CHAPTER 3

SOFTWARE DESIGN

ACRONYMS

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

INTRODUCTION

Design is defined in [1] 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 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 Std 12207 Software Life Cycle Processes [2], 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.

The current Software Design Knowledge Area (KA) description does not discuss every topic whose name contains the word "design." In Tom DeMarco's terminology [3], the KA discussed in this chapter deals mainly with D-design (decomposition design, whose goal is to map software into component pieces). However, because of its importance in the 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, which is usually performed during the software requirements process with the goal of conceptualizing and specifying software to satisfy discovered needs and requirements), since this topic should be considered part of the Software Requirements KA.

The Software Design KA description is related specifically to Software Requirements, Software Construction, Software Engineering Management, Software Engineering Models and Methods, Software Quality, and Computing Foundations KAs.

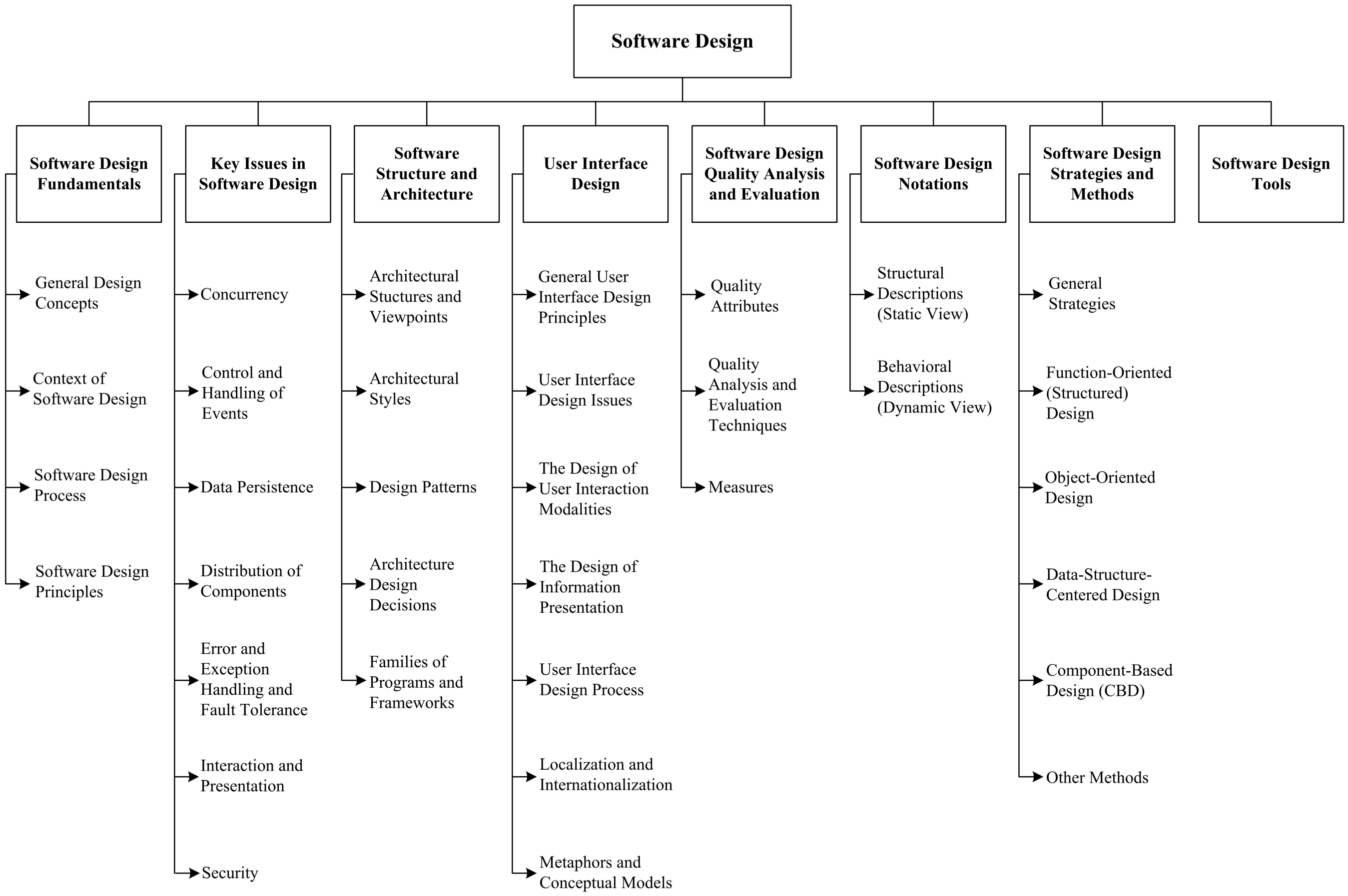

Figure 1: Breakdown of Topics

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

[4* c1]

