed in the above paragraphs (mapping a system into component pieces). However, because of its importance within the growing field of Software Architecture, FP-design (Family Pattern design, whose goal is to establish exploitable commonalities over a family of systems) will also be addressed. On the other hand, I-design (Invention design, usually done by system analysts with the objective of conceptualizing and specifying a system to satisfy discovered needs and requirements) will not be addressed, since this latter topic should be considered part of the requirements analysis and specification activity. Finally, also note that because of the requirements that the KA description had to include knowledge not specific to any application domains and the fact that some topics are better addressed in knowledge areas of related disciplines (see Appendix D of the whole Guide), certain specialized areas – for example, User Interface Design or Real-time Design – are not explicitly discussed in the present Software Design KA description. See Section 4 of the present chapter for further details concerning these and other specialized "design" topics. Of course, many of the topics included in the present Software Design KA description may still apply to these specialized areas.

3. BREAKDOWN OF TOPICS FOR SOFTWARE DESIGN

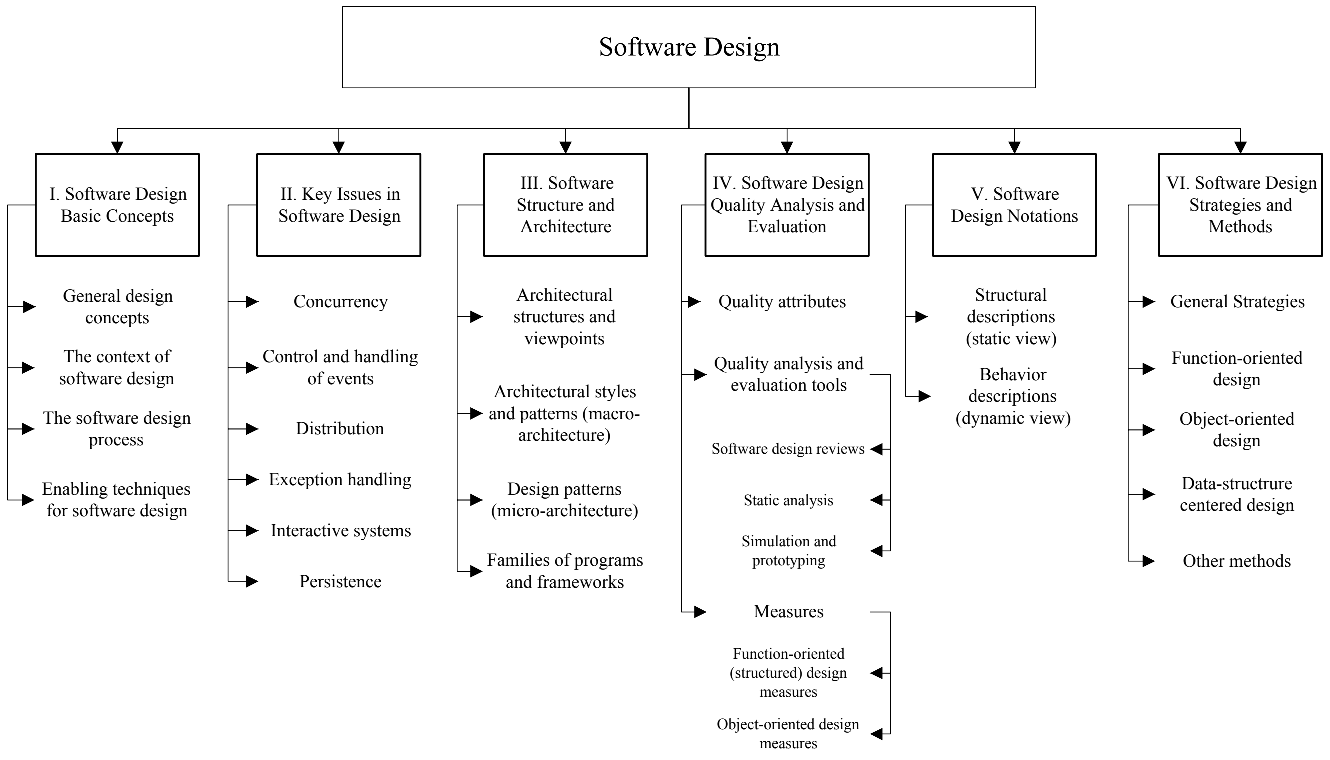
This section presents the breakdown of the Software Design Knowledge Area together with brief descriptions of each of the major topics. Appropriate references are also given for each of the topic. Figure 1 gives a graphical presentation of the top-level decomposition of the breakdown for the Software Design Knowledge Area. The detailed breakdown is presented in the following pages.

Note: The numbers in the reference keys, e.g., [Bud94:8, Pre97:23], indicate specific chapter(s) of the reference. In the case of Mar94, e.g., [Mar94:D], the letters indicates specific entries of the encyclopedia: "D" = Design; "DR" = Design Representation; "DD" = Design of Distributed systems". Note also that, contrary to the matrix presented in Section 5, only the appropriate chapter (or part) number, not the specific sections or pages, have been indicated.

I. Software Design Basic Concepts

This first section introduces a number of concepts and notions which form an underlying basis to the understanding of the role and scope of Software Design.

- General design concepts: Software is not the only field where design is involved. In the general sense, design can be seen as a form of problem-solving [Bud94:1]. For example, the notion of *wicked* problem – a problem that has no definitive solution – is interesting for understanding the limits of design [Bud94:1]. A number of notions and concepts are also interesting to understand design in its general sense: goals, constraints, alternatives, representations, and solutions [SB93].
- The context of software design: To understand the role and place of software design, it is important to understand the context in which software design fits, i.e., the software development life cycle. Thus, the major characteristics of software requirements analysis vs. software design vs. software construction vs. testing must be understood [ISO95b, LG01:11, Mar94:D, Pfl98:2, Pre97:2].
- The software design process: Software design is generally considered a two steps process: architectural design describes how the system is decomposed and organized into components (the software architecture), whereas detailed design describes the specific behavior of these components [DT97:7, FW83:I, ISO95b, LG01:13, Mar94:D]. The output of this process is a set of models and artifacts that record the major decisions that have been taken [Bud94:2, IEE98, LG01:13, Pre97:13].
- Enabling techniques for software design: According to the Oxford dictionary, a principle is "a basic truth or a general law […] that is used as a basis of reasoning or a guide to action". Such principles for software design, called *enabling techniques* in [BMR+96], are key notions considered fundamental to many different software design approaches, concepts and notions that form a kind of foundation for many of those approaches. Some of the key notions are the following [BCK98:6, BMR+96:6, IEE98, Jal97:5,6, LG01:1,3, Pfl98: 5, Pre97:13,23]:
  - Abstraction: "the process of forgetting information so that things that are different can be treated as if they are the same" [LG01]. In the context of software design, two key abstraction mechanisms are abstraction by parameterization and by specification, which in turn lead to three major kinds of abstraction: procedural abstraction, data abstraction and control (iteration) abstraction [BCK98:6, LG01:1,3,5,6 Jal97:5, Pre97:13].
  - Coupling and cohesion: whereas coupling measures the strength of the relationships that exist *between* modules, cohesion measures how the elements making up a module are related [BCK98:6, Jal97:5, Pfl98:5, Pre97:13].
  - Decomposition and modularization: the operation of decomposing a large system into a number of smaller independent ones, usually with the goal of placing different functionalities or responsibilities in different components [BCK98:6, BMR+96:6, Jal97:5, Pfl98:5, Pre97:13].



