Software Design: An Overview

Guy Tremblay and Anne Pons

1. INTRODUCTION

According to the IEEE definition, *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" [IEE90]. Viewed as a process, *software design* is the software development 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—how the system is broken down and organized into components—and the interfaces between those components. It must also describe the components at a level of detail that enables their construction.

Software design plays an important role in developing a software system. During design, developers produce various models that form a kind of blueprint of the solution to be implemented. Developers can then analyze and evaluate these models to determine if they will fulfill the various requirements and to evaluate various alternative solutions and trade-offs. Finally, developers can use the resulting models to plan the subsequent development activities, in addition to using these models as input and starting point for construction and testing.

Before proceeding further, let us stress that not every aspect related to the "design" of software will be addressed in this overview. In DeMarco's terminology [DeM99], we will discuss mainly *D-design*—decomposition design, "a mapping of a system into its components pieces" [DeM99]. Because of its growing importance in the field of software architecture, we will also discuss, briefly, *FP-design*—family pattern design, whose goal is to establish exploitable commonalities over a family of systems. On the other hand, we will discuss neither *I-design*—invention design, "a conceptualization of a system to satisfy discovered needs and constraints" [DeM99], done by system analysts during requirements analysis and specification—nor *user inter-face design,* which is better done by specialists. The scope of the present overview thus reflects the scope of the Software Design Knowledge Area as presented in the *Guide to the Software Engineering Body of Knowledge* [AMBD04].

2. SOFTWARE DESIGN CONCEPTS

The concepts, notions, and terminology introduced in the first section form a basis for understanding the role and scope of software design. They are generally applicable to all software design methods and approaches.

2.1. General Design Concepts

Software is not the only field involving some form of design. In a general sense, design is a *problem-solving* activity; the inverse, however, is not necessarily true. For example, a person solving a crossword puzzle is not doing design, as opposed to the person who *designed* that crossword puzzle.

Design problems are generally characterized by a number of properties; for instance, there are usually many different feasible solutions and a specific solution can be considered *good or bad,* not *true or false.* Problems having these properties—together with additional ones, for example, formulating the problem often *is* the problem, the absence of specific rules to determine when the problem is indeed solved, the lack of test for evaluating a possible solution—have been called *wicked problems,* a notion initially introduced for planning problems [RW84], and subsequently applied to software design [PT76].

Design, in its general sense, can be understood in terms of five key concepts: goals, constraints, alternatives, representations, and solutions [SB93]. The design of a modern car can help illustrate these concepts. When designing a car, the general *goal* is to develop a means of transportation; of course, especially nowadays, marketing goals must also be taken into account. A large number of *constraints* clearly limit the possible solutions: type of engine and fuel, existing roads, environmental and safety regulations, target price, and so on. Based on these goals and constraints, the engineers, using their creativity and experience, then consider a number of *alternatives,* which are precursors to the final solution, not complete solutions yet. Starting from these various alternatives, appropriate *representations* are then developed in order to better understand the artifact and its

components, for example, blueprints of the car's engine and frame, sketches of the carossery, and so on. The descriptions of the acceptable alternatives that, together, make it possible to attain the target goals while satisfying the constraints then constitute a design *solution,* descriptions that will subsequently enable the car's construction.

2.2. Software Design Context

To understand the role of software design, its context must be understood, namely, the software development life cycle. The software development activities that are more directly coupled with software design are the following [ISO95]:

- Software requirements analysis, in which the intended use of the system to be developed is analyzed and the requirements (functional as well as nonfunctional ones) are specified. Software design then uses these requirements specification as input.
- Software coding and testing (also known as software construction), in which the software units identified by the software design activity are developed and (unit) tested.
- Software integration and qualification testing, in which the various software units and components identified during design and built during construction are combined together and tested to ensure that the initial requirements are satisfied.

In a software life cycle process, these activities, and others, are coupled with one another based on a *life cycle model,* of which there are two main types [Bud03]:

- *Linear models,* in which the process runs linearly through the activities; for example, the waterfall model.
- *Incremental models,* in which the process runs iteratively through the activities; for example, the spiral model or the iterative development approach.

Software development *methods* can also guide the software development process by offering procedures and guidelines to go through these various activities; see Section 6.

2.3. Software Design Process

In a standard listing of software life cycle processes such as ISO/IEC 12207, *Software Life Cycle Processes* [ISO95], software design consists of two activities that fit between software requirements analysis and software construction:

- *Software architectural design* (sometimes called top-level design) describes how the system is broken down and organized into components the software architecture [IEE00].
- *Software detailed design* describes the specific behavior of the various components identified by the software architecture.

The output of the design process is a set of models that records the major decisions that have been taken, and describes each of the software components and units sufficiently to enable their construction [IEE98, Pre01, Bud03].

3. SOFTWARE STRUCTURE AND ARCHITECTURE

In its usual sense, a software architecture defines the internal *structure.* According to the *Oxford English Dictionary,* the structure is "the way in which something is constructed or organized." For a software system that is its internal design. Since the mid-1990s, however, software architecture has taken on a broader meaning. For instance, IEEE Standard 1471 (*Recommended Practice for Architectural Descriptions of Software-Intensive systems*) [IEE00] defines *software architecture* as follows:

> The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.

Software architecture, in fact, has been emerging as a discipline on its own, involved with the study, in a generic way, of software structures and architectures [SG96]. This broader meaning of software architecture gave rise to a number of interesting ideas and concepts about software design at different levels of abstraction. Some of these concepts can be useful during architectural design (for example, architectural styles) whereas some pertain more specifically to detailed design (for example, design patterns). Other notions can also be useful for designing families of systems (also known as product lines). Interesting-

ly, most of these concepts can be seen as attempts to describe, and thus reuse, generic design knowledge. A key concept, though, is the notion of view (or viewpoint).

3.1. Architectural Structures and Views

A software architecture description is a complex entity as it serves many purposes, a key one being its use for communication among the various stakeholders involved in the development of the software system. Those various stakeholders—analysts, implementers, managers, testers, quality assurance team, and so on—have different roles and needs. Thus, different high-level facets, or perspectives, of a software design can and should be described and documented. These facets are called *views*. A view "represents a partial aspect of a software architecture that shows specific properties of a software system" [BMR+96]. It is "a collection of models that represent one aspect of an entire system" [MEH01].

The key role of views in design documentation was recognized as early as the mid-1980s in IEEE Standard 1016, *Recommended Practice for Software Design Description* (SDD) [IEE98],[1] whose purpose was to specify "the necessary information content, and recommended organization" for an SDD. IEEE Standard 1016 recommended that the overall organization of an SDD be obtained as the composition of a number of "design views," each containing a subset of the various attributes describing design entities: decomposition (how the system is partitioned into design entities); dependency (the relationships among entities and system resources); interface (what a designer, programmer, or tester needs to know to use the design entities); and detail description (internal design details).

Since then, various authors have proposed different sets of views for describing software architectures. A well-known approach, used within the Rational Unified Process (RUP) [Kru00], is Kruchten's "4+1 view model" [Kru95], consisting of the following views:

1. The *logical view* describes how the functional requirements are satisfied. It identifies the major design packages, subsystems, and classes.
2. The *implementation view* describes how the design is broken down into implementation units. It identifies the major software modules such as source code, data files, executables, and so on.
3. The *process view* addresses issues related to concurrency and distribution; for example, how the various threads of control are organized and distributed over the various programs, and how they interact.
4. The *deployment view* shows how the runtime units and components are distributed onto the various processing nodes.
5. The *use-case view,* which consists of a small number of use cases (see Section 6.3), ties together the other views, illustrating how they all work together.

Other sets of views have been proposed, for example, conceptual versus module interconnection versus execution versus code views [SNH95], and constructional (structural) versus behavioral versus functional versus data modeling views [Bud03].

More generally, according to Clements et al. [CBB+03], views can be classified into three categories, called *viewtypes:*

1. *Module viewtype.* These views describe the units of implementation, for example, classes, collections of classes, and layers.
2. *Component-and-connector viewtype.* These views describe the units of execution, that is, elements having a run-time presence; for example, processes, objects, clients, servers, and data stores.
3. *Allocation viewtype.* These views describe the relationships between a system and its development and execution environment, that is, the mapping of software units to elements of the environment; for example, hardware, file system, and development team.

The set of views selected to document a software architecture depends on various factors, a question discussed in slightly more detail in Section 5.2. Whatever the exact choice of views, the key idea is that a software architecture is a multifaceted artifact produced by the design process and composed of a set of relatively independent and orthogonal views.

3.2. Macro/Microarchitectural Patterns: Architectural Styles Versus Design Patterns

Over the last decade, starting with the seminal work of Gamma et al. [GHJV95], the notion of *pattern* has drawn a lot of attention. Described succinctly, a pattern is "a common solution to a common problem in a given context" [JBR99].

[1]The 1998 standard is, in fact, an updated version of the previous 1987 standard.

More precisely, the key idea behind patterns is that, over the years, software development practitioners have observed and identified a number of recurring problems and solutions. The key goal of patterns is then to describe—thus, to codify and document—those commonly recurring solutions to typical problems.

Patterns can be described and documented in various ways, ranging from informal textual descriptions [GHJV95, BMR+96] to more formal specifications [JTM00]. Because the goal is to make explicit thus codify—the associated design knowledge in order to make it transferable, pattern descriptions generally consist of a number of elements. For example, Buschmann et al. introduce a *system of patterns* in which each pattern is described by, among others, the following attributes [BMR+96]:

- The name of the pattern
- The context, that is, the key situations in which the pattern may apply
- An example illustrating the need for the pattern
- The general problem—its essence—that the pattern tries to solve
- The solution underlying the pattern, both the (static) structure of the pattern's elements and their run-time (dynamic) behavior
- Guidelines for the pattern's implementation

Additional descriptive elements may also be presented; for example, aliases, possible variants or related patterns, known uses, and consequences (advantages/disadvantages).

Patterns can be classified into three key major categories, depending on their scope and level of abstraction [BMR+96]:

1. Architectural styles
2. Design patterns
3. Coding idioms

In the following, we elaborate on the first two categories, the latter category being the domain of software construction.

Architectural Styles (Macroachitectural Patterns)

An *architectural style* has been defined as "a set of constraints on an architecture [that] define a set or family of architectures that satisfy them" [BCK03]. More precisely, an architectural style can be seen as a meta-model that provides a software system's high-level organization—its *macro*architecture:

> An *architectural [style]* expresses a fundamental structural organization schema for software systems. It provides a rich set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them. [BMR+96]

Various authors have identified a number of major architectural styles [BMR+96, BCK03, BRJ99, Bos00]:

- General structure (for example, layers, pipes and filters, blackboard)
- Distributed systems (for example, client–server, three-tiers, broker)
- Interactive systems (for example, model-view-controller, presentation-abstraction-control)
- Adaptable systems (for example, microkernel, reection)
- Other styles (for example, batch, interpreters, process control, rule-based)

The choice of a particular architectural style depends on the quality attributes that must be satisfied: whereas a given style may help attain certain quality attributes, it may also hinder others. Of course, heterogeneous styles are also possible.