Software is not the only field where design is involved. In the general sense, we can view design as a form of problem solving. For example, the concept of a *wicked* problem–a problem with no definitive solution–is interesting in terms of understanding the limits of design. A number of other notions and concepts are also of interest in understanding design in its general sense: goals, constraints, alternatives, representations, and solutions. (See also Problem Solving Techniques in Computing Foundations KA.)

1.2. Context of Software Design

[4* c3]

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.

1.3. Software Design Process

[4* c2]

Software design is generally considered a two-step process.

1.3.1. Architectural design

Architectural design describes how software is decomposed and organized into components.

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.

1.4. Software Design Principles

[4 c1]*

[5 c6,c7,c21]*

[6 c1,c8,c9]*

According to [7], a *principle* is "a comprehensive and fundamental law, doctrine, or assumption." Software design principles are key notions considered fundamental to many different software design approaches and concepts. Software design principles include abstraction, coupling, and cohesion; decomposition and modularization; encapsulation/information hiding; separation of interface and implementation; sufficiency, completeness, and primitiveness; and separation of concerns.

1.4.1. Abstraction

According to [1], *abstraction* is "a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information" (see also Abstraction in Computing Foundations). 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.

1.4.2. Coupling and cohesion

Coupling is defined as " a measure of the interdependence among modules in a computer program," whereas *cohesion* is defined as "a measure of the strength of association of the elements within a module" [1].

1.4.3. Decomposition and modularization

Decomposing and *modularizing* means that large software are divided into a number of smaller independent ones, usually with the goal of placing different functionalities or responsibilities in different components.

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.

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) that is separate from the details of how the component is realized.

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.

1.4.7. Separation of concerns

Separation of concerns suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A *concern* is an "area of interest with respect to a software design" [8]. By separating concerns into smaller—and therefore more manageable—pieces, a problem takes less effort and time to solve.

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, *"Software Design Principles,"* and subarea 7, *"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" [9]. Such issues, which often cross-cut the system's functionality, have been referred to as *aspects*, which "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" [10]. A number of these key, cross-cutting issues are discussed in the following sections (presented in alphabetical order):

2.1. Concurrency

[5* c18]

This issue looks at how to decompose the software into processes, tasks, and threads and deal with related efficiency, atomicity, synchronization, and scheduling issues.

2.2. Control and Handling of Events

[5* c21]

This issue looks at how to organize data and control flow as well as how to handle reactive and temporal events through various mechanisms such as implicit invocation and call-backs.

2.3. Data Persistence

[11* c9]

This issue looks at how to handle long-lived data.

2.4. Distribution of Components

[5*c18]

This issue looks at how to distribute the software across the hardware, how the components communicate, and how middleware can be used to deal with heterogeneous software.

2.5. Error and Exception Handling and Fault Tolerance

[5* c18]

This issue looks at how to prevent and tolerate faults and deal with exceptional conditions.

2.6. Interaction and Presentation

[5* c16]

This issue looks at how to structure and organize interactions with users as well as the presentation of information (for example, separation of presentation and business logic using the Model-View-Controller approach). Note that this topic does not specify user interface details, which is the task of user interface design (see subarea 4, "User Interface Design").

2.7. Security

[5* c12, c18, 12* c4]

This issue looks at how to preventunauthorized disclosure, creation, changing, deleting or denying of information and other resources; and how to tolerate security-related attacks or violations by limiting damage, continuing service, speeding repair and recovery, and failing and recovering securely. Access control is fundamental to much of security; one must also ensure the proper use of cryptology.

3. Software Structure and Architecture

In its strict sense, a *software architecture* is "the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both" [13*]. Architecture thus attempts to define the internal *structure* of the resulting software. During the mid-1990s, however, software architecture started to emerge as a broader discipline that involved the study of software structures and architectures in a more generic way. 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 the detailed design (for example, lower-level design patterns) of that software. But they can also be useful for designing genericsoftware, 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 [13 c1]*

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” [13*]. 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.

3.2. Architectural Styles

[13* c1, c2, c3, c4, c5]

An architectural style is “a specialization of element and relation types, together with a set of constraints on how they can be used” [13*]. An architectural style can thus be seen as providingthe software’s high-level organization. Various authors have identified a number of major architectural styles.