Software Design
I. Software Design Basic Concepts
General design concepts
The context of software design
The software design process
Enabling techniques for software design
II. Key Issues in Software Design
Concurrency
Control and handling of events
Distribution
Exception handling
Interactive systems
Persistence
III. Software Structure and Architecture
Architectural structures and viewpoints
Architectural styles and patterns (macro-architecture)
Design patterns (micro-architecture)
Families of programs and frameworks
IV. Software Design Quality Analysis and Evaluation
Quality attributes
Quality analysis and evaluation tools
Software design reviews
Static analysis
Simulation and prototyping
Measures
Function-oriented (structured) design measures
Object-oriented design measures
V. Software Design Notations
Structural descriptions (static view)
Behavior descriptions (dynamic view)
VI. Software Design Strategies and Methods
General Strategies
Function-oriented design
Object-oriented design
Data-structure centered design
Other methods

Figure 1 Breakdown of the Software Design KA

- Encapsulation/information hiding: deals with grouping and packaging the elements and internal details of an abstraction and making those details inaccessible [BCK98:6, BMR+96:6, Jal97:6, Pfl98:5, Pre97:13, 23].
- Separation of 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 [BCK98:6, Bos00:10, LG01:1,9].
- Sufficiency, completeness and primitiveness: deals with ensuring that a software component captures all the important characteristics of an abstraction, and nothing more [BMR+96:6, LG01:5].

II. Key Issues in Software Design

A number of key issues must be dealt with when designing software systems. Some of these are really quality concerns that must be addressed by all systems, for example, performance. Another important issue is how to decompose, organize and package the software components. This is so fundamental that it must be addressed, in one way or another, by all approaches to design; this is discussed in the Enabling techniques and in the Software Design Strategies topics. On the other hand, there are also other issues that "deal with some aspect of the system's behaviour 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 the system's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways" [KLM+97]. A number of these major, cross-cutting issues are the following (presented in alphabetical order):

- Concurrency: how to decompose the systems into processes, tasks and threads and deal with related efficiency, atomicity, synchronization and scheduling issues [Bos00:5, Mar94:DD, Mey97:30, Pre97:21].
- Control and handling of events: how to organize the flow of data and the flow of control, how to handle reactive and temporal events through various mechanisms, e.g., implicit invocation and call-backs [BCK98:5, Mey97:32, Pfl98:5].
- Distribution: how the software is distributed on the hardware, how the components communicate, how middleware can be used to deal with heterogeneous systems [BCK98:8, BMR+96:2, Bos00:5, Mar94:DD, Mey97:30, Pre97:28].

- Error and exception handling and fault tolerance: how to prevent and tolerate faults and deal with exceptional conditions [LG01:4, Mey97:12, Pfl98:5].
- Interactive systems: which approach to use to interact with users [BCK98:6, BMR+96:2.4, Bos00:5, LG01:13, Mey97:32].
  (Note: this topic is *not* about the specifications of the details of the user interface, which would be considered the task of the UI design, a topic beyond the scope of the current KA.)
- Persistence: how long-lived data is to be handled [Bos00:5, Mey97:31].

III. 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" [BMR+96:6]. An architecture thus attempts to define the internal *structure* – "the way in which something is constructed or organized" (Oxford dictionary) – of the resulting software. During the mid-90s, however, Software Architecture started to emerge as a broader discipline involved with studying software structures and architectures in a more generic way [SG96]. This gave rise to a number of interesting notions involved with the design of software at different levels of abstraction. Some of these notions can be useful during the architectural design (e.g., architectural style) as well as during the detailed design (e.g., lower-level design patterns) of a *specific* software system. But they can also be useful for designing *generic* systems, leading to the design of families of systems (aka. product lines). Interestingly, most of these notions can be seen as attempts to describe, and thus reuse, generic design knowledge.

- 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" [BMR+96]. These different views pertain to different issues associated with the design of software, 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, e.g., behavioral vs. functional vs. structural vs. data modeling views. The key idea is that a software design is a multi-faceted artifact produced by the design process and generally composed of relatively independent and orthogonal views [BCK98:2, BMR+96:6, BRJ99:31, Bud94:5, IEE98].
- Architectural styles (macro-architectural patterns): An architectural style is "a set of constraints on an architecture [that] define a set or family of architectures that satisfy them" [BCK98:2]. An architectural style can thus be seen as a meta-model that can provide the high-level organization (the *macro*-architecture) of a software system. A number of major styles have been identified by various authors. These styles can (tentatively) be organized as follows [BCK98:5, BMR+96:1,6, Bos00:6, BRJ99:28, Pfl98:5]:
  - General structure (e.g., layers, pipes and filters, blackboard);
  - Distributed systems (e.g., client-server, three-tiers, broker);
  - Interactive systems (e.g., Model-View-Controller, Presentation-Abstraction-Control);
  - Adaptable systems (e.g., micro-kernel, reflection);
  - Other styles (e.g., batch, interpreters, process control, rule-based).
- Design patterns (micro-architectural patterns): Described succinctly, a pattern is "a common solution to a common problem in a given context" [JBR99:p. 447]. Whereas architectural styles can be seen as patterns describing the high-level organization of software systems, thus their *macro*-architecture, other design patterns can be used to describe details at a lower, more local level, thus describing their *micro*-architecture. A wide range of patterns have been discussed in the literature. Such design patterns can (tentatively) be categorized as follows [BCK98:13, BMR+96:1, BRJ99:28]:
  - Creational patterns: e.g., builder, factory, prototype, singleton.
  - Structural patterns: e.g., adapter, bridge, composite, decorator, façade, flyweight, proxy.
  - Behavioral patterns: e.g., command, interpreter, iterator, mediator, memento, observer, state, strategy, template, visitor.
- Families of programs and frameworks: One possible approach to allow the reuse of software designs and components is to design *families* of systems – also known as *software product lines* – which 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 the various members of the family [BCK98:15, Bos00:7,10, Pre97:26].
  In the field of OO programming, a key related notion is that of framework [BMR+96:6, Bos00:11, BRJ99:28]: a framework is a partially complete software subsystem which can be extended by appropriately instantiating some specific plug-ins (also known as hot spots).

IV. Software Design Quality Analysis and Evaluation

A whole knowledge area is dedicated to Software Quality (see chapter 11). Here, we simply mention a number of topics more specifically related with software design.

- Quality attributes: Various attributes are generally considered important for obtaining a design of good quality, e.g., various "ilities" (e.g., maintainability, portability, testability, traceability), various "nesses" (e.g., correctness, robustness), including "fitness of purpose" [BMR+96:6, Bos00:5, Bud97:4, Mar94:D, Mey97:3, Pfl98:5]. An interesting distinction is the one between quality attributes discernable at run-time (e.g., performance, security, availability, functionality, usability), those not discernable at run-time (e.g., modifiability, portability, reusability, integrability and testability) and those related with the intrinsic qualities of the architecture (e.g., conceptual integrity, correctness and completeness, buildability) [BCK98:4].