Design Patterns (Microachitectural Patterns)

Although architectural styles can be viewed as patterns describing the high-level organization of software systems—the macroarchitecture—design patterns are used to describe details at a lower and more local level—the *micro*architecture:

> [Architectural styles and design patterns] are different in that a style *tends* to refer to a coarser grain of design solution than a pattern, which *tends* to refer to a design solution localized within a few (or one) of a system's many architectural components. [CBB+03]

Thus, whereas the application of an architectural style will generally have a significant impact on the general organization of the various components, applying a design pattern will usually have a much more limited and localized impact.

Although the notion of design pattern needs not be restricted to the object-oriented paradigm, most of the literature in fact describes *object-oriented design patterns.* Such patterns can be categorized in various ways. For instance, Gamma et al. use the following categories [GHJV95]:

- *Creational patterns* deal with the creation of objects (for example, builder, factory, prototype, singleton).
- *Structural patterns* deal with the composition of objects (for example, adapter, bridge, composite, decorator, façcade, flyweight, proxy).
- *Behavioral patterns* describe how objects interact (for example, command, interpreter, iterator, mediator, memento, observer, state, strategy, template, visitor).

On the other hand, Buschmann et al. classify patterns into the following categories [BMR+96]:

- *Structural decomposition* patterns address the "decomposition of subsystems and complex components into cooperating parts" (for example, whole–part).
- *Organization of work* patterns define "how components collaborate together to solve a complex problem" (for example, master–slave).
- *Access control* patterns define "guards and control access to services and components" (for example, Proxy).
- *Management* patterns handle "homogeneous collections of objects, services and components in their entirety" (for example, command processor, view handler).
- *Communication* patterns "help organize communication between components" (for example, forward–receiver, dispatcher–server, publisher–subscriber).

Given the large number of design patterns and styles described in the literature [GHJV95, BMR+96, Fow03], it is beyond the scope of this overview to give a detailed presentation of those various design patterns. Let us conclude, though, that a modern software designer should understand the key styles and patterns, as this will help avoid "reinventing the wheel" each time a new design problem is tackled, while establishing a common communication vocabulary among software developers.

3.3. Design of Families of Systems and Frameworks

An important goal of software design has always been to allow for the *reuse* of software elements. Recent approaches toward that goal are based on software product lines and software components. A *software product line* is "a collection of systems sharing a managed set of features constructed from a common set of core software assets" [BCK03]. A product line thus defines a *family* of systems and is based on and populated with *software components,* which are "unit[s] of composition with explicitly specified provided, required and configuration interfaces and quality attributes" [Bos00].

The detailed presentation of the principles and techniques underlying the design of software product lines is beyond the scope of the present paper. Let us simply indicate that building a common set of software assets involves identifying the key *commonalities* encountered among the various members of the possible family of products—done through *domain analysis* [McC97, WL99, Bos00]—as well as accounting for their possible *variabilities*—done by identifying and defining reusable and *customizable* components [McC97, Bos00].

Customization of components can be supported through a number of mechanisms, for example, inheritance, extension, configuration, template instantiation, and generation [Bos00]. In an object-oriented context, a related notion is that of *framework,* a partially complete software subsystem that can be extended by instantiating specific *plug-ins* (also known as *hot spots*) [Pre95, BMR+96, Bos00].

4. SOFTWARE DESIGN QUALITY ANALYSIS AND EVALUATION

According to ISO/IEC Standard 9126-1 [ISO01], software quality—defined as "the totality of features and characteristics of a software product or service that bear on its ability to satisfy stated or implied needs"—can be characterized by the following six properties: functionality, reliability, usability, efficiency, maintainability, and portability. Each of these may in turn be defined through an appropriate set of attributes [ISO01]. In the following, we briefly introduce some of the quality attributes applicable to design as well as some techniques to help attain those quality attributes.

4.1. Design Quality Attributes

A key distinction between the various quality attributes concerns whether their influence is observable or not at run time [BCK03]:

- *Run-time qualities* are observable only while the system is functioning; for example, functionality, usability, performance, reliability and availability, and security.
- *Development-time* qualities have an impact on the work of the development and maintenance teams, but are not directly observable at run time; for example, integrability, modifiability, portability, reusability, and testability.

Although some qualities can be achieved through appropriate *architectural choices*—for example, modifiability and reusability, performance—some others cannot—for example, functionality and usability [KB00]. An informal test, suggested by Kazman and Bass [KB00], to see if a particular quality attribute can be achieved through architectural choices is to ask the following question: "[C]an I improve [the] rating for that attribute by making structural changes?"

Parnas and Weiss, in their active design review approach [PW85], identify the key desirable properties of a design as being the following: it should be well structured, simple, efficient, adequate (satisfying the requirements), flexible (easy to change), practical (module interfaces sufficient for the job), implementable, and standardized (documentation organized in a standard way).

Another important quality attribute related with design concerns the architecture's intrinsic quality known as *conceptual integrity* [Bro95], which characterizes an architecture that "reflects one single set of design ideas," leading to simplicity, consistency, and elegance.

4.2. Measures

A number of *measures* can be defined to obtain *quantitative estimates* of a design's size, structure, or quality. Such measures generally depend on the selected design approach:

- *Function-oriented (structured) measures:* the design's structure, obtained through functional decomposition, is represented as a structure chart on which measures can be computed; for example, fan-in/fanout, cyclomatic complexity, integration complexity [MB89, Pre01].
- *Object-oriented measures:* the design structure is represented as class diagrams, on which measures can be computed; for example, weighted methods per class, depth of inheritance tree, number of children, coupling between object classes, responses for a class [CK94, Pre01].

4.3. Quality Analysis and Evaluation Tools

Although measures can be used to estimate certain quality attributes—for instance, complexity metrics can be used to evaluate the testability of a software unit and to determine how much testing needs to be performed—many quality attributes are hard to quantify. Thus, other techniques must be used to evaluate the quality of a design:

- *Software design reviews* are informal or semiformal, often group-based, techniques used to verify the quality of design artifacts; for example, architecture reviews [BCK03], design reviews and inspections [PW85, Bud03], scenario-based architecture evaluation [BCK03, Bos00], and requirements tracing [TD02].
- *Simulation and prototyping* are dynamic techniques used to evaluate a design; for example, simulation-based performance or reliability analysis [BCK03, KB00, Bos00], and feasibility prototyping [BCK03, Bos00].

5. SOFTWARE DESIGN NOTATIONS AND DOCUMENTATION

Many different notations exist to represent software design artifacts, for instance, 18 different kinds of notations are mentioned in the Software Design Knowledge Area of the Guide to the SWEBOK [AMBD04]. Some notations are used mostly during architectural design, whereas others mainly during detailed design, although some can be used in both phases. Some notations are also used mostly within specific design methods, whereas others are more widely used.

Budgen [Bud03] categorizes the various design notations in terms of black box versus white box: as a *black-box* notation "is concerned with the *external* properties of the elements of a design model," whereas a *white-box* notation "is largely concerned with describing some aspect of the detailed realization of a design element" [Bud03].

An alternative characterization, which we use below to present briefly a small number of notations, is to distinguish between notations for describing *structural* (static) properties—a design's structural organization—and those for describing *behavioral* (dynamic) properties—the behavior of the software components.

5.1. A Selection of Design Notations

Over the last few years, UML (Unified Modeling Language) [BRJ99] has become an almost de facto standard for software development notations. In what follows, we briefly present a (small) selection of software design notations; notations whose name appear in italics are part of UML. Structural descriptions (static view) These 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):

- *Class and object diagrams.* These are used to represent a set of classes (and objects) and their relationships [BRJ99]. These diagrams are used in object-oriented design. A related, although somewhat older, notation is entity-relationship diagrams (ERDs), used to represent conceptual models of data stored in information systems [Mar94, TD02].
- *Component diagrams.* These are used to model the static implementation view of a system, that is, physical things (and their relationships) such as executables, libraries, tables, files, and documents [BRJ99]. Although their main use is during construction, such diagrams can also be used during design; for example, to document the module (work assignment) structure [BCK03].
- *Deployment diagrams.* These are used to model the static deployment view of a system, that is, "the configuration of run time processing nodes and the components that live on them" [BRJ99]. Typically, such diagrams can be used to represent distribution aspects, for example, to model embedded, client/server or distributed systems.
- Structure charts. These are used to describe the calling structure of programs (which procedure or module calls/is called by which other) [PJ88, Pre01, Bud03]. Such diagrams are at the heart of the structured (functionoriented) design approach.
- Structure (Jackson) diagrams. These are used to describe the data structures manipulated by a program in terms of sequence, selection and iteration [Mar94, Bud03]. These diagrams were initially introduced in JSP (Jackson Structured Programming) [Jac75].

Behavioral Descriptions (Dynamic View)

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

- *Activity diagrams.* These are used to show the control flow from activity ("ongoing nonatomic execution within a state machine") to activity [BRJ99]. These diagrams are related to the older owcharts [Pre01].
- *Interaction diagrams.* These are used to show the interactions among a group of objects [BRJ99]. These diagrams come in two flavors: *sequence diagrams* put the emphasis on the time-ordering of messages, whereas *collaboration diagrams* put the emphasis on the objects, their links, and the messages they exchange on these links.
- Data flow diagrams (DFDs). These are used to show the data flow among a set of processes [PJ88, Pre01, Bud03]. These diagrams were introduced and used by the structured analysis and design approach [YC79].
- State transition diagrams and *statechart diagrams.* These are used to show the control flow from state to state in a state machine [BRJ99, Bud03].
- Pseudocode and program design languages (PDLs). These are structured, programming-like languages used to describe, generally at the detailed design stage, the behavior of a procedure or method [Pre01, Bud03].

5.2. Design Documentation

Given the variety of notations available for design, a key question is how these various notations can be combined to obtain a coherent design document. There is no clearcut answer to this question, as it depends on many aspects, for instance, the type of software, the software development method being used, the organization in which/for which the software is developed, the stakeholders involved, and so on. A key practice, though, is the use of views, introduced in Section 3.

The selection of an appropriate set of views strongly depends on the stakeholders involved: project managers, developers, testers and integrators, customers, end users, and so on, all have different needs. Satisfying these different needs can best be described in terms of the relative importance of the various views from each viewtype (module, component-and-connector,

and allocation; see Section 3.1) [CBB+03]. For instance, a project manager would need detailed allocation views, whereas a developer would need mostly detailed module and component-and-connector views.