- General structure (for example, layers, pipes, 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).

3.3. Design Patterns

[14* c3,c4,c5]

Succinctly described, a pattern is “a common solution to a common problem in a given context” [15]. While architectural styles can be viewed as patterns describing the high-level organization of software, other design patterns can be used to describe details at a lower, more local level (their *micro*architecture).

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

3.4. Architecture Design Decisions

[5* c6]

Architectural design is a creative process. During this process, software architects have to make a number of fundamental decisions that profoundly affect the software and its development process.It is more useful to think of the architectural design process from a decision perspective rather than from an activity perspective.

3.5. Families of Programs and Frameworks

[5* c6,c7,c16]

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.

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.

4. User Interface Design

User interface design is an essential part of the software design process. To achieve a software’s full potential, its user interface should be designed to match the skills, experience, and expectations of its anticipated users.

4.1 General User Interface Design Principles

[5* c29-web, 16* c2][1]

- *Learnability*. The software should be easy to learn so that the user can rapidly start getting some work done with the software.
- *User familiarity*. The interface should use terms and concepts drawn from the experiences of the people who will make most use of the software.
- *Consistency*. The interface should be consistent so that comparable operations are activated in the same way.
- *Minimal surprise.* The behavior of software should not surprise users.
- *Recoverability.* The interface should provide mechanisms allowing users to recover from errors.
- *User guidance.* The interface should give meaningful feedback when errors occur and provide context-related help to users.
- *User diversity*. The interface should provide appropriate interaction mechanisms for different types of users.

4.2 User Interface Design Issues

[5* c29-web, 16* c2]

User interface design should solve two key issues:

(1) How should the user interact with the software?

(2) How should information from the softwarebe presented to the user?

User interface must integrate user interaction and information presentation. User interface design should consider a compromise between the most appropriate styles of interaction and presentation for the software, the background and experience of the softwareusers, and the available devices.

[1] Chapter 29 is a web-based chapter available at http://www.cs.st-andrews.ac.uk/~ifs/Books/SE9/WebChapters/index.html

4.3 The Design of User Interaction Modalities

[5* c29-web, 16* c2]

User interaction means issuing commands and associated data to the software. User interaction styles can be classified into the primary styles as follows.

- *Question-answer.* The interaction is essentially restricted to a single question-answer exchange between the user and the software. The user issues a question to the software, and the software returns the answer to the question. It is line-oriented.
- *Direct manipulation*. Users interact directly with objects on the screen. Direct manipulation often includes a pointing device (such as a mouse, trackball, or finger on touch screens) that guides the manipulated object and action that specifies what should be done with that object.
- *Menu selection*. The user selects a command from a menu list of commands.
- *Form fill-in*. The user fills in the fields of a form. Sometimes fields include menus, in which case the form has action buttons for the user to initiate action.
- *Command language*. The user issues a command and related parameters to direct the software what to do.
- *Natural language*. The user issues a command in natural language. That is, the natural language is a front end to a command language and is parsed and translated to software commands.

4.4 The Design of Information Presentation

[5* c29-web, 16* c2]

Software often needs to provide some way of presenting information to users. Such information presentation may be a direct representation of the input informationor it may begraphical information. A good design should keep the information presentation separatefrom the information itself.

Software engineers must consider software response time and feedback in the design of information presentation. Software response time is generally measured from the point at which a user executes a certain control action until the software responses with the desired output or response. Before the software returns the desired response, it should give feedback on what the software is doing. Software feedback should not be expressed in abstract and general terms but should restate and rephrase the user's input to indicate what processing is being completed from this input..

When large amounts of information have to be presented, abstract visualizations that link data items can be used.

According to the style of information presentation, designers should think about how to color the interface. There are several important guidelines, which follow.

- *Limit the number of colors used.*
- *Use color change to show the change of software status.*
- *Use color coding to support the user's task.*
- *Use color coding in a thoughtful and consistent way.*

4.5 User Interface Design Process

[5* c29-web, 16* c2]

User interface design is an iterative process; interface prototypes are often used to decide the features, organization, and look of the software user interface. This process includes three core activities:

- *User analysis.* In this phase, the designer should analyze the users' tasks, working environment, and other software as well as how users interact with other people.
- Software *prototyping.* Developing prototype software and exposing them to users can guide the evolution of the interface.
- *Interface evaluation.* Designers can formally evaluate users' actual experiences with the interface.