- Quality analysis and evaluation tools: There exists a variety of tools and techniques that can help ensure the quality of a design. These can be decomposed into a number of categories:
  - Software design reviews: informal or semi-formal, often group-based, techniques to verify and ensure the quality of design artifacts, e.g., architecture reviews [BCK98:10], design reviews and inspections [Bud94:4, FW83:VIII, Jal97:5,7, LG01:14, Pfl98:5], scenario-based techniques [BCK98:9, Bos00:5], requirements tracing [DT97:6, Pfl98:10].
  - Static analysis: formal or semi-formal static (non-executable) analysis that can be used to evaluate a design, e.g., fault-tree analysis or automated cross-checking [Jal97:5, Pfl98:5].
  - Simulation and prototyping: dynamic techniques to evaluate a design, e.g., performance simulation or feasibility prototype [BCK98:10, Bos00:5, Bud94:4, Pfl98:5].
- Measures: Formal measures (a.k.a. metrics) can be used to estimate, in a quantitative way, various aspects of the size, structure or quality of a design. Most measures that have been proposed generally depend on the approach used for producing the design. These measures can thus be classified in two broad categories:
  - Function-oriented (structured) design measures: these measures are used for designs developed using the structured design approach, where the emphasis is mostly on functional decomposition. The structure of the design is generally represented as a structure chart (sometimes called a hierarchical diagram), on which various measures can be computed [Jal97:5,7, Pre97:18].
  - Object-oriented design measures: these measures are used for designs based on object-oriented decomposition. The overall structure of the design is often represented as a class diagram, on which various measures can be defined [Jal97:6,7, Pre97:23]. Measures can also be defined on properties of the internal content of each class.

V. Software Design Notations

A large number of notations and languages exist to represent software design artifacts. Some are used mainly to describe the structural organization of a design, whereas others are used to represent the behavior of such software systems. Note that certain notations are used mostly during architectural design whereas others are useful mainly during detailed design, although some can be used in both steps. In addition, some notations are used mostly in the context of certain specific methods (see section VI). Here, we categorize them into notations for describing the structural (static) view vs. the behavioral (dynamic) view.

- Structural descriptions (static view): These notations, mostly (but not always) graphical, can be used to describe and represent the structural aspects of a software design, that is, to describe what the major components are and how they are interconnected (static view).
  - Architecture Description Languages (ADL): textual, often formal, languages used to describe an architecture in terms of components and connectors [BCK98:12];
  - Class and object diagrams: diagrams used to show a set of classes (and objects) and their relationships [BRJ99:8,14, Jal97:5-6];
  - Component diagrams: used to show a set of components ("physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces" [BRJ99]) and their relationships [BRJ99:12,31]
  - CRC Cards: used to denote the name of components (class), their responsibilities and the names of their collaborating components [BRJ99:4, BMR+96];
  - Deployment diagrams: used to show a set of (physical) nodes and their relationships and, thus, to model the physical aspects of a system [BRJ99:30];
  - Entity-Relationship Diagrams (ERD): used to define conceptual models of data stored in information systems [Bud94:6, DT97:4, Mar94:DR];
  - Interface Description Languages (IDL): programming-like languages used to define the interface (name and types of exported operations) of software components [BCK98:8, BJR99:11];
  - Jackson structure diagrams: used to describe the structure of data in terms of sequence, selection and iteration [Bud94.6, Mar94:DR];
  - Structure charts: used to describe the calling structure of programs (which procedure/module calls/is called by which other) [Bud94:6, Jal97:5, Mar94:DR, Pre97:14];
- Behavioral descriptions (dynamic view): These notations and languages are used to describe the dynamic behavior of systems and components. Such notations include

various graphical notations (e.g., activity diagrams, DFD, sequence diagrams, state transition diagrams) as well as some textual notations (e.g., formal specification languages, pseudo-code and PDL). Many of these notations are useful mostly, but not exclusively, during detailed design.

- Activity diagrams: used to show the flow of control from activity ("ongoing non-atomic execution within a state machine") to activity [BRJ99:19];
- 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 [BRJ99:18];
- Data flow diagrams: used to show the flow of data among a set of processes [Bud94:6, Mar94:DR, Pre97:14];
- Decision tables and diagrams: used to represent complex combination of conditions and actions [Pre97:14];
- Flowcharts and structured flowcharts: used to represent the flow of control and the associated actions to be performed [FW83:VII, Mar94:DR, Pre97:14];
- Formal specification languages: textual languages that use basic notions from mathematics (e.g., logic, set, sequence) to rigorously and abstractly define the interface and behavior of software components, often in terms of pre/post-conditions: [Bud94:14, DT97:5, Mey97:11];
- Pseudo-code and Program Design Languages (PDL): structured, programming-like languages used to describe, generally at the detailed design stage, the behavior of a procedure or method [Bud94:6, FW83:VII, Jal97:7, Pre97:12,14];
- Sequence diagrams: used to show the interactions among a group of objects, with the emphasis on the time-ordering of messages [BRJ99:18];
- State transition and statechart diagrams: used to show the flow of control from state to state in a state machine [BRJ99:24, Bud94:6, Mar94:DR, Jal97:7].

VI. Software Design Strategies and Methods

Various general *strategies* can be used to help guide the design process [Bud94:8, Mar94:D]. By contrast with general strategies, *methods* are more specific in that they generally suggest and provide i) a set of notations to be used with the method; ii) a description of the process to be used when following the method; iii) a set of heuristics that provide guidance in using the method [Bud97:7]. Such methods are useful as a means of transferring knowledge and as a common framework for teams of developers [Bud97:7]. In the following paragraphs, a number of general strategies are first briefly mentioned, followed by a number of methods.

- General strategies: Some often cited examples of general strategies useful in the design process are divide-and-conquer and stepwise refinement [FW83:V], top-down vs. bottom-up strategies [Jal97:5, LG01:13], data abstraction and information hiding [FW83:V], use of heuristics [Bud94:7], use of patterns and pattern languages [BMR+96:5], use of an iterative and incremental approach [Pfl98:2].
- Function-oriented (structured) design [DT97:5, FW83:V, Jal97:5, Pre97:13-14]: This is one of the classical approach to software design, where the decomposition is centered around the identification of the major systems functions and their elaboration and refinement in a top-down manner. Structured design is generally used after structured analysis has been performed, thus producing, among other things, dataflow diagrams and associated processes descriptions. Various strategies (e.g., transformation analysis, transaction analysis) and heuristics (e.g., fan-in/fan-out, scope of effect vs. scope of control) have been proposed to transform a DFD into a software architecture generally represented as a structure chart.
- Object-oriented design [DT97:5, FW83:VI, Jal97:6, Mar94:D, Pre97:19,21]: Numerous software design methods based on objects have been proposed. The field evolved from the early object-based design of the mid-1980's (noun = object; verb = method; adjective = attribute) through object-oriented design, where inheritance and polymorphism play a key role, and to the field of component-based design, where meta-information can be defined and accessed (e.g., through reflection). Although object-oriented design's deep roots stem from the concept of data abstraction, the notion of responsibility-driven design has also been proposed as an alternative approach to object-oriented design.
- Data-structure centered design [FW83:III,VII, Mar94:D]: Although less popular in North America than in Europe, there has been some interesting work (e.g., Jackson, Warnier-Orr) on designing a program starting from the data structures it manipulates rather than from the function it performs. The structures of the input and output data are first described (e.g., using Jackson structure diagrams) and then the control structure of the program is developed based on these data structure diagrams. Various heuristics have been proposed to deal with special cases, for example, when there is mismatch between the input and output structures.
- Other methods: Although software design based on functional decomposition or on object-oriented approaches are probably the most well-known methods to software design, other interesting approaches, although probably less "mainstream", do exist, e.g., formal and rigorous methods [Bud94:14, DT97:5,

Mey97:11, Pre97:25], transformational methods [Pfl98:2].