Documenting a view involves, among other things, describing the *interfaces* of the elements from that view. How such an interface is defined will depend on the type of element. A key characteristic of any interface specification, though, is that it should be a *two-way* description: what the element *provides* and what it *requires*—the resources used by the element, and the assumptions it makes from the environment. Clements et al. [CBB+03] offer a good presentation of these ideas.

Another key idea, formulated initially by Parnas and Clements, is that a design should be presented and documented in a *rational* way [CP86, Cle00], even though the process that lead to this design may not have been perfectly rational. As an analogy, consider the presentation of a major discovery that need not follow the process that led to that discovery (often by trial and error). In addition, even though this part is not strictly *rational,* the *rationale* behind the key decisions should also be recorded; for instance, the design alternatives that were considered and rejected should be described.

6. SOFTWARE DESIGN STRATEGIES AND METHODS

Various general principles and *strategies* have been proposed to guide the design process and help improve the quality of the resulting software [Mar94, BMR+96, Bud03]. In contrast with strategies, *methods* are more specific in that they generally suggest a particular set of *notations* together with a description of a *process* to be followed when designing software, as well as *heuristics* that provide guidance in adapting the method to a particular context [Bud03]. Such methods, which generally incorporate, in various ways, the general design principles and strategies, can help improve the quality of the resulting software when applied in a proper context. They are also useful as a means for transferring knowledge and as a common framework for teams of developers.

6.1. General Strategies and Enabling Techniques

General software design strategies can be described in terms of *enabling techniques,* a notion introduced by Buschmann et al. to denote fundamental principles and techniques of software design which are "independent of [any] specific software development method, and [. . .] have been known for years" [BMR+96]:

- *Abstraction.* Abstraction is "the process of forgetting information so that things that are different can be treated as if they were the same" [LG01]. Two key mechanisms are *abstraction by parameterization*—abstract from specific data by introducing parameters—and *abstraction by specification*—abstract how a module is implemented by referring to an appropriate specification. These mechanisms lead to three major kinds of abstraction: procedural abstraction (to introduce new operations), data abstraction (to introduce new data types), and control (iteration) abstraction (to iterate over collections of elements).
- *Coupling and cohesion.* Coupling is defined as the strength of the relationships *between* software components, whereas cohesion is defined by how the elements making up a component are related [BCK03, Pre01]. As a general rule, coupling between components should be weak, whereas the (internal) cohesion of a component should be high. Although these concepts were initially introduced for structured design [YC79], they also apply to object-oriented design [PJ00].
- *Divide and conquer.* In an algorithmic sense, divide and conquer is a technique that solves a complex problem by dividing it into two or more simpler problems, which are then solved recursively and whose solutions are subsequently combined to obtain the solution to the initial problem. In a function-oriented sense, divide and conquer involves breaking down a complex problem or task into simpler subproblems or subtasks that can be solved independently, a strategy at the root of *stepwise refinement* [Wir71, Bud03]. A related strategy is the *separation of concerns,* which suggests that "different or unrelated responsibilities should be separated from each other" [BMR+96].
- *Information hiding and encapsulation.* Information hiding is a general design strategy introduced by Parnas in which "every module [. . .] is characterized by its knowledge of a design decision which it hides from all others" [Par72]. A key principle associated with information hiding is the *separation of interface and implementation,* wherein the "interface or definition [of a module is] chosen to reveal as little as possible about its inner workings" [Par72]. In other words, a public interface (known to the clients) is specified, separate from the details of how the component is realized. Another related notion is *encapsulation,* defined as "the grouping of related ideas into one unit, which can thereafter be referred to by a single name" [PJ00]. Thus, encapsulation combines elements to create a new entity, whose internal details are hidden; in other words, encapsulation creates a new abstraction.
- *Sufficiency, completeness,* and *primitiveness.* These notions pertain to the idea that a software component should capture all the important characteristics of an abstraction needed to interact with it, and nothing more [BMR+96, LG01].

6.2. Function-oriented (Structured) Design

Structured (function-oriented) design [YC79, PJ88, Pre01, Bud03] is one of the early software design paradigms, in which decomposition centers on identifying the major systems *functions,* which are then elaborated and refined in a top-down manner, that is, using a divide-and-conquer approach based on functional decomposition.

Structured design is generally performed after structured analysis. A typical structured analysis [You89] produces, among other things, data flow diagrams (DFD) of the various system functions together with associated process descriptions, that is, descriptions of the processing performed by each subtask, usually using informal pseudocode. Entity-relationship diagrams describing the data stores can also be used.

Two key strategies have been proposed to help derive a software architecture, represented as a structure chart, from a DFD:

1. *Transaction analysis.* A *transaction* is characterized by some *event* in the environment that generates a *stimulus* to the system, which in turn triggers some *system's activity* that produces a *response* having an *effect* upon the environment. Transaction analysis consists in identifying the key transaction types of a system and using them as the units of design, that is, designing separately the processing of each transaction type.
2. *Transformation analysis.* The key step in deriving a structure chart from a DFD (for a given transaction) is to identify the *central transform,* that is, "the portion of the DFD that contains the essential functions of the system and is independent of the particular implementation of the input and output" [PJ88]. A *first-cut* (draft) structure chart can then be obtained by lifting the bubbles associated with the central transform, promoting them at the top level of the structure chart, as illustrated in Figure 1. Remember that a structure chart is a *hierarchical* diagram that shows the calls or is called the relationships. Of course, this initial structure chart will have to be revised and completed, in line with the quality criteria of cohesion and coupling as well as with various heuristics. Other details may also need to be revised or added; for example, error handling modules, initialization and termination details, required control flags, and so on.

Key concepts of structured design are those of *coupling* and *cohesion,* which characterize a design of good *quality.* For instance, a good design should restrict the coupling between modules to *normal* types of coupling—data, stamp, and control coupling, data coupling being the preferred form, where communication between modules is through parameters, where each parameter is an elementary piece of data—and should avoid other *pathological* forms of coupling—namely, *common* and *content* coupling

Similarly, a good design should give preference to modules having high cohesion; more precisely, modules exhibiting *functional cohesion* when the module "contains elements that all contribute to the execution of one and only one problem-related task" [PJ88]. Other, weaker, types of cohesion have been identified, from more cohesive to less cohesive: sequential and com-

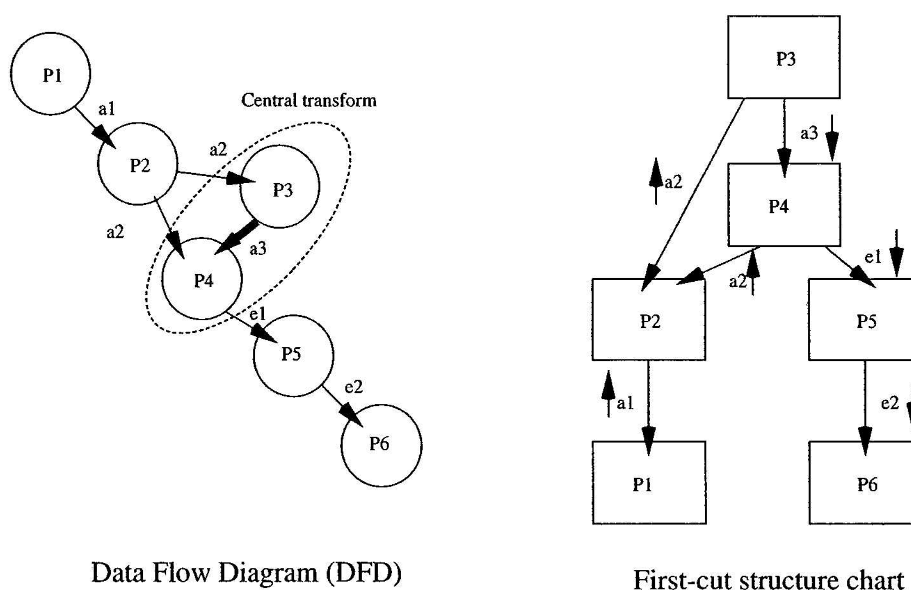

Figure 1. Using transform analysis to derive a structure chart from a DFD.

municational, procedural and temporal, logical and coincidental. Those weaker forms of cohesion can sometimes be acceptable, although the less cohesive ones should preferably be avoided.

Additional heuristics have also been suggested to help improve the quality of the resulting design:

- *Fan-in/fan-out.* A high fan-in—the number of modules that call a given module *M*—is considered good, as it indicates reuse of *M*. On the other hand, a low to moderate fan-out (maximum 5–7)—the number of modules that *M* calls—is generally preferable.
- *Decision splitting.* Decision splits, which occur when the recognition of a condition and the execution of the associated action are not kept within the same module, should be avoided.
- *Balanced systems.* A balanced system, when the top-level modules deal with logical and abstract data (clean and valid data, independent of implementation format), is preferable.

Structured design, being one of the first well-described and well-known design methods, made important contributions to the field of software design. Its integration with an appropriate analysis method—structured analysis [You89]—was also one of its key strengths. With the emergence of object-oriented languages and programming, though, structured design, with its emphasis on functional decomposition, started to reach its limits.

6.3. Object-oriented Design

The notion of *object* is intimately tied to the notions of data abstraction, encapsulation, and abstract data type (ADT). More precisely, an object is described by the following characteristics [Mac82, Boo86]: an object can be created/destroyed, has a unique (immutable) *identity,* possesses a (mutable) *state* (i.e., evolves in time), and exhibits some well-defined *behavior* through services it offers. Objects are generally organized into *classes,* which describe collections of objects sharing the same structure and behavior, thus the link with data types.[2]

Over the years, numerous software design methods based on objects, collectively known as *object-oriented design (OOD) methods,* have been proposed [Boo86, Boo94, WBWW90, CY91, JBP+91]. Early approaches, in which objects were mostly similar to entities in entity-relationship modeling and inheritance was not used [Abb83, Boo86], were said to be *object-based.* Later approaches, in which inheritance and polymorphism play a key role, are said to be *object-oriented* (*OO*).

OO design methods aim at developing software systems composed of interacting objects that are highly modular and, thus, easy to modify, extend, and maintain. OO design models address structural (static) aspects—classes and objects, their relationships and their grouping as well as behavioral (dynamic) ones—objects' behavior and interactions. The notations used for documenting these models take various forms; for example, diagrammatic, textual, and even mathematical. Much like structured design was intimately tied with structured analysis, existing OO design methods are generally associated with OO analysis methods. Contrary to structured analysis and design, though, many of the notations used for OO design can also be used during requirements analysis, leading to some degree of *seamlessness* between the two. This seamlessness, however, must not make one forget that requirements analysis and design do deal with different concerns; put succinctly, problem domain versus solution domain.