4.6 Localization and Internationalization

[16* c8,c9]

User interface design needs to consider internationalization and localization, which are means of adapting software to the different languages, regional differences, and technical requirements of a target market. Internationalization is the process of designing a software application so that it can be adapted to various languages and regions without engineering changes. Localization is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translating text. Localization and internationalization should notably consider characters, numbers and currency, time and measurement units.

4.7 Metaphors and Conceptual Models

[16 c5]*

User interface design often needs to set up the mappings between the information display and the user's conceptual model of the information.

User interface design can use interface metaphors to set up a mapping between the software and some reference system known to the users in the real world in order to help them to learn and use the interface. For example, the operation "delete file" can be metaphorized as the icon of a trash can in user interfaces.

When designing a user interface, software engineers should pay attention to not imply more than one intended metaphor. Metaphors also present potential problems with respect to internationalization, since not all metaphors are meaningful or not in the same way to all cultures.

5. 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.

5.1. Quality Attributes

[4* c4]

Various attributes are generally considered important for obtaining a software design of good quality—various “ilities” (for example, maintainability, portability, testability, traceability) and “nesses” (for example, correctness, robustness), including “fitness of purpose.” There is an interesting distinction between quality attributes discernible at run-time (for example, performance, security, availability, functionality, usability), those not discernible at run-time (for example, modifiability, portability, reusability, integrability, and testability), and those related to the architecture’s intrinsic qualities (for example, conceptual integrity, correctness, completeness, and buildability). (See also the Software Quality KA for further discussion on this topic.)

5.2. Quality Analysis and Evaluation Techniques

[4* c4, 5* c24]

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, design reviews, and inspections; scenario-based techniques, requirements tracing). Software design reviews can also examine security, including performing vulnerability analysis. Installer, operator, and user aids (for example, manuals and help files) can be reviewed to ensure that they include security considerations.
- 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). Design vulnerability analysis (for example, static analysis for security weaknesses) can be performed if security is a concern. Formal design analysisuses mathematically based models that allow designers to predicate the behavior and validate the accuracy of a softwareinstead of having to rely entirely on non-assuring exhaustive testing. Formal design analysis can eliminate residual specification and design errors (caused by imprecision, ambiguity, and sometimesplain mistakes). (Also see the Software Engineering Models and Methods KA.)
- Simulation and prototyping: dynamic techniques to evaluate a design (for example, performance simulation or feasibility prototype).

5.3. Measures

[4* c4, 5* c24]

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.
- 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.

6. 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 subarea 7,*“Software Design Strategies and*

Methods”). Please note that software design is often accomplished using multiple notations. Here, they are categorized into notations for describing the structural (static) view vs. the behavioral (dynamic) view.

6.1. Structural Descriptions (Static View)

[4* c7, 5* c6, c7, 6* c4, c5, c6, c7, 11* c7, 13* c7]

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.
- Class and object diagrams: used to represent a set of classes (and objects) and their interrelationships.
- Component diagrams: used to represent a set of components (“physical and replaceable part[s] of asystemthat [conform] to and [provide] the realization of a set of interfaces” [17]) and their interrelationships.
- Class responsibility collaborator cards (CRCs): used to denote the names of components (class), their responsibilities, and their collaborating components’ names.
- Deployment diagrams: used to represent a set of (physical) nodes and their interrelationships, and, thus, to model the physical aspects of a software. Usually, only certain deployed configurations are secure.
- Entity-relationship diagrams (ERDs): used to represent conceptual models of data stored in information systems.
- Interface description languages (IDLs): programming-like languages used to define the interfaces (names and types of exported operations) of software components.
- Jackson structure diagrams: used to describe the data structures in terms of sequence, selection, and iteration.
- Structure charts: used to describe the calling structure of programs (which module calls, and is called by, which other module).

6.2. Behavioral Descriptions (Dynamic View)

[4* c7, c13, 5* c6, c7, 6* c4, c5, c6, c7, 13* c8]

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. Moreover, behavioral descriptions can include a rationale for why design will meet security requirements.

- Activity diagrams: used to show the control flow from activity (ongoing non-atomic execution within a state machine) to activity.
- Collaboration diagrams: used to show the interactions that occur among a group of objects; emphasis is on the objects, their links, and the messages they exchange on those links.
- Data flow diagrams (DFDs): used to show data flow among a set of processes. A data flow diagram provides “a description based on modeling the flow of information around a network of operational elements, which each element making use of or modifying the information flowing into that element” [4*]. Data flows (and therefore possibly data-flow diagrams) are important to security as they offer possible paths for attack and disclosure of confidential information.
- Decision tables and diagrams: used to represent complex combinations of conditions and actions.