4. Breakdown Rationale

This section explains the rationale behind the breakdown of topics for the Software Design KA. This is done informally by going through a number of the requirements described in the "Knowledge Area Description Specifications for the Stone Man Version of the Guide to the SWEBOK" (see Appendix A of the whole Guide) and by trying to explain how these requirements influenced the organization and content of the Software Design KA description.

First and foremost, the breakdown of topics must describe "generally accepted" knowledge, that is, knowledge for which there is a "widespread consensus". Furthermore, and this is clearly where this becomes difficult, such knowledge must be "generally accepted" today and expected to be so in a 3 to 5 years timeframe. This latter requirement first explains why elements related with software architecture (see below), including notions related with architectural styles have been included, even though these are relatively recent topics that might not *yet* be generally accepted.

The need for the breakdown to be independent of specific application domains, life cycle models, technologies, development methods, etc., and to be compatible with the various schools within software engineering, is particularly apparent within the "Software Design Strategies and Methods" section. In that section, numerous approaches and methods have been included and references given. This is also the case in the "Software Design Notations", which incorporates pointers to many of the existing notations and description techniques for software design artifacts. Although many of the design methods use specific design notations and description techniques, many of these notations are generally useful independently of the particular method that uses them. Note that this is also the approach used in many software engineering books, including the recent UML series of books by Booch, Jacobson and Rumbaugh, which describe "The Unified Modeling Language" apart from "The Unified Software Development Process".

One point worth mentioning about UML is that although "UML" (Unified Modeling Language) is not explicitly mentioned in the Design Notations section, many of its elements are indeed present, for example: class and object diagrams, collaboration diagrams, deployment diagrams, sequence diagrams, statecharts.

The specifications document also specifically asked that the breakdown be as inclusive as possible and that it includes topics related with quality and measurements. Thus, a certain number of topics have been included in the list of topics even though they may not yet be fully considered as generally accepted. For example, although there are a number of books on measures and metrics, design measures *per se* are rarely discussed in detail and few "mainstream" software engineering books formally discuss this topic. But they are indeed discussed in some books and may become more mainstream in the coming years. Note that although those measures can sometimes be categorized into high-level (architectural) design vs. component-level (detailed) design, the way such measures are defined and used generally depend on the approach used for producing the design, for example, structured vs. object-oriented design. Thus, the measures sub-topics have been divided into function- (structured-) vs. object-oriented design. As the software engineering field matures and classes of software designs evolve, the measures appropriate to each class will become more apparent.

Similarly, there may not yet be a generally accepted list of basic principles and concepts (what was called here the "enabling techniques": see next paragraph for the choice of these terms) on which all authors and software engineers would agree. Only those that seemed the most commonly cited in the literature were included.

As required by the KA Description Specifications, the breakdown is at most three levels deep and use topic names which, based on our survey of the existing literature and on the various reviewers' comments, should be meaningful when cited outside Guide to the SWEBOK. One possible exception might be the use of the terms "enabling techniques", taken from [BMR+96]. In the current context, the term "concept" seemed too general, not specific enough, whereas the term "principle", sometimes used in the literature for some of these notions, sounded too strong (see the definition provided in Section 3).

The rationale for the section "Key Issues in Software Design" is that a number of reviewers of an earlier version suggested that certain topics, not explicitly mentioned in that previous version, be added, e.g., concurrency and multi-threading, exception handling. Although some of these aspects are addressed by some of the existing design methods, it seemed appropriate that these key issues be explicitly identified and that more specific references be given for them, thus the addition of this new section. However, like for the enabling techniques, there does not seem to yet be a complete consensus on what these issues should be, what aspects they should really be addressing, especially since some of those that have been indicated may also be addressed by other topics (e.g., quality). Thus, this section should be seen as a tentative and prototype description that could yet be improved: the author of the Software Design KA Description would gladly welcome any suggestions that could improve and/or refine the content of this section.

In the KA breakdown, as mentioned earlier, an explicit "Software Architecture" section has been included. Here, the notion of "architecture" is to be understood in the large sense of defining the structure, organization and interfaces of the components of a software system, by opposition to producing the "detailed design" of the specific components. This is what really is at the heart of Software Design. Thus, the "Software Architecture" section includes topics which pertain to the macro-architecture of a system – what is now becoming known as "Architecture" *per se,* including notions such as "architectural styles" and "family of programs" – as well as topics related with the micro-architecture of the smaller subsystems – for example, lower-level design patterns which can be used at the detailed design state. Although some of these topics are *relatively* new, they should become much more generally accepted within the 3-5 years timeframe expected from the Guide to the SWEBOK specifications. By contrast, note that no explicit "Detailed Design" section has been included: topics relevant to detailed design can implicitly be found in many places: the "Software Design Notations" and "Software Design Strategies and Methods" sections, "Software Architecture" (design patterns), as well as in "The software design process" subsection.

The "Software Design Strategies and Methods" section has been divided, as is done in many books discussing software design, in a first section that presents general strategies, followed by subsequent sections that present the various classes of approaches (data-, function-, object-oriented or other approaches). For each of these approaches, numerous methods have been proposed and can be found in the software engineering literature. Because of the limit on the number of references, mostly general references have been given, pointers that can then be used as starting point for more specific references.

Another issue, alluded to in the introduction but worth explaining in more detail, is the exclusion of a number of topics which contain the word "design" in their name and which, indeed, pertain to the development of software systems. Among these are the followings: User Interface Design, Real-time Design, Database Design, Participatory Design, Collaborative Design. The first two topics were specifically excluded, in the Straw Man document [BDA+98], from the Software Design KA: User Interface Design was considered to be a related discipline (see the Relevant knowledge areas of related disciplines, where both Computer Science and Cognitive Sciences can be pertinent for UI Design) whereas Real-time Design was considered a specialized sub-field of software design, thus did not have to be addressed in this KA description. The third one, Database Design, can also be considered a relevant (specialized) knowledge area of a related discipline (Computer Science). Note that issues related with user-interfaces and databases still have to be dealt with during the software design process, which is why they are mentioned in the "Key Issues in Software Design" section. However, the specific tasks of designing the details of the user interface or database structure are not considered part of Software Design *per se.* Note also that UI Design is not really part of design for an additional reason: UI Design deals with specifying the external view of the system, not its internal structure and organization, thus should really be considered part of requirements specification.