The Unified Modeling Language (UML), because it evolved from the integration of a number of OO methods, provides a wide variety of notations for OO analysis and design. UML includes various notations in addition to those mentioned in Section 5.1, for example, real-time modeling, formal specification using the Object Constraint Language (OCL) [WK99], and so on. UML is not an OO design method, though; UML is simply a set of notations, neutral with respect to any specific design method. On the other hand, the Unified Process (UP), elaborated by the same people who developed UML [JBR99], does define a software development process that incorporates OO analysis and design.

The Unified Process consists of four *phases:* inception, elaboration, construction, and transition. Each phase, delimited by an appropriate *milestone,* consists of one or more *iterations,* where each iteration generally results in an executable release and involves a number of core *workows,* namely, requirements, analysis, design, implementation, and test. From a software design perspective, the important phases are the elaboration and construction phases, since the *architectural baseline* (i.e., the software architecture description; see the "4+1 view model" described in Section 3.1) is developed during the elaboration phase whereas most of the detailed design is developed during the construction phase.

The key input to the design workow is a collection of use cases that describe the functional requirements a use case is "a description of a set of sequences of actions [. . .] that a system performs to yield an observable result [. . .]" [BRJ99]—together with the applicable non-functional requirements. The output of the design workflow is a *design model* consisting of classes and their collaborations, possibly organized into packages and subsystems, that provide the intended behavior while satisfying the non-functional requirements.

[2]However, note that an ADT does not necessarily define a class of objects. See [Mac82] for a clear exposition of the notions of *values* versus *objects.*

A class diagram models a set of classes and their relationships, for example, association, aggregation, inheritance, dependency, and so on. During design, class diagrams play a central role as they identify the major kinds of objects that will cooperate to produce the system behavior. The analysis model also contains class diagrams. Although these latter classes can provide a starting point for identifying some of the design model classes, there is not necessarily a direct correspondence between the two sets of classes; while the analysis classes describe the system intended behavior—the *what?,* the *black-box* view—the design classes instead describe how this behavior is obtained—the *how?,* the *white-box* view.

Figure 2 shows a simple UML class diagram for bank accounts. Two different kinds of account are available: `SavingAccount` and `CheckingAccount`. Both are specializations, indicated by an *inheritance* relationship, of a general `BankAccount`. A Customer owns a bank account, in fact, can own multiple bank accounts (the "*" annotation). Since this is a class diagram for the design model, some methods have been indicated.

How objects from the various classes collaborate to provide the desired system behavior is described using *interaction diagrams.* As mentioned in Section 5.1, interaction diagrams come in two flavors: sequence diagrams and collaboration diagrams. Figure 3 shows a simple collaboration diagram, which involves objects (underlined names), not classes. The example illustrates how the indicated objects collaborate to perform a `transferTo` operation, associated with the `BankAccount`.class. Each arrow indicates a message being sent (a method being called), conditionally, in the cases of messages 2 and 3. The numbering of the messages indicates their time ordering.

Although interaction diagrams describe how a number of objects collaborate to realize a given operation, in reaction to a specific event, they do not describe the behavior of a specific class of objects in reaction to all possible events. Such class specific behavior can be described using a *statechart* diagram, a generalized form of state transition diagram. A statechart diagram thus describes an object behavior from an internal viewpoint—how the internal state of an object changes in reaction to the various events.

The early OO methods focused mostly on data abstraction and ADTs, viewing primarily objects through their components and static structural relationships, an approach called "data-driven design" [WBW89]. The Unified Process design approach, as do many modern OOD methods, instead focuses on properly identifying and assigning responsibilities to classes and objects, an approach named "responsibility-driven," initially introduced by Wirfs-Brock et al. [WBW89, WBWW90]. More precisely, a *responsibility* is defined as "a contract or an obligation of a class" [BRJ99], or "an obligation to perform a task or know information" [WBM03]. Although such contracts can be defined formally, for example, using formal pre/postconditions as is done in Meyer's Design by Contract (DBC) approach [Mey92], even informal descriptions can be useful [LG01]. Responsibility-driven design generally improves encapsulation, produces a less complex design with improved coupling and cohesion [SC93], and tends to produce systems in which the overall control is better organized and balanced, thus more modular.

The growth of the OO approach over the last two decades has been phenomenal: the rise from OO programming, to design, then to analysis; the development of a *lingua franca* (UML); the design patterns movement; and so on. For lack of space, however, several aspects related with OO design have not been addressed; for example, persistence, error handling, component-based design, and so on. Once the key ideas of OOD are understood, a next important step, as mentioned in Section 3.2, is to address the vast literature on OO design patterns.

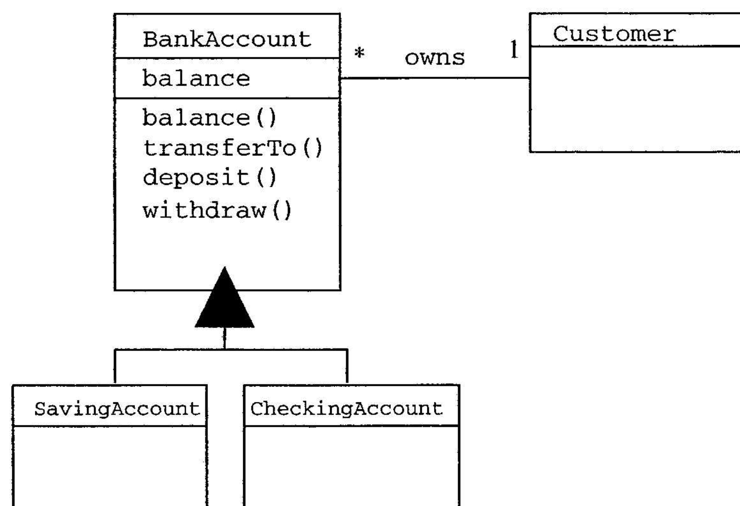

Figure 2. A UML class diagram for bank accounts.

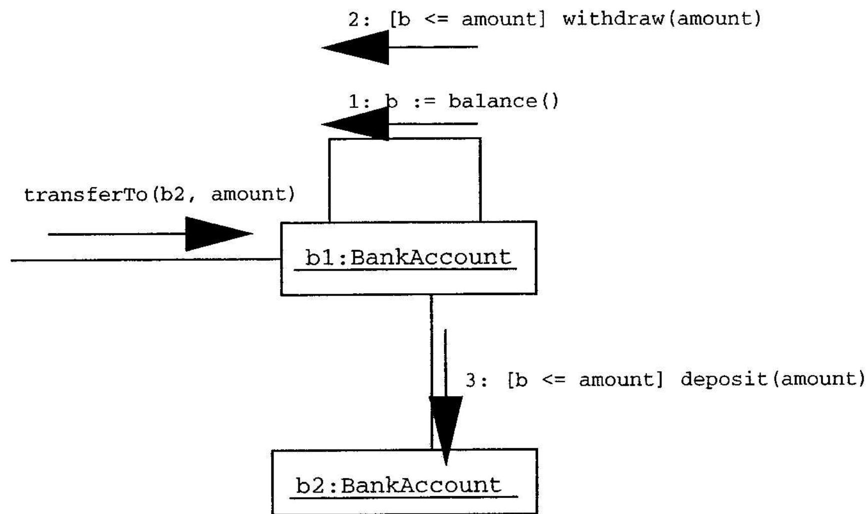

Figure 3. A UML collaboration diagram for a `transferTo` operation.

6.4. Data-structure-oriented Design

Data-structure-oriented design—also known as Jackson Structured Programming (JSP) [Jac75] is an approach in which the emphasis is on the data that a program manipulates rather than the functions it performs [Cam89, Pre01, Bud03]. This emphasis on data is motivated by the fact that such data is generally more *stable* (less subject to change) than the functions that need to be performed.

In JSP, the designer first describes the input and output data—for instance, using Jackson structure diagrams—and then develops the program's control structure by establishing an appropriate correspondence between the input and output data structure diagrams. Once the program control structure is properly defined, appropriate program actions and conditions are then added to obtain the final program. A number of heuristics have also been proposed to deal with special cases; for example, how to deal with various kinds of mismatches (also known as *structure clashes*) between the input and output structures by using program inversion.

JSP's scope was mostly restricted to the design of data-processing programs using sequential (batch-style) files and processes. Jackson later introduced the JSD method (Jackson System Development) [Jac83] that deals with the analysis and design of systems composed of more complex interacting processes involving various entities performing actions, an approach similar, in certain ways, to object-oriented design.

7. CONCLUSION

Software design is a rich and still evolving field, so this overview could only scratch the surface. In fact, following the spirit of the *Guide to the Software Engineering Body of Knowledge,* which aims at presenting *generally accepted knowledge*—"knowledge and practices [that] are applicable to most projects most of the time, and [such] that there is a widespread consensus about their value and usefulness" [AMBD04]—this overview should be considered as a *guide* to the field of software design. Further detail on the topics discussed in this overview or on more specialized topics related with design (for example, design of real-time or distributed systems) can be obtained by consulting the various references mentioned in the next section or in the accompanying papers.

REFERENCES

[Abb83] R. Abbott. Program design by informal english description. *CACM, 26*(11):882–894, 1983.

[AMBD04] A. Abran, J. W. Moore, P. Bourque, and R. Dupuis, editors. *Guide to the Software Engineering Body of Knowledge* (2004 Version). IEEE Computer Society Press, 2004.

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice* (2nd ed.). SEI Series in Software Engineering. Addison-Wesley Professional, 2003.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture—A System of Patterns.* Wiley, 1996.

[Boo86] G. Booch. Object-oriented development. *IEEE Transactions on Software Engineering, SE-12*(2):211–221, Feb. 1986.

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

[Bos00] J. Bosch. *Design and Use of Software Architectures—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.

[Bro95] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1995.

[Bud03] D. Budgen. *Software Design* (2nd ed.). Pearson Education/Addison-Wesley, 2003.

[Cam89] J. Cameron. *JSP and JSD: The Jackson Approach to Software Development* (2nd ed.). IEEE Computer Society Press, 1989.

[CBB+03] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond.* SEI Series in Software Engineering. Addison-Wesley, 2003.

[CK94] S. R. Chimader and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20*(6):476–493, June 1994.

[Cle00] P. C. Clements. Rationalize your design. In P. C. Clements, editor, *Constructing Superior Software,* chapter 5, pp. 105–124. McMillan Technical Publishing, 2000.

[CP86] P. Clements and D. L. Parnas. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering, SE-12*(2):251–257, 1986.

[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.

[Fow03] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2003.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[IEE90] IEEE. IEEE standard glossary of software engineering terminology. Technical Report IEEE Std 610.12-1990, IEEE, New York, 1990.

[IEE98] IEEE. An american national standard—IEEE recommended practice for software design descriptions. Technical Report IEEE Std 1016-1998, IEEE, New York, 1998.

[IEE00] IEEE. Recommended practice for architectural description of software-intensive systems. Technical Report IEEE Std 1471-2000, IEEE, New York, 2000.