- Flowcharts and structured flowcharts: used to represent the flow of control and the associated actions to be performed.
- Sequence diagrams: used to show the interactions among a group of objects, with emphasis on the time-ordering of messages.
- State transition and statechart diagrams: used to show the control flow from state to state in a state machine.
- 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. (Also see the Software Engineering Models and Methods KA for more information.)
- 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.

7. Software Design Strategies and Methods

There exist various general *strategies* to help guide the design process. 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. Such methods are useful as a means of transferring knowledge and as a common framework for teams of software engineers. (See also the Software Engineering Models and Methods KA).

7.1. General Strategies

[4* c8,c9,c10, 11* c7]

Some often-cited examples of general strategies useful in the design process include the divide-and-conquer and stepwise refinement strategies, top-down vs. bottom-up strategies, and strategies making use of heuristics, use of patterns and pattern languages, and use of an iterative and incremental approach.

7.2. Function-Oriented (Structured) Design

[4* c13]

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.

7.3. Object-Oriented Design

[4* c16]

Numerous software design methods based on objects have been proposed. The field has evolved from the early object-oriented (OO) design of the mid-1980s (noun = object; verb = method; adjective = attribute), 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.

7.4. Data-Structure-Centered Design

[4* c14,c15]

Data-structure-centered design 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.

7.5. Component-Based Design (CBD)

[4* c17]

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. Reused and off-the-shelf software components should meet the same security requirements as new software. Trust management is a design concern; components treated as having a certain degree of trustworthiness cannot depend on less trustworthy components or services.

7.6. Other Methods

[5 c19,c21]*

Other interesting approaches also exist (see the Software Engineering Models and Methods KA for more information). Agile methods propose to quickly implement an incremental basis by reducing emphasis on rigorous software requirement and design. Aspect-oriented design is a method which designs a softwareby using aspects to implement the cross-cutting concerns and extensions that are identified during the software requirements engineering process. And, finally, service-oriented architecture is a way to build distributed software using web service.

8. Software Design Tools

[13* c10, Appendix A]

Software design tools are used for design activities during the software development process. They assist designers in transforming software requirement specificationsinto software design artifacts. In detail, they implement part or whole of the following functions: (1) to translate the requirements model into a design representation; (2) to provide a notation for representing functional components and their interface; (3) to implement heuristics refinement and partitioning; (4) to provide guidelines for quality assessment.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Budgen [4*]	Sommerville [5*]	Page-Jones [6*]	Brookshear [11*]	Allen [12*]	Clements [13*]	Gamma [14*]	Nielsen [16*]
1. Software Design Fundamentals								
1.1 General Design Concepts	c1							
1.2 The Context of Software Design	c3							
1.3 The Software Design Process	c2							
1.4 Software Design Principles	c1	c6,c7,c21	c1,c8,c9					
2. Key Issues in Software Design								
2.1 Concurrency		c18						
2.2 Control and Handling of Events		c21						
2.3 Data Persistence				c9				
2.4 Distribution of Components		c18						
2.5 Error and Exception Handling and Fault Tolerance		c18						
2.6 Interaction and Presentation		c16						
2.7 Security		c12,c18			c4			
3. Software Structure and Architecture								
3.1 Architectural Structures and Viewpoints						c1		
3.2 Architectural Styles						c1,c2,c3,c4,c5		
3.3 Design Patterns							c3,c4,c5	
3.4 Architecture Design Decisions		c6						
3.5 Families of Programs and Frameworks		c6,c7,c16						
4. User Interface Design								
4.1 General User Interface Design Principle		c29-web						c2
4.2 User Interface Design Issues		c29-web						
4.3 The Design of		c29-web						