As for the last two topics – Participatory and Collaborative Design –, they are more appropriately related with the Requirements Engineering KA, rather than Software Design. In the terminology of DeMarco (DeM99), these latter two topics belong more appropriately to I-Design (invention design, done by system analysts) rather than D-design (decomposition design, done by designers and coders) or FP-design (family pattern design, done by architecture groups). It is mainly D-design and FP-design, with a major emphasis on D-design, that can be considered as generally accepted knowledge related with Software Design.

Finally, concerning standards, there seems to be few standards that *directly* pertain to the design task or work product *per se*. However, standards having some indirect relationships with various issues of Software Design do exist, e.g., OMG standards for UML or CORBA. Since the need for the explicit inclusion of standards in the KA breakdown has been put aside ("Proposed changes to the […] specifications […]", Dec. 1999), a few standards having a direct connection with the Software Design KA were included in the Recommended references section. A number of standards related with design in a slightly more indirect fashion were also added to the list of further readings. Finally, additional standards having only an indirect yet not empty connection with design were simply mentioned in the general References section. As for topics related with tools, they are now part of the Software Development Methods and Tools KA.

5. Matrix of Topics vs. Reference Material

The figure below presents a matrix showing the coverage of the topics of the Software Design KA by the various recommended references described in more detail in the following section. A number in an entry indicates a specific section or chapter number. A "*" indicates a reference to the whole document, generally either a journal paper or a standard. An interval of the form "n1-n2" indicates a specific range of pages, whereas an interval of the form "n1:n2" indicates a range of sections. For Mar94, the letters refer to one of the encyclopedia's entry: "D" = Design; "DR" = Design Representation; "DD" = Design of Distributed systems".

Note: Only the top two levels of the breakdown have been indicated in the matrix. Otherwise, especially in the "Software Design Notations" subsections, this would have lead to very sparse lines (in an already quite sparse matrix).

	BCK98	BMR+96	Bos00	BRJ99	Bud94	DT97	FW83	IEE98	ISO95b	Jal97	LG01	Mar94	Mey97	Pfl98	Pre97	SB93
I. Software Design Basic Concepts																
General design concepts					1											*
The context of software design									*		11.1	D		2.2	2.2 : 2.7	
The software design process	2.1, 2.4				2	266- 276	2-22	*	*		13.1 13.2	D			13.8	
Enabling techniques	6.1	6.3	10.3					*		5.1, 5.2, 6.2	1.1, 1.2, 3.1:3. 3, 77- 85, 5.8, 125- 128,9 .1:9.3			5.2, 5.5	13.4, 13.5, 23.2	
II. Key issues in software design																
Concurrency			5.4.1									DD	30		21.3	
Control and events	5.2												32.4, 32.5	5.3		
Distribution	8.3, 8.4	2.3	5.4.1									DD	30		28.1	
Exception handling											4.3:4. 5		12	5.5		
Interactive systems	6.2	2.4	5.4.1								13.3		32.2			
Persistence			5.4.1										31			
III. Software structure and architecture																
Architectural structures and viewpoints	2.5	6.1		31	5.2			*								
Architectural styles and patterns (macro-arch.)	5.1, 5.2, 5.4	1.1: 1.3, 6.2	6.3.1	28										5.3		
Design patterns (micro-arch.)	13.3	1.1: 1.3		28												
Families of programs and frameworks	15.1, 15.3	6.2	7.1, 7.2, 10.2: 10.4, 11.2, 11.4	28											26.4	
IV. Software design quality analysis and evaluation																
Quality attributes	4.1	6.4	5.2.3		4.1: 4.3							D	3	5.5		
Quality analysis and evaluation	9.1, 9.2, 10.2, 10.3		5.2.1 5.2.2 5.3, 5.4		4.4	266- 276	542- 576			5.5, 7.3	14.1			5.6, 5.7, 10.5		
Measures										5.6, 6.5, 7.4					18.4, 23.4, 23.5	

	BCK98	BMR+96	Bos00	BRJ99	Bud94	DT97	FW83	IEE98	ISO95b	Jal97	LG01	Mar94	Mey97	Pfl98	Pre97	SB93
V. Software design notations																
Structural descriptions (static)	8.4, 12.1, 12.2	p. 429		4, 8, 11, 12, 14, 30, 31	6.3, 6.4, 6.6					5.3, 6.3		DR			12.3, 12.4	
Behavioral descriptions (dynamic)				18, 19, 24	6.2, 6.7: 6.9, 14.2.2 14.3.2	181-192	485-490, 506-513			5.3, 7.2		DR	11		14.11 12.5	
VI. Software design strategies and methods																
General strategies		5.1: 5.4			7.1, 7.2, 8		304-320, 533-539			5.1.4	13.13	D		2.2		
Function-oriented design						170-180	328-352			5.4					13.5, 13.6, 14.3: 14.5	
OO design						148-159, 160-169	420-436			6.4		D			19.2, 19.3, 21.1: 21.3	
Data-centered design							201-210,514-532					D				
Other methods					14	181-192	395-407						11	2.2	25.1: 25.3	

6. Recommended References for Software Design

In this section, we give a brief presentation of each of the recommended references. Note that few references to existing standards have been included in this list, for the reasons explained in Section 4; instead, references to interesting standards have been included in the list of further readings. Also note that, because of the constraints on the size of the recommended references list, few specific and detailed references have been given for the various design methods; instead, general software engineering textbook references have been given. See the list of further readings in section 7 for more precise and detailed references on such methods, especially for references to various OO design methods.

Finally, also note that, both in this section and the following, only the author(s) and title of the recommended reference are given, together with an appropriate key that then refers to an entry in the general and detailed References section at the end of the chapter.

[BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice,* Addison-Wesley.

A recent and major work on software architecture. It covers all the major topics associated with software architecture: what software architecture is, quality attributes, architectural styles, enabling concepts and techniques (called unit operations), architecture description languages, development of product lines, etc. Furthermore, it presents a number of case studies illustrating major architectural concepts, including a chapter on CORBA and one on the WWW. Some sections also address the issue of product lines design.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture – A System of Patterns,* J. Wiley and Sons.

Probably one of the best and clearest introduction to the notions of software architecture and patterns (both architectural and lower-level ones). Distinct chapters are dedicated to architectural patterns, design patterns and lower-level idioms. Another chapter discusses the relationships between patterns, software architecture, methods, frameworks, etc. This chapter also includes an brief presentation of "enabling techniques for software architecture", e.g., abstraction, encapsulation, information hiding, coupling and cohesion, etc.

[Bos00] J. Bosch. *Design & Use of Software Architectures – Adopting and Evolving a Product-line Approach*, ACM Press.

The first part of this book is about the design of software architectures and proposes a functionality-based approach coupled with subsequent phases of evaluation and transformation of the resulting architecture. These transformations are expressed in terms of different levels of patterns (architectural styles, architectural patterns and design patterns) and the impact they have on a number of key quality factors (performance, maintainability, reliability and security). The second part of the book is more specifically about the design of software product lines, including a whole chapter on OO frameworks.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide,* Addison-Wesley.