[ISO95] ISO/IEC. Information technology—software life cycle processes. Technical Report ISO/IEC Std 12207: 1995, ISO/IEC, 1995.

[ISO01] ISO/IEC. Software engineering—product quality—part I: Quality model. Technical Report ISO/IEC 9126.1-2001, ISO/IEC, 2001.

[Jac75] M. A. Jackson. *Principles of Program Design.* Academic Press, 1975.

[Jac83] M. A. Jackson. *System Development.* Prentice-Hall, 1983.

[JBP+91] Rumbaugh J., 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.

[JTM00] J.-M. Jézéquel, M. Train, and C. Mingins. *Design Patterns and Contracts.* Addison-Wesley, 2000.

[KB00] R. Kazman and L. Bass. Software architecture and quality. In P. C. Clements, editor, *Constructing Superior Software,* chapter 4, pages 83–104. McMillan Technical Publishing, 2000.

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

[Kru00] P. B. Kruchten. *The Rational Unified Process: An Introduction* (2nd ed.). Addison-Wesley, 2000.

[LG01] B. Liskov and J. Guttag. *Program Development in Java.* Addison-Wesley, 2001.

[Mac82] B. J. MacLennan. Values and objects in programming languages. *SIGPLAN Notices, 17*(12):70–79, 1982.

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

[MB89] T. McCabe and C. W. Butler. Design complexity measurement and testing. *CACM, 32*(12):1415–1425, 1989.

[McC97] C. McClure. *Software Reuse Techniques: Adding Reuse to the System Development Process.* Prentice-Hall, Inc., 1997.

[MEH01] M. W. Maier, D. Emery, and R. Hilliard. Software architecture: Introducing IEEE Standard 1471. *IEEE Computer, 34*(4):107–109, April 2001.

[Mey92] B. Meyer. Applying "design by contract". *IEEE Computer, 25*(10):40–51, 1992.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM, 15*(12):1053–1058, 1972.

[PJ88] M. Page-Jones. *The Practical Guide to Structured Systems Design* (2nd ed.). Prentice-Hall, 1988.

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

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

[Pre01] R. S. Pressman. *Software Engineering—A Practitioner's Approach* (5th ed.). McGraw-Hill, 2001.

[PT76] L. J. Peters and L. L. Tripp. Is software design "wicked"? *Datamation, 22*(5):176, 1976.

[PW85] D. L. Parnas and D. M. Weiss. Active design reviews: Principles and practices. In *Proceedings of the Eigth International Conference on Software Engineering,* pp. 215–222, 1985.

[RW84] H. W. J. Rittel and M. W. Webber. *Developments in Design Methodology,* pages 136–144. Wiley., 1984.

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

[SC93] R. C. Sharble and S. S. Cohen. The object-oriented brewery: A comparison of two object-oriented development methods. *ACM SIGSOFT Software Engineering Notes, 18*(2):60–73, 1993.

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

[SNH95] D. Soni, R. Nord, and C. Hofmeister. Software architecture in industrial applications. In *Proceedings of the 17th International Conference on Software Engineering,* pp. 196–207, Seattle, WA, 1995. ACM Press.

[TD02] R. H. Thayer and M. Dorman, editors. *Software Engineering—Volume 1: The Development Process* (2nd ed.). IEEE Computer Society Press, 2002.

[WBM03] R. Wirfs-Brock and A. McKean. *Object Design—Roles, Responsibilities, and Collaborations.* Addison-Wesley, 2003.

[WBW89] R. Wirfs-Brock and B. Wilkerson. Object-oriented design: A responsibility-driven approach. In *Proceedings of OOPSLA '89,* pp. 71–75, New Orleans, LA, 1989.

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

[Wir71] N. Wirth. Program development by stepwise refinement. *CACM, 14*(4):221–227, 1971.

[WK99] J. Warmer and A. Kleppe. *The Object Constraint Language-Precise Modeling with UML.* Addison-Wesley, 1999.

[WL99] D. J. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999.

[YC79] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Programming and Systems Development.* Prentice-Hall, 1979.

[You89]. E. Yourdon. *Modern Structured Analysis.* Yourdon Press, 1989.

components, for example, blueprints of the car's engine and frame, sketches of the carossery, and so on. The descriptions of the acceptable alternatives that, together, make it possible to attain the target goals while satisfying the constraints then constitute a design *solution,* descriptions that will subsequently enable the car's construction.

2.2. Software Design Context

To understand the role of software design, its context must be understood, namely, the software development life cycle. The software development activities that are more directly coupled with software design are the following [ISO95]:

- Software requirements analysis, in which the intended use of the system to be developed is analyzed and the requirements (functional as well as nonfunctional ones) are specified. Software design then uses these requirements specification as input.
- Software coding and testing (also known as software construction), in which the software units identified by the software design activity are developed and (unit) tested.
- Software integration and qualification testing, in which the various software units and components identified during design and built during construction are combined together and tested to ensure that the initial requirements are satisfied.

In a software life cycle process, these activities, and others, are coupled with one another based on a *life cycle model,* of which there are two main types [Bud03]:

- *Linear models,* in which the process runs linearly through the activities; for example, the waterfall model.
- *Incremental models,* in which the process runs iteratively through the activities; for example, the spiral model or the iterative development approach.

Software development *methods* can also guide the software development process by offering procedures and guidelines to go through these various activities; see Section 6.

2.3. Software Design Process

In a standard listing of software life cycle processes such as ISO/IEC 12207, *Software Life Cycle Processes* [ISO95], software design consists of two activities that fit between software requirements analysis and software construction:

- *Software architectural design* (sometimes called top-level design) describes how the system is broken down and organized into components the software architecture [IEE00].
- *Software detailed design* describes the specific behavior of the various components identified by the software architecture.

The output of the design process is a set of models that records the major decisions that have been taken, and describes each of the software components and units sufficiently to enable their construction [IEE98, Pre01, Bud03].

3. SOFTWARE STRUCTURE AND ARCHITECTURE

In its usual sense, a software architecture defines the internal *structure.* According to the *Oxford English Dictionary,* the structure is "the way in which something is constructed or organized." For a software system that is its internal design. Since the mid-1990s, however, software architecture has taken on a broader meaning. For instance, IEEE Standard 1471 (*Recommended Practice for Architectural Descriptions of Software-Intensive systems*) [IEE00] defines *software architecture* as follows:

> The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.

Software architecture, in fact, has been emerging as a discipline on its own, involved with the study, in a generic way, of software structures and architectures [SG96]. This broader meaning of software architecture gave rise to a number of interesting ideas and concepts about software design at different levels of abstraction. Some of these concepts can be useful during architectural design (for example, architectural styles) whereas some pertain more specifically to detailed design (for example, design patterns). Other notions can also be useful for designing families of systems (also known as product lines). Interesting-

ly, most of these concepts can be seen as attempts to describe, and thus reuse, generic design knowledge. A key concept, though, is the notion of view (or viewpoint).

3.1. Architectural Structures and Views

A software architecture description is a complex entity as it serves many purposes, a key one being its use for communication among the various stakeholders involved in the development of the software system. Those various stakeholders—analysts, implementers, managers, testers, quality assurance team, and so on—have different roles and needs. Thus, different high-level facets, or perspectives, of a software design can and should be described and documented. These facets are called *views*. A view "represents a partial aspect of a software architecture that shows specific properties of a software system" [BMR+96]. It is "a collection of models that represent one aspect of an entire system" [MEH01].

The key role of views in design documentation was recognized as early as the mid-1980s in IEEE Standard 1016, *Recommended Practice for Software Design Description* (SDD) [IEE98],[1] whose purpose was to specify "the necessary information content, and recommended organization" for an SDD. IEEE Standard 1016 recommended that the overall organization of an SDD be obtained as the composition of a number of "design views," each containing a subset of the various attributes describing design entities: decomposition (how the system is partitioned into design entities); dependency (the relationships among entities and system resources); interface (what a designer, programmer, or tester needs to know to use the design entities); and detail description (internal design details).

Since then, various authors have proposed different sets of views for describing software architectures. A well-known approach, used within the Rational Unified Process (RUP) [Kru00], is Kruchten's "4+1 view model" [Kru95], consisting of the following views:

1. The *logical view* describes how the functional requirements are satisfied. It identifies the major design packages, subsystems, and classes.
2. The *implementation view* describes how the design is broken down into implementation units. It identifies the major software modules such as source code, data files, executables, and so on.
3. The *process view* addresses issues related to concurrency and distribution; for example, how the various threads of control are organized and distributed over the various programs, and how they interact.
4. The *deployment view* shows how the runtime units and components are distributed onto the various processing nodes.
5. The *use-case view*, which consists of a small number of use cases (see Section 6.3), ties together the other views, illustrating how they all work together.

Other sets of views have been proposed, for example, conceptual versus module interconnection versus execution versus code views [SNH95], and constructional (structural) versus behavioral versus functional versus data modeling views [Bud03].

More generally, according to Clements et al. [CBB+03], views can be classified into three categories, called *viewtypes:*

1. *Module viewtype.* These views describe the units of implementation, for example, classes, collections of classes, and layers.
2. *Component-and-connector viewtype.* These views describe the units of execution, that is, elements having a run-time presence; for example, processes, objects, clients, servers, and data stores.
3. *Allocation viewtype.* These views describe the relationships between a system and its development and execution environment, that is, the mapping of software units to elements of the environment; for example, hardware, file system, and development team.

The set of views selected to document a software architecture depends on various factors, a question discussed in slightly more detail in Section 5.2. Whatever the exact choice of views, the key idea is that a software architecture is a multifaceted artifact produced by the design process and composed of a set of relatively independent and orthogonal views.

3.2. Macro/Microarchitectural Patterns: Architectural Styles Versus Design Patterns

Over the last decade, starting with the seminal work of Gamma et al. [GHJV95], the notion of *pattern* has drawn a lot of attention. Described succinctly, a pattern is "a common solution to a common problem in a given context" [JBR99].

[1]The 1998 standard is, in fact, an updated version of the previous 1987 standard.

More precisely, the key idea behind patterns is that, over the years, software development practitioners have observed and identified a number of recurring problems and solutions. The key goal of patterns is then to describe—thus, to codify and document—those commonly recurring solutions to typical problems.

Patterns can be described and documented in various ways, ranging from informal textual descriptions [GHJV95, BMR+96] to more formal specifications [JTM00]. Because the goal is to make explicit thus codify—the associated design knowledge in order to make it transferable, pattern descriptions generally consist of a number of elements. For example, Buschmann et al. introduce a *system of patterns* in which each pattern is described by, among others, the following attributes [BMR+96]:

- The name of the pattern
- The context, that is, the key situations in which the pattern may apply
- An example illustrating the need for the pattern
- The general problem—its essence—that the pattern tries to solve
- The solution underlying the pattern, both the (static) structure of the pattern's elements and their run-time (dynamic) behavior
- Guidelines for the pattern's implementation

Additional descriptive elements may also be presented; for example, aliases, possible variants or related patterns, known uses, and consequences (advantages/disadvantages).