	Budgen [4*]	Sommerville [5*]	Page-Jones [6*]	Brookshear [11*]	Allen [12*]	Clements [13*]	Gamma [14*]	Nielsen [16*]
User Interaction Modalities								
4.4 The Design of Information Presentation		c29-web						
4.5 User Interface Design Process		c29-web						
4.6 Localization and internationalization								c8,c9
4.7 Metaphors and Conceptual Models								c5
5. Software Design Quality Analysis and Evaluation								
5.1 Quality Attributes	c4							
5.2 Quality Analysis and Evaluation Techniques	c4	c24						
5.3 Measures	c4	c24						
6. Software Design Notations								
6.1 Structural Descriptions (Static View)	c7	c6,c7	c4,c5,c6,c7	c7		c7		
6.2 Behavioral Descriptions (Dynamic View)	c7,c13,c18	c6,c7	c4,c5,c6,c7			c8		
7. Software Design Strategies and Methods								
7.1 General Strategies	c8,c9,c10			c7				
7.2 Function-Oriented (Structured) Design	c13							
7.3 Object-Oriented Design	c16							
7.4 Data-Structure-Centered Design	c14,c15							
7.5 Component-Based Design (CBD)	c17							
7.6 Other Methods		c19,c21						
8Software Design Tools						c10,Apendix A		

901

APPENDIX A. LIST OF FURTHER READINGS

Software Engineering: A Practitioner's Approach (Seventh Edition) [18]

For roughly three decades, Roger Pressman's *Software Engineering: A Practitioner's Approach* has been one of the world's leading textbooks in software engineering. Notably, this complementary textbook to [5*] comprehensively presents software design—including design concepts, architectural design, component-level design, user interface design, pattern-based design, and web application design.

The 4+1 View Model of Architecture [19]

The 4+1 View Model seminal paper organizes a description of a software architecture using five concurrent views. The four views of the model are the logical view, the development view, the process view, and the physical view. In addition, selected use cases or scenarios are utilized to illustrate the architecture. Hence, the model contains 4+1 views. The views are used to describe the software as envisioned by different stakeholders—such as end-users, developers, and project managers.

Software Architecture in Practice [20]

This book introduces the concepts and best practices of software architecture, meaning how a software is structured and how the software's components interact. Drawing on their own experience, the authors cover the essential technical topics for designing, specifying, and validating software architectures. They also emphasize the importance of the business context in which large software is designed. Their aim is to present software architecture in a real-world setting, reflecting both the opportunities and constraints that organizations encounter. This is one of the best books currently available on software architecture.

[1] IEEE/ISO/IEC, "IEEE/ISO/IEC 24765 Systems and Software Engineering Vocabulary," 1st ed, 2010.

[2] ISO/IEC/IEEE, "ISO/IEC/IEEE 12207:2008: Information Technology – Software Life Cycle Processes," 2nd ed, 2008.

[3] T. DeMarco, "“The Paradox of Software Architecture and Design”, *Stevens Prize Lecture*," ed, Aug. 1999.

[4*] D. Budgen, *Software Design*, 2nd ed. New York: Addison-Wesley, 2003.

[5*] I. Sommerville, *Software Engineering*, 9th ed. New York: Addison-Wesley, 2010.

[6*] M. Page-Jones, *Fundamentals of Object-Oriented Design in UML*, 1st ed. Reading, MA: Addison-Wesley, 1999.

[7] "Merriam-Webster's Collegiate Dictionary," 11th ed. Springfield, MA: Merriam-Webster, 2003.

[8] I. 1016-2009, "*IEEE standard for Information Technology-Systems Design-Software Design Descriptions*," ed, 2009.

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

[10] G. K. e. al., "“Aspect-Oriented Programming,”" presented at the ECOOP ’97 — Object-Oriented Programming, 1997.

[11*] J. G. Brookshear, *Computer Science: An Overview*, 10th ed. Boston: Addison-Wesley, 2008.

[12*] J. H. Allen*, et al.*, *Software Security Engineering: A Guide for Project Managers*. Upper Saddle River, NJ: Addison-Wesley, 2008.

[13*] P. Clements*, et al.*, *Documenting Software Architectures: Views and Beyond*, 2nd Edition ed. Boston: Pearson Education, 2010.

[14*] E. Gamma*, et al.*, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Reading, MA: Addison-Wesley Professional, 1994.

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

[16*] J. Nielsen, *Usability Engineering*, 1st ed. Boston: Morgan Kaufmann, 1993.

[17] J. R. G. Booch, and I. Jacobson, "*The Unified Modeling Language User Guide*," ed: Addison-Wesley, 1999.

[18] R. S. Pressman, "*Software Engineering: A Practitioner's Approach(Seventh Edition)*," ed: the McGraw-Hill Companies, 2010.

[19] P. B. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software,* vol. 12, pp. 42-55, 1995.

[20] P. C. Len Bass, Rick Kazman, "Software Architecture in Practice (2nd Edition)," ed: Addison-Wesley Professional, 2003.