A comprehensive and thorough presentation of the various elements of UML, which incorporates many of the notations mentioned in the "Software Design Notations" section.

[Bud94] D. Budgen. *Software Design,* Addison-Wesley.

One of the few books discussing software design known to the author of the SD KA description – maybe the only one – which is neither a general software engineering textbook nor a book describing a specific software design method. This is probably the book that comes closest to the spirit of the present Software Design KA description, as it discusses topics such as the followings: the nature of design; the software design process; design qualities; design viewpoints; design representations; design strategies and methods (including brief presentations of a number of such methods, e.g., JSP, SSASD, JSD, OOD, etc.). Worth reading to find, in a single book, many notions, views and approaches to/about software design.

[DT97] M. Dorfman and R.H. Thayer (eds.). *Software Engineering,* IEEE Computer Society.

This book contains a collection of papers on software engineering in general. Two chapters deal more specifically with software design. One of them contains a general introduction to software design, briefly presenting the software design process and the notions of software design methods and design viewpoints. The other chapter contains an introduction to object-oriented design and a comparison of some existing OO methods. The following articles are particularly interesting for Software Design:

- D. Budgen, Software Design: An Introduction, pp. 104-115.
- L.M. Northrop, Object-Oriented Development, pp. 148-159.
- A.G. Sutcliffe, Object-Oriented Systems Development: A Survey of Structured Methods, pp.160-169.
- C. Ashworth, Structured Systems Analysis and Design Method (SSADM), pp. 170-180.
- R. Vienneau, A Review of Formal Methods, pp. 181-192.
- J.D. Palmer, Traceability, pp. 266-276.

[FW83] P. Freeman and A.I. Wasserman. *Tutorial on Software Design Techniques*, 4th edition, IEEE Computer Society Press.

Although this is an old book, it is an interesting one because it allows to better understand the evolution of the software design field. This book is a collection of papers where each paper presents a software design technique. The techniques range from basic strategies like stepwise refinement to, at the time, more refined methods such as structured design à la Yourdon and Constantine. An historically important reference. The following articles are particularly interesting:

- P. Freeman, Fundamentals of Design, pp. 2-22.
- D.L. Parnas, On the Criteria to be Used in Decomposing Systems into Modules, pp. 304-309.
- D.L. Parnas, Designing Software for Ease of Extension and Contraction, pp. 310-320.
- W.P. Stevens, G.J. Myers and L.L. Constantine, Structured Design, pp. 328-352.
- G. Booch, Object-Oriented Design, pp. 420-436.
- S.H. Caine and E.K. Gordon, PDL – A Tool for Software Design, pp. 485-490.
- C.M. Yoder and M.L. Schrag, Nassi-Schneiderman Charts: An Alternative to Flowcharts for Design, pp. 506-513.
- M.A. Jackson, Constructive Methods of Program Design, pp. 514-532.
- N. Wirth, Program Development by Stepwise Refinement, pp. 533-539.
- P. Freeman, Toward Improved Review of Software Design, pp. 542-547.
- M.E. Fagan, Design and Code Inspections to Reduce Errors in Program Development, pp. 548-576.

[IEE98] IEEE Std 1016-1998. IEEE Recommended Practice for Software Design Descriptions.

This document describes the information content and recommended organization that should be used for software design descriptions. The attributes describing design entities are briefly described: identification, type, purpose, function, subordinates, dependencies, interfaces, resources, processing and data. How these different elements should be organized is then presented.

[ISO95b] ISO/IEC Std 12207. Information technology – Software life cycle processes.

A detailed description of the ISO/IEC-12207 life cycle model. Clearly shows where Software Design fits in the whole software development life cycle.

[Jal97] P. Jalote. *An integrated approach to software engineering, 2nd ed.,* Springer-Verlag.

A general software engineering textbook with a good coverage of software design, as three chapters discuss this topic: one on function-oriented design, one on object-oriented design, and the other on detailed design. Another interesting point is that all these chapters have a section on measures and metrics.

[LG01] B. Liskov and J. Guttag. *Program Development in Java – Abstraction, Specification, and Object-Oriented Design,* Addison-Wesley, 2000.

A Java version of a classic book on the use of abstraction and specification in software development [LG86]. This new book still discusses, in a clear and insightful way, the notions of procedural vs. data vs. control (iteration) abstractions. It also stresses the importance of appropriate specifications of these abstractions, although this is now done rather informally (with stylized pre/post-conditions in the style of Clu [LG86]). The book also contains a chapter on design patterns. A very good introduction to some of the basic notions of design.

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

A general software engineering encyclopedia that contains (at least) three interesting articles discussing software design. The first one, "Design" (K. Shumate), is a general overview of design discussing alternative development processes (e.g., waterfall, spiral, prototyping), design methods (structured, data-centered, modular, object-oriented). Some issues related with concurrency are also mentioned. The second one discusses the "Design of distributed systems" (R.M. Adler): communication models, client-server and services models. The third one, "Design representation" (J. Ebert), presents a number of approaches to the representation of design. It is clearly not a detailed presentation of any method; however, it is interesting in that it tries to explicitly identify, for each such method, the kinds of components and connectors used within the representation.

[Mey97] B. Meyer. *Object-Oriented Software Construction (Second Edition),* Prentice-Hall*, 2000.*

A detailed presentation of the Eiffel OO language and its associated Design-By-Contract approach, which is based on the use of formal assertions (pre/post-conditions, invariants, etc). It introduces the basic concepts of OO design, along with a discussion of many of the key issues associated with software design, e.g., user interface, exceptions, concurrency, persistence.

[Pfl98] S.L. Pfleeger. *Software Engineering – Theory and Practice,* Prentice-Hall.

A general software engineering book with one chapter devoted to design. Briefly presents and discusses some of the major architectural styles and strategies and some of the concepts associated with the issue of concurrency. Another section presents the notions of coupling and cohesion and also deals with the issue of exception handling. Techniques to improve and to evaluate a design are also presented: design by contract, prototyping, reviews. Although this chapter does not delve into any topic, it can be an interesting starting point for a number of issues not discussed in some of the other general software engineering textbooks.

[Pre97] R.S. Pressman. *Software Engineering – A Practitioner's Approach (Fourth Edition),* McGraw-Hill.

A classic general software engineering textbook (4th edition!). It contains over 10 chapters that deal with notions associated with software design in one way or another. The basic concepts and the design methods are presented in two distinct chapters. Furthermore, the topics pertaining to the function-based (structured) approach are separated (part III) from those pertaining to the object-oriented approach (part IV). Independent chapters are also devoted to measures applicable to each of those approaches, a specific section addressing the measures specific to design. A chapter discusses formal methods and another presents the Clean-room approach. Finally, another chapter discusses client-server systems and distribution issues.

[SB93] G. Smith and G. Browne. Conceptual foundations of design problem-solving, *IEEE Transactions on Systems, Man and Cybernetics*, vol. 23, no. 5 Sep-Oct. 1993, pp. 1209-1219.