Patterns can be classified into three key major categories, depending on their scope and level of abstraction [BMR+96]:

1. Architectural styles
2. Design patterns
3. Coding idioms

In the following, we elaborate on the first two categories, the latter category being the domain of software construction.

Architectural Styles (Macroachitectural Patterns)

An *architectural style* has been defined as "a set of constraints on an architecture [that] define a set or family of architectures that satisfy them" [BCK03]. More precisely, an architectural style can be seen as a meta-model that provides a software system's high-level organization—its *macro*architecture:

> An *architectural [style]* expresses a fundamental structural organization schema for software systems. It provides a rich set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them. [BMR+96]

Various authors have identified a number of major architectural styles [BMR+96, BCK03, BRJ99, Bos00]:

- General structure (for example, layers, pipes and filters, blackboard)
- Distributed systems (for example, client–server, three-tiers, broker)
- Interactive systems (for example, model-view-controller, presentation-abstraction-control)
- Adaptable systems (for example, microkernel, reection)
- Other styles (for example, batch, interpreters, process control, rule-based)

The choice of a particular architectural style depends on the quality attributes that must be satisfied: whereas a given style may help attain certain quality attributes, it may also hinder others. Of course, heterogeneous styles are also possible.

Design Patterns (Microachitectural Patterns)

Although architectural styles can be viewed as patterns describing the high-level organization of software systems—the macroarchitecture—design patterns are used to describe details at a lower and more local level—the *micro*architecture:

> [Architectural styles and design patterns] are different in that a style *tends* to refer to a coarser grain of design solution than a pattern, which *tends* to refer to a design solution localized within a few (or one) of a system's many architectural components. [CBB+03]

Thus, whereas the application of an architectural style will generally have a significant impact on the general organization of the various components, applying a design pattern will usually have a much more limited and localized impact.

Although the notion of design pattern needs not be restricted to the object-oriented paradigm, most of the literature in fact describes *object-oriented design patterns.* Such patterns can be categorized in various ways. For instance, Gamma et al. use the following categories [GHJV95]:

- *Creational patterns* deal with the creation of objects (for example, builder, factory, prototype, singleton).
- *Structural patterns* deal with the composition of objects (for example, adapter, bridge, composite, decorator, façcade, flyweight, proxy).
- *Behavioral patterns* describe how objects interact (for example, command, interpreter, iterator, mediator, memento, observer, state, strategy, template, visitor).

On the other hand, Buschmann et al. classify patterns into the following categories [BMR+96]:

- *Structural decomposition* patterns address the "decomposition of subsystems and complex components into cooperating parts" (for example, whole–part).
- *Organization of work* patterns define "how components collaborate together to solve a complex problem" (for example, master–slave).
- *Access control* patterns define "guards and control access to services and components" (for example, Proxy).
- *Management* patterns handle "homogeneous collections of objects, services and components in their entirety" (for example, command processor, view handler).
- *Communication* patterns "help organize communication between components" (for example, forward–receiver, dispatcher–server, publisher–subscriber).

Given the large number of design patterns and styles described in the literature [GHJV95, BMR+96, Fow03], it is beyond the scope of this overview to give a detailed presentation of those various design patterns. Let us conclude, though, that a modern software designer should understand the key styles and patterns, as this will help avoid "reinventing the wheel" each time a new design problem is tackled, while establishing a common communication vocabulary among software developers.

3.3. Design of Families of Systems and Frameworks

An important goal of software design has always been to allow for the *reuse* of software elements. Recent approaches toward that goal are based on software product lines and software components. A *software product line* is "a collection of systems sharing a managed set of features constructed from a common set of core software assets" [BCK03]. A product line thus defines a *family* of systems and is based on and populated with *software components,* which are "unit[s] of composition with explicitly specified provided, required and configuration interfaces and quality attributes" [Bos00].

The detailed presentation of the principles and techniques underlying the design of software product lines is beyond the scope of the present paper. Let us simply indicate that building a common set of software assets involves identifying the key *commonalities* encountered among the various members of the possible family of products—done through *domain analysis* [McC97, WL99, Bos00]—as well as accounting for their possible *variabilities*—done by identifying and defining reusable and *customizable* components [McC97, Bos00].

Customization of components can be supported through a number of mechanisms, for example, inheritance, extension, configuration, template instantiation, and generation [Bos00]. In an object-oriented context, a related notion is that of *framework,* a partially complete software subsystem that can be extended by instantiating specific *plug-ins* (also known as *hot spots*) [Pre95, BMR+96, Bos00].

4. SOFTWARE DESIGN QUALITY ANALYSIS AND EVALUATION

According to ISO/IEC Standard 9126-1 [ISO01], software quality—defined as "the totality of features and characteristics of a software product or service that bear on its ability to satisfy stated or implied needs"—can be characterized by the following six properties: functionality, reliability, usability, efficiency, maintainability, and portability. Each of these may in turn be defined through an appropriate set of attributes [ISO01]. In the following, we briefly introduce some of the quality attributes applicable to design as well as some techniques to help attain those quality attributes.

4.1. Design Quality Attributes

A key distinction between the various quality attributes concerns whether their influence is observable or not at run time [BCK03]:

- *Run-time qualities* are observable only while the system is functioning; for example, functionality, usability, performance, reliability and availability, and security.
- *Development-time* qualities have an impact on the work of the development and maintenance teams, but are not directly observable at run time; for example, integrability, modifiability, portability, reusability, and testability.

Although some qualities can be achieved through appropriate *architectural choices*—for example, modifiability and reusability, performance—some others cannot—for example, functionality and usability [KB00]. An informal test, suggested by Kazman and Bass [KB00], to see if a particular quality attribute can be achieved through architectural choices is to ask the following question: "[C]an I improve [the] rating for that attribute by making structural changes?"

Parnas and Weiss, in their active design review approach [PW85], identify the key desirable properties of a design as being the following: it should be well structured, simple, efficient, adequate (satisfying the requirements), flexible (easy to change), practical (module interfaces sufficient for the job), implementable, and standardized (documentation organized in a standard way).

Another important quality attribute related with design concerns the architecture's intrinsic quality known as *conceptual integrity* [Bro95], which characterizes an architecture that "reflects one single set of design ideas," leading to simplicity, consistency, and elegance.

4.2. Measures

A number of *measures* can be defined to obtain *quantitative estimates* of a design's size, structure, or quality. Such measures generally depend on the selected design approach:

- *Function-oriented (structured) measures:* the design's structure, obtained through functional decomposition, is represented as a structure chart on which measures can be computed; for example, fan-in/fanout, cyclomatic complexity, integration complexity [MB89, Pre01].
- *Object-oriented measures:* the design structure is represented as class diagrams, on which measures can be computed; for example, weighted methods per class, depth of inheritance tree, number of children, coupling between object classes, responses for a class [CK94, Pre01].

4.3. Quality Analysis and Evaluation Tools

Although measures can be used to estimate certain quality attributes—for instance, complexity metrics can be used to evaluate the testability of a software unit and to determine how much testing needs to be performed—many quality attributes are hard to quantify. Thus, other techniques must be used to evaluate the quality of a design:

- *Software design reviews* are informal or semiformal, often group-based, techniques used to verify the quality of design artifacts; for example, architecture reviews [BCK03], design reviews and inspections [PW85, Bud03], scenario-based architecture evaluation [BCK03, Bos00], and requirements tracing [TD02].
- *Simulation and prototyping* are dynamic techniques used to evaluate a design; for example, simulation-based performance or reliability analysis [BCK03, KB00, Bos00], and feasibility prototyping [BCK03, Bos00].

5. SOFTWARE DESIGN NOTATIONS AND DOCUMENTATION

Many different notations exist to represent software design artifacts, for instance, 18 different kinds of notations are mentioned in the Software Design Knowledge Area of the Guide to the SWEBOK [AMBD04]. Some notations are used mostly during architectural design, whereas others mainly during detailed design, although some can be used in both phases. Some notations are also used mostly within specific design methods, whereas others are more widely used.

Budgen [Bud03] categorizes the various design notations in terms of black box versus white box: as a *black-box* notation "is concerned with the *external* properties of the elements of a design model," whereas a *white-box* notation "is largely concerned with describing some aspect of the detailed realization of a design element" [Bud03].

An alternative characterization, which we use below to present briefly a small number of notations, is to distinguish between notations for describing *structural* (static) properties—a design's structural organization—and those for describing *behavioral* (dynamic) properties—the behavior of the software components.

5.1. A Selection of Design Notations

Over the last few years, UML (Unified Modeling Language) [BRJ99] has become an almost de facto standard for software development notations. In what follows, we briefly present a (small) selection of software design notations; notations whose name appear in italics are part of UML. Structural descriptions (static view) These 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):

- *Class and object diagrams.* These are used to represent a set of classes (and objects) and their relationships [BRJ99]. These diagrams are used in object-oriented design. A related, although somewhat older, notation is entity-relationship diagrams (ERDs), used to represent conceptual models of data stored in information systems [Mar94, TD02].
- *Component diagrams.* These are used to model the static implementation view of a system, that is, physical things (and their relationships) such as executables, libraries, tables, files, and documents [BRJ99]. Although their main use is during construction, such diagrams can also be used during design; for example, to document the module (work assignment) structure [BCK03].
- *Deployment diagrams.* These are used to model the static deployment view of a system, that is, "the configuration of run time processing nodes and the components that live on them" [BRJ99]. Typically, such diagrams can be used to represent distribution aspects, for example, to model embedded, client/server or distributed systems.
- Structure charts. These are used to describe the calling structure of programs (which procedure or module calls/is called by which other) [PJ88, Pre01, Bud03]. Such diagrams are at the heart of the structured (functionoriented) design approach.
- Structure (Jackson) diagrams. These are used to describe the data structures manipulated by a program in terms of sequence, selection and iteration [Mar94, Bud03]. These diagrams were initially introduced in JSP (Jackson Structured Programming) [Jac75].

Behavioral Descriptions (Dynamic View)

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

- *Activity diagrams.* These are used to show the control flow from activity ("ongoing nonatomic execution within a state machine") to activity [BRJ99]. These diagrams are related to the older owcharts [Pre01].
- *Interaction diagrams.* These are used to show the interactions among a group of objects [BRJ99]. These diagrams come in two flavors: *sequence diagrams* put the emphasis on the time-ordering of messages, whereas *collaboration diagrams* put the emphasis on the objects, their links, and the messages they exchange on these links.
- Data flow diagrams (DFDs). These are used to show the data flow among a set of processes [PJ88, Pre01, Bud03]. These diagrams were introduced and used by the structured analysis and design approach [YC79].
- State transition diagrams and *statechart diagrams.* These are used to show the control flow from state to state in a state machine [BRJ99, Bud03].
- Pseudocode and program design languages (PDLs). These are structured, programming-like languages used to describe, generally at the detailed design stage, the behavior of a procedure or method [Pre01, Bud03].

5.2. Design Documentation

Given the variety of notations available for design, a key question is how these various notations can be combined to obtain a coherent design document. There is no clearcut answer to this question, as it depends on many aspects, for instance, the type of software, the software development method being used, the organization in which/for which the software is developed, the stakeholders involved, and so on. A key practice, though, is the use of views, introduced in Section 3.

The selection of an appropriate set of views strongly depends on the stakeholders involved: project managers, developers, testers and integrators, customers, end users, and so on, all have different needs. Satisfying these different needs can best be described in terms of the relative importance of the various views from each viewtype (module, component-and-connector,

and allocation; see Section 3.1) [CBB+03]. For instance, a project manager would need detailed allocation views, whereas a developer would need mostly detailed module and component-and-connector views.

Documenting a view involves, among other things, describing the *interfaces* of the elements from that view. How such an interface is defined will depend on the type of element. A key characteristic of any interface specification, though, is that it should be a *two-way* description: what the element *provides* and what it *requires*—the resources used by the element, and the assumptions it makes from the environment. Clements et al. [CBB+03] offer a good presentation of these ideas.

Another key idea, formulated initially by Parnas and Clements, is that a design should be presented and documented in a *rational* way [CP86, Cle00], even though the process that lead to this design may not have been perfectly rational. As an analogy, consider the presentation of a major discovery that need not follow the process that led to that discovery (often by trial and error). In addition, even though this part is not strictly *rational,* the *rationale* behind the key decisions should also be recorded; for instance, the design alternatives that were considered and rejected should be described.

6. SOFTWARE DESIGN STRATEGIES AND METHODS

Various general principles and *strategies* have been proposed to guide the design process and help improve the quality of the resulting software [Mar94, BMR+96, Bud03]. In contrast with strategies, *methods* are more specific in that they generally suggest a particular set of *notations* together with a description of a *process* to be followed when designing software, as well as *heuristics* that provide guidance in adapting the method to a particular context [Bud03]. Such methods, which generally incorporate, in various ways, the general design principles and strategies, can help improve the quality of the resulting software when applied in a proper context. They are also useful as a means for transferring knowledge and as a common framework for teams of developers.

6.1. General Strategies and Enabling Techniques

General software design strategies can be described in terms of *enabling techniques,* a notion introduced by Buschmann et al. to denote fundamental principles and techniques of software design which are "independent of [any] specific software development method, and [. . .] have been known for years" [BMR+96]:

- *Abstraction.* Abstraction is "the process of forgetting information so that things that are different can be treated as if they were the same" [LG01]. Two key mechanisms are *abstraction by parameterization*—abstract from specific data by introducing parameters—and *abstraction by specification*—abstract how a module is implemented by referring to an appropriate specification. These mechanisms lead to three major kinds of abstraction: procedural abstraction (to introduce new operations), data abstraction (to introduce new data types), and control (iteration) abstraction (to iterate over collections of elements).
- *Coupling and cohesion.* Coupling is defined as the strength of the relationships *between* software components, whereas cohesion is defined by how the elements making up a component are related [BCK03, Pre01]. As a general rule, coupling between components should be weak, whereas the (internal) cohesion of a component should be high. Although these concepts were initially introduced for structured design [YC79], they also apply to object-oriented design [PJ00].
- *Divide and conquer.* In an algorithmic sense, divide and conquer is a technique that solves a complex problem by dividing it into two or more simpler problems, which are then solved recursively and whose solutions are subsequently combined to obtain the solution to the initial problem. In a function-oriented sense, divide and conquer involves breaking down a complex problem or task into simpler subproblems or subtasks that can be solved independently, a strategy at the root of *stepwise refinement* [Wir71, Bud03]. A related strategy is the *separation of concerns,* which suggests that "different or unrelated responsibilities should be separated from each other" [BMR+96].
- *Information hiding and encapsulation.* Information hiding is a general design strategy introduced by Parnas in which "every module [. . .] is characterized by its knowledge of a design decision which it hides from all others" [Par72]. A key principle associated with information hiding is the *separation of interface and implementation,* wherein the "interface or definition [of a module is] chosen to reveal as little as possible about its inner workings" [Par72]. In other words, a public interface (known to the clients) is specified, separate from the details of how the component is realized. Another related notion is *encapsulation,* defined as "the grouping of related ideas into one unit, which can thereafter be referred to by a single name" [PJ00]. Thus, encapsulation combines elements to create a new entity, whose internal details are hidden; in other words, encapsulation creates a new abstraction.
- *Sufficiency, completeness,* and *primitiveness.* These notions pertain to the idea that a software component should capture all the important characteristics of an abstraction needed to interact with it, and nothing more [BMR+96, LG01].

6.2. Function-oriented (Structured) Design

Structured (function-oriented) design [YC79, PJ88, Pre01, Bud03] is one of the early software design paradigms, in which decomposition centers on identifying the major systems *functions,* which are then elaborated and refined in a top-down manner, that is, using a divide-and-conquer approach based on functional decomposition.

Structured design is generally performed after structured analysis. A typical structured analysis [You89] produces, among other things, data flow diagrams (DFD) of the various system functions together with associated process descriptions, that is, descriptions of the processing performed by each subtask, usually using informal pseudocode. Entity-relationship diagrams describing the data stores can also be used.

Two key strategies have been proposed to help derive a software architecture, represented as a structure chart, from a DFD:

1. *Transaction analysis.* A *transaction* is characterized by some *event* in the environment that generates a *stimulus* to the system, which in turn triggers some *system's activity* that produces a *response* having an *effect* upon the environment. Transaction analysis consists in identifying the key transaction types of a system and using them as the units of design, that is, designing separately the processing of each transaction type.
2. *Transformation analysis.* The key step in deriving a structure chart from a DFD (for a given transaction) is to identify the *central transform,* that is, "the portion of the DFD that contains the essential functions of the system and is independent of the particular implementation of the input and output" [PJ88]. A *first-cut* (draft) structure chart can then be obtained by lifting the bubbles associated with the central transform, promoting them at the top level of the structure chart, as illustrated in Figure 1. Remember that a structure chart is a *hierarchical* diagram that shows the calls or is called the relationships. Of course, this initial structure chart will have to be revised and completed, in line with the quality criteria of cohesion and coupling as well as with various heuristics. Other details may also need to be revised or added; for example, error handling modules, initialization and termination details, required control flags, and so on.

Key concepts of structured design are those of *coupling* and *cohesion,* which characterize a design of good *quality.* For instance, a good design should restrict the coupling between modules to *normal* types of coupling—data, stamp, and control coupling, data coupling being the preferred form, where communication between modules is through parameters, where each parameter is an elementary piece of data—and should avoid other *pathological* forms of coupling—namely, *common* and *content* coupling

Similarly, a good design should give preference to modules having high cohesion; more precisely, modules exhibiting *functional cohesion* when the module "contains elements that all contribute to the execution of one and only one problem-related task" [PJ88]. Other, weaker, types of cohesion have been identified, from more cohesive to less cohesive: sequential and com-

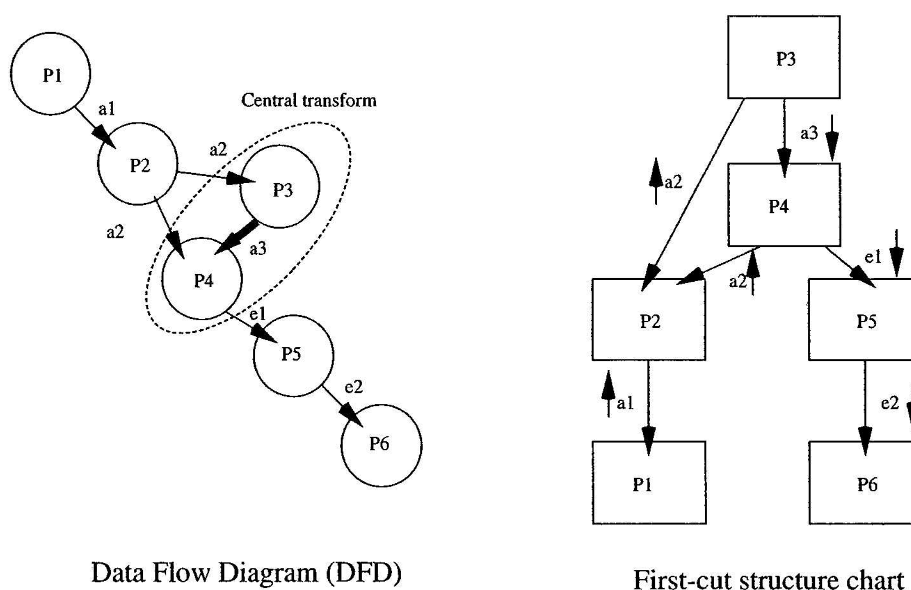

Figure 1. Using transform analysis to derive a structure chart from a DFD.

municational, procedural and temporal, logical and coincidental. Those weaker forms of cohesion can sometimes be acceptable, although the less cohesive ones should preferably be avoided.

Additional heuristics have also been suggested to help improve the quality of the resulting design:

- *Fan-in/fan-out.* A high fan-in—the number of modules that call a given module *M*—is considered good, as it indicates reuse of *M*. On the other hand, a low to moderate fan-out (maximum 5–7)—the number of modules that *M* calls—is generally preferable.
- *Decision splitting.* Decision splits, which occur when the recognition of a condition and the execution of the associated action are not kept within the same module, should be avoided.
- *Balanced systems.* A balanced system, when the top-level modules deal with logical and abstract data (clean and valid data, independent of implementation format), is preferable.

Structured design, being one of the first well-described and well-known design methods, made important contributions to the field of software design. Its integration with an appropriate analysis method—structured analysis [You89]—was also one of its key strengths. With the emergence of object-oriented languages and programming, though, structured design, with its emphasis on functional decomposition, started to reach its limits.

6.3. Object-oriented Design

The notion of *object* is intimately tied to the notions of data abstraction, encapsulation, and abstract data type (ADT). More precisely, an object is described by the following characteristics [Mac82, Boo86]: an object can be created/destroyed, has a unique (immutable) *identity,* possesses a (mutable) *state* (i.e., evolves in time), and exhibits some well-defined *behavior* through services it offers. Objects are generally organized into *classes,* which describe collections of objects sharing the same structure and behavior, thus the link with data types.[2]

Over the years, numerous software design methods based on objects, collectively known as *object-oriented design (OOD) methods,* have been proposed [Boo86, Boo94, WBWW90, CY91, JBP+91]. Early approaches, in which objects were mostly similar to entities in entity-relationship modeling and inheritance was not used [Abb83, Boo86], were said to be *object-based.* Later approaches, in which inheritance and polymorphism play a key role, are said to be *object-oriented* (*OO*).