A paper that discusses what is design in general. More specifically, it presents the five basic concepts of design: goals, constraints, alternatives, representations, and solutions. The bibliography is a good starting point for obtaining additional references on design in general.

APPENDIX A – LIST OF FURTHER READINGS

The following section suggests a list of additional reading material related with Software Design. A number of standards are mentioned; additional standards that may be pertinent or applicable to Software Design, although in a somewhat less direct way, are also mentioned, although not further described, in the general References section at the end of the document.

[Boo94] G. Booch. *Object Oriented Analysis and Design with Applications, 2nd ed.*

A classic in the field of OOD. The book introduces a number of notations that were to become part of UML (although sometimes with some slight modifications): class vs. objects diagrams, interaction diagrams, statecharts-like diagrams, module and deployment, process structure diagrams, etc. It also introduces a process to be used for OOA and OOD, both a higher-level (life cycle) process and a lower-level (micro-) process. (Note that a third edition of this book is expected.)

[Cro84] N. Cross (ed.). *Developments in Design Methodology.*

This book consists in a series of papers related to design in general, that is, design in other contexts than software. Still, many notions and principles discussed in some of these papers do apply to Software Design, e.g., the idea of design as wicked-problem solving.

[CY91] P. Coad and E. Yourdon. *Object-Oriented Design.*

This is yet another classic in the field of OOD – note that the second author is one of the father of classical Structured Design. An OOD model developed with their approach consists of the following four components that attempt to separate how some of the key issues should be handled: problem domain, human interaction, task management and data management.

[DW99] D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML – The Catalysis Approach.*

A thorough presentation of a specific OO approach with an emphasis on component design. The development of static, dynamic and interaction models is discussed. The notions of components and connectors are presented and illustrated with various approaches (Java Beans, COM, Corba); how to use such components in the development of frameworks is also discussed. Another chapter discusses various aspects of software architecture. The last chapter introduces a pattern system for dealing with both high-level and detailed design, the latter level touching on many key issues of design such as concurrency, distribution, middleware, dialogue independence, etc.

[Fow99] M. Fowler. *Refactoring – Improving the Design of Existing Code.*

A book about how to improve the design of some existing (object-oriented) code. The first chapter is a simple and illustrative example of the approach. Subsequent chapter present various categories of strategies, e.g., composing methods, moving features between objects, organizing data, simplifying conditional expressions, making methods calls simpler.

[FP97] N.E. Fenton and S.L. Pfleeger. *Software Metrics – A Rigorous & Practical Approach (Second Edition).*

This book contains a detailed presentation of numerous software measures and metrics. Although the measures are not necessarily presented based on the software development life cycle, many of those measures, especially those in chapters 7 and 8, are applicable to software design.

[GHJV95] E. Gamma et *al. Design Patterns – Elements of Reusable Object-Oriented Software.*

The seminal work on design patterns. A detailed catalogue of patterns related mostly with the micro-architecture level.

[Hut94] A.T.F. Hutt. *Object Analysis and Design – Description of Methods. Object Analysis and Design – Comparison of Methods.*

These two books describe (first book) and compare (second book), in an outlined manner, a large number of OO analysis and design methods. Useful as a starting point for obtaining additional pointers and references to OOD methods, not so much as a detailed presentation of those methods.

[IEE90] IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology.

This standard is not specifically targeted to Software Design, which is why it has not been included in the recommended references. It describes and briefly explains many of the common terms used in the Software Engineering field, including many terms from Software Design.

[ISO91] ISO/IEC Std 9126. Information technology – Software product evaluation – Quality characteristics and guidelines for their use.

This standard describes six high-level characteristics that describe software quality: functionality, reliability, usability, efficiency, maintainability, portability.

[JBP+91] J. Rumbaugh et *al. Object-Oriented Modeling and Design.*

This book is another classic in the field of OOA and OOD. It was one of the first to introduce the distinctions between object, dynamic and functional modeling. However, contrary to [Boo94] whose emphasis is mostly on design, the emphasis here is slightly more on analysis, although a number of elements do apply to design too.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.*

A detailed and thorough presentation of the Unified Software Development Process proposed by the Rational

Software Corporation. The notion of architecture plays a central role in this development process, the process being said to be architecture-centric. However, the associated notion of architecture seems to be slightly different from the traditional purely design-based one: an architecture description is supposed to contain views not only from the design model but also from the use-case, deployment and implementation models. A whole chapter is devoted to the presentation of the iterative and incremental approach to software development. Another chapter is devoted to design *per se*, whose goal is to produce both the design model, which includes the logical (e.g., class diagrams, collaborations, etc.) and process (active objects) views, and the deployment model (physical view).

[Kru95] P.B. Kruchten. The 4+1 view model of architecture.

A paper that explains in a clear and insightful way the importance of having multiple views to describe an architecture. Here, architecture is understood in the sense mentioned earlier in reference [JBR99], not in its strictly design-related way. The first four views discussed in the paper are the logical, process, development and physical views, whereas the fifth one (the "+1") is the use case view, which binds together the previous views. The views more intimately related with Software Design are the logical and process ones.

[Lar98] C. Larman. *Applying UML and Patterns – An introduction to Object-Oriented Analysis and Design*.

An introductory book that covers object-oriented analysis and design, doing so through a case study used throughout the book. Part IV and VII are dedicated to the design phase. They introduce a number of patterns to guide the assignment of responsibilities to classes and objects. Various issues regarding design are also addressed, e.g., multi-tiers architecture, model-view separation. The patterns of [GHJV95] are also examined in the context of the case study.

[McC93] S. McConnell. *Code Complete.*

Although this book is probably more closely related with Software Construction, it does contain a section on Software Design with a number of interesting chapters, e.g., "Characteristics of a High-Quality Routines", "Three out of Four Programmers Surveyed Prefer Modules", "High-Level Design in Construction". One of these chapters ("Characteristics […]") contains an interesting discussion on the use of assertions in the spirit of Meyer's Design-by-Contract; another chapter ("Three […]") discusses cohesion and coupling as well as information hiding; the other chapter ("High-Level […]") gives a brief introduction to some design methodologies (structured design, OOD).

[otSESC98] Draft recommended practice for information technology – System design – Architectural description. Technical Report IEEE P1471/D4.1.

"This recommended practice establishes a conceptual framework for architectural description. This framework covers the activities involved in the creation, analysis, and sustainment of architectures of software-intensive systems, and the recording of such architectures in terms of *architectural descriptions*." (from the Abstract)

[Pet92] H. Petroski. *To Engineer is Human – The role of failure in successful design.*

This book is not about software design *per se*. The author, a civil engineer, discusses how a designer, an engineer can and should learn from previous failures and how a design should be seen as a kind of hypothesis to be tested. Interestingly, considering that Software Design is only one out of the 10 knowledge areas for software engineering, the author "take[s] design and engineering to be virtually synonymous".

[PJ00] M. Page-Jones. *Fundamentals of Object-Oriented Design in UML.*

Part III of this book ("Principles of object-oriented design") addresses a number of the enabling techniques in the specific context of OO design. This part of the book contains chapters such as the followings: Encapsulation and connascence; Domains, encumbrance, and cohesion; Type conformance and closed behavior; The perils of inheritance and polymorphism. The book also contains a chapter on the design of software components.

[Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development.*

This book is particularly interesting for its discussion of framework design using what is called the "hot-spot driven" approach to the design of frameworks. The more specific topic of design patterns is better addressed in [BMR+96].

[Rie96] A.J. Riel. *Object-Oriented Design Heuristics.*

This book, targeted mainly towards OO design, presents a large number of heuristics that can be used in software design. Those heuristics address a wide range of issues, both at the architectural level and at the detailed design level.

[SG96] M. Shaw, D. Garlan. *Software architecture: Perspectives on an emerging discipline.*

One of the early book on software architecture that addresses many facets of the topic: architectural styles (including a chapter with a number of small case studies), shared information systems, user-interface architectures, formal specifications, linguistic issues, tools and education.

[Som95] I. Sommerville. *Software Engineering (fifth edition)*. Addison-Wesley, 1995.

Part Three is dedicated to software design, giving an overview of a number of topics through the following chapters: the design process, architectural design, OO design, functional design. (Note: a sixth edition may already be available.)

[WBWW90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.*

A book that introduced the notion of responsibility-driven design to OOD. Until then, OOD was often considered synonymous with data abstraction-based design. Although it is true that an object does encapsulate data and associated behavior, focusing strictly on this aspect may not lead, according to the responsibility-driven design approach, to the best design.

[Wie98] R. Wieringa. A Survey of Structured and Object-Oriented Software Specification Methods and Techniques.

An interesting survey article that presents a wide range of notations and methods for specifying software systems and components. It also introduces an interesting framework for comparison based on the kinds of system properties to be specified: functions, behavior, communication or decomposition.

Appendix B – References Used to Write and Justify the Knowledge Area Description

[BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* SEI Series in Software Engineering. Addison-Wesley, 1998.

[BDA+98] P. Bourque, R. Dupuis, A. Abran, J.W. Moore, L. Tripp, J. Shyne, B. Pflug, M. Maya, and G. Tremblay. Guide to the software engineering body of knowledge – a straw man version. Technical report, Dépt. d'Informatique, UQAM, Sept. 1998.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture – A System of Patterns*. John Wiley & Sons, 1996.

[Boo94] G. Booch. *Object Oriented Analysis and Design with Applications, 2nd ed.* The Benjamin/Cummings Publishing Company, Inc., 1994.

[Bos00] J. Bosch. *Design & Use of Software Architecture – Adopting and Evolving a Product-line Approach.* ACM Press, 2000.

[BRJ99] G. Booch, J. Rumbauch, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[Bud94] D. Budgen. *Software Design.* Addison-Wesley, 1994.

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

[CY91] P. Coad and E. Yourdon. *Object-Oriented Design.* Yourdon Press, 1991.

[DeM99] T. DeMarco. *The Paradox of Software Architecture and Design.* Stevens Prize Lecture, August 1999.

[DT97] M. Dorfman and R.H. Thayer. *Software Engineering.* IEEE Computer Society Press, 1997.

[DW99] D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML – The Catalysis Approach.* Addison-Wesley, 1999.

[Fow99] M. Fowler. *Refactoring – Improving the Design of Existing Code.* Addison-Wesley, 1999.

[FP97] N.E. Fenton and S.L. Pfleeger. *Software Metrics – A Rigorous & Practical Approach (Second Edition).* International Thomson Computer Press, 1997.

[FW83] P. Freeman and A.I. Wasserman. *Tutorial on Software Design Techniques*, fourth edition. IEEE Computer Society Press, 1983.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Professional Computing Series. 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.

[IEE88] IEEE. IEEE Standard Dictionary of Measures to Produce Reliable Software. IEEE Std 982.1-1988, IEEE, 1988.

[IEE88b] IEEE. IEEE Guide for the Use of Standard Dictionary of Measures to Produce Reliable Software. IEEE Std 982.2-1988, IEEE, 1988.

[IEE90] IEEE. IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990, IEEE, 1990.

[IEE98] IEEE. IEEE Recommended Practice for Software Design Descriptions. IEEE Std 1016-1998, IEEE, 1998.

[ISO91] ISO/IEC. Information technology – Software product evaluation – Quality characteristics and guidelines for their use. ISO/IEC Std 9126: 1991, ISO/IEC, 1991.

[ISO95] ISO/IEC. Open distributed processing – Reference model. ISO/IEC Std 10746: 1995, ISO/IEC, 1995.

[ISO95b] ISO/IEC. Information technology – Software life cycle processes. ISO/IEC Std 12207: 1995, ISO/IEC, 1995.

[Jal97] P. Jalote. *An Integrated Approach to Software Engineering, 2nd ed.* Springer, 1997.

[JBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering – A Use Case Driven Approach.* Addison-Wesley, 1992.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97 – Object-Oriented Programming*, pages 220-242. LNCS-1241, Springer-Verlag, 1997.

[Kru95] P.B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[Lar98] C. Larman. *Applying UML and Patterns – An introduction to Object-Oriented Analysis and Design*. Prentice-Hall, 1998.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. The MIT Press, 1986.

[LG01] 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.* John Wiley & Sons, Inc., 1994.

[McCr93] S. McConnell. *Code Complete.* Microsoft Press, 1993.

[Mey97] B. Meyer. *Object-Oriented Software Construction (Second Edition).* Prentice-Hall, 1997.

[OMG98] OMG. The common object request broker: Architecture and specification. Technical Report Revision 2.2, Object Management Group, February 1998.

[OMG99] UML Revision Task Force. OMG Unified Modeling Language specification, v. 1.3. document ad/99-06-08, Object Management Group, June 1999.

[otSESC98] Architecture Working Group of the Software Engineering Standards Committee. Draft recommended practice for information technology – System design – Architectural description. Technical Report IEEE P1471/D4.1, IEEE, December 1998.

[Pet92] H. Petroski. *To Engineer is Human – The role of failure in successful design.* Vintage Books, 1992.

[Pfl98] S.L. Pfleeger. *Software Engineering – Theory and Practice.* Prentice-Hall, Inc., 1998.

[PJ00] M. Page-Jones. *Fundamentals of Object-Oriented Design in UML.* Addison-Wesley, 2000.

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

[Pre97] R.S. Pressman. *Software Engineering – A Practitioner's Approach (Fourth Edition).* McGraw-Hill, Inc., 1997.

[Rie96] A.J. Riel. *Object-Oriented Design Heuristics.* Addison-Wesley, 1996.

[SB93] G. Smith and G. Browne. Conceptual foundations of design problem-solving. *IEEE Trans. on Systems, Man, and Cybernetics,* 23(5):1209–1219, 1993.

[SG96] M. Shaw, D. Garlan. *Software architecture: Perspectives on an emerging discipline*. Prentice-Hall, 1996.

[Som95] I. Sommerville. *Software Engineering (fifth edition)*. Addison-Wesley, 1995.

[WBWW90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice-Hall, 1990.

[Wie98] R. Wieringa. A Survey of Structured and Object-Oriented Software Specification Methods and Techniques. *ACM Computing Surveys*, 30(4): 459–527, 1998.