OO design methods aim at developing software systems composed of interacting objects that are highly modular and, thus, easy to modify, extend, and maintain. OO design models address structural (static) aspects—classes and objects, their relationships and their grouping as well as behavioral (dynamic) ones—objects' behavior and interactions. The notations used for documenting these models take various forms; for example, diagrammatic, textual, and even mathematical. Much like structured design was intimately tied with structured analysis, existing OO design methods are generally associated with OO analysis methods. Contrary to structured analysis and design, though, many of the notations used for OO design can also be used during requirements analysis, leading to some degree of *seamlessness* between the two. This seamlessness, however, must not make one forget that requirements analysis and design do deal with different concerns; put succinctly, problem domain versus solution domain.

The Unified Modeling Language (UML), because it evolved from the integration of a number of OO methods, provides a wide variety of notations for OO analysis and design. UML includes various notations in addition to those mentioned in Section 5.1, for example, real-time modeling, formal specification using the Object Constraint Language (OCL) [WK99], and so on. UML is not an OO design method, though; UML is simply a set of notations, neutral with respect to any specific design method. On the other hand, the Unified Process (UP), elaborated by the same people who developed UML [JBR99], does define a software development process that incorporates OO analysis and design.

The Unified Process consists of four *phases:* inception, elaboration, construction, and transition. Each phase, delimited by an appropriate *milestone,* consists of one or more *iterations,* where each iteration generally results in an executable release and involves a number of core *workows,* namely, requirements, analysis, design, implementation, and test. From a software design perspective, the important phases are the elaboration and construction phases, since the *architectural baseline* (i.e., the software architecture description; see the "4+1 view model" described in Section 3.1) is developed during the elaboration phase whereas most of the detailed design is developed during the construction phase.

The key input to the design workow is a collection of use cases that describe the functional requirements a use case is "a description of a set of sequences of actions [. . .] that a system performs to yield an observable result [. . .]" [BRJ99]—together with the applicable non-functional requirements. The output of the design workflow is a *design model* consisting of classes and their collaborations, possibly organized into packages and subsystems, that provide the intended behavior while satisfying the non-functional requirements.

---

[2]However, note that an ADT does not necessarily define a class of objects. See [Mac82] for a clear exposition of the notions of *values* versus *objects.*

A class diagram models a set of classes and their relationships, for example, association, aggregation, inheritance, dependency, and so on. During design, class diagrams play a central role as they identify the major kinds of objects that will cooperate to produce the system behavior. The analysis model also contains class diagrams. Although these latter classes can provide a starting point for identifying some of the design model classes, there is not necessarily a direct correspondence between the two sets of classes; while the analysis classes describe the system intended behavior—the *what?,* the *black-box* view—the design classes instead describe how this behavior is obtained—the *how?,* the *white-box* view.

Figure 2 shows a simple UML class diagram for bank accounts. Two different kinds of account are available: `SavingAccount` and `CheckingAccount`. Both are specializations, indicated by an *inheritance* relationship, of a general `BankAccount`. A Customer owns a bank account, in fact, can own multiple bank accounts (the "*" annotation). Since this is a class diagram for the design model, some methods have been indicated.

How objects from the various classes collaborate to provide the desired system behavior is described using *interaction diagrams.* As mentioned in Section 5.1, interaction diagrams come in two flavors: sequence diagrams and collaboration diagrams. Figure 3 shows a simple collaboration diagram, which involves objects (underlined names), not classes. The example illustrates how the indicated objects collaborate to perform a `transferTo` operation, associated with the `BankAccount`.class. Each arrow indicates a message being sent (a method being called), conditionally, in the cases of messages 2 and 3. The numbering of the messages indicates their time ordering.

Although interaction diagrams describe how a number of objects collaborate to realize a given operation, in reaction to a specific event, they do not describe the behavior of a specific class of objects in reaction to all possible events. Such class specific behavior can be described using a *statechart* diagram, a generalized form of state transition diagram. A statechart diagram thus describes an object behavior from an internal viewpoint—how the internal state of an object changes in reaction to the various events.

The early OO methods focused mostly on data abstraction and ADTs, viewing primarily objects through their components and static structural relationships, an approach called "data-driven design" [WBW89]. The Unified Process design approach, as do many modern OOD methods, instead focuses on properly identifying and assigning responsibilities to classes and objects, an approach named "responsibility-driven," initially introduced by Wirfs-Brock et al. [WBW89, WBWW90]. More precisely, a *responsibility* is defined as "a contract or an obligation of a class" [BRJ99], or "an obligation to perform a task or know information" [WBM03]. Although such contracts can be defined formally, for example, using formal pre/postconditions as is done in Meyer's Design by Contract (DBC) approach [Mey92], even informal descriptions can be useful [LG01]. Responsibility-driven design generally improves encapsulation, produces a less complex design with improved coupling and cohesion [SC93], and tends to produce systems in which the overall control is better organized and balanced, thus more modular.

The growth of the OO approach over the last two decades has been phenomenal: the rise from OO programming, to design, then to analysis; the development of a *lingua franca* (UML); the design patterns movement; and so on. For lack of space, however, several aspects related with OO design have not been addressed; for example, persistence, error handling, component-based design, and so on. Once the key ideas of OOD are understood, a next important step, as mentioned in Section 3.2, is to address the vast literature on OO design patterns.

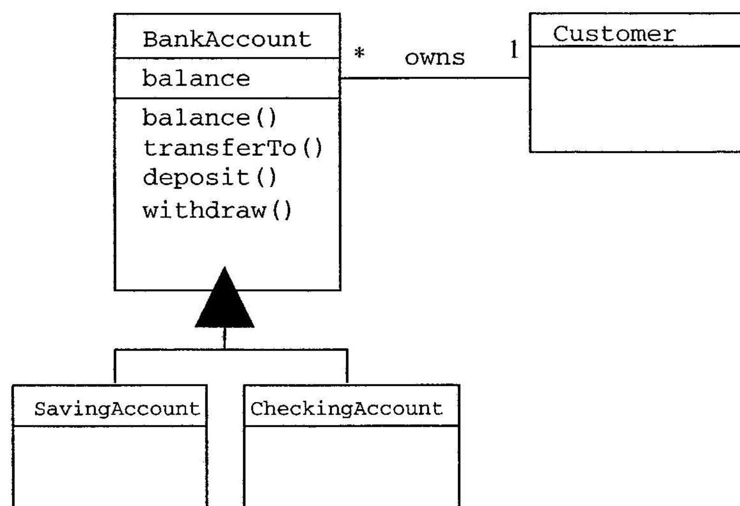

Figure 2. A UML class diagram for bank accounts.

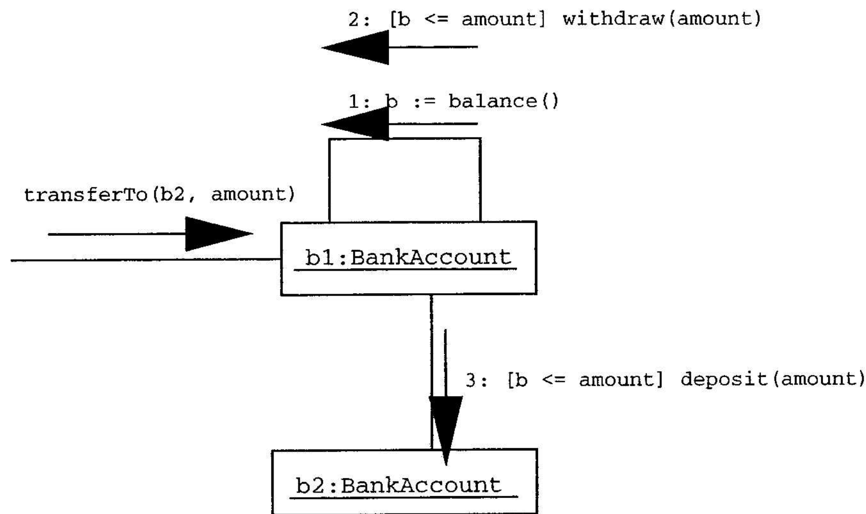

Figure 3. A UML collaboration diagram for a `transferTo` operation.

6.4. Data-structure-oriented Design

Data-structure-oriented design—also known as Jackson Structured Programming (JSP) [Jac75] is an approach in which the emphasis is on the data that a program manipulates rather than the functions it performs [Cam89, Pre01, Bud03]. This emphasis on data is motivated by the fact that such data is generally more *stable* (less subject to change) than the functions that need to be performed.

In JSP, the designer first describes the input and output data—for instance, using Jackson structure diagrams—and then develops the program's control structure by establishing an appropriate correspondence between the input and output data structure diagrams. Once the program control structure is properly defined, appropriate program actions and conditions are then added to obtain the final program. A number of heuristics have also been proposed to deal with special cases; for example, how to deal with various kinds of mismatches (also known as *structure clashes*) between the input and output structures by using program inversion.

JSP's scope was mostly restricted to the design of data-processing programs using sequential (batch-style) files and processes. Jackson later introduced the JSD method (Jackson System Development) [Jac83] that deals with the analysis and design of systems composed of more complex interacting processes involving various entities performing actions, an approach similar, in certain ways, to object-oriented design.

7. CONCLUSION

Software design is a rich and still evolving field, so this overview could only scratch the surface. In fact, following the spirit of the *Guide to the Software Engineering Body of Knowledge,* which aims at presenting *generally accepted knowledge*—"knowledge and practices [that] are applicable to most projects most of the time, and [such] that there is a widespread consensus about their value and usefulness" [AMBD04]—this overview should be considered as a *guide* to the field of software design. Further detail on the topics discussed in this overview or on more specialized topics related with design (for example, design of real-time or distributed systems) can be obtained by consulting the various references mentioned in the next section or in the accompanying papers.

REFERENCES

[Abb83] R. Abbott. Program design by informal english description. *CACM, 26*(11):882–894, 1983.

[AMBD04] A. Abran, J. W. Moore, P. Bourque, and R. Dupuis, editors. *Guide to the Software Engineering Body of Knowledge* (2004 Version). IEEE Computer Society Press, 2004.

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice* (2nd ed.). SEI Series in Software Engineering. Addison-Wesley Professional, 2003.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture—A System of Patterns.* Wiley, 1996.

[Boo86] G. Booch. Object-oriented development. *IEEE Transactions on Software Engineering, SE-12*(2):211–221, Feb. 1986.

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

[Bos00] J. Bosch. *Design and Use of Software Architectures—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.

[Bro95] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1995.

[Bud03] D. Budgen. *Software Design* (2nd ed.). Pearson Education/Addison-Wesley, 2003.

[Cam89] J. Cameron. *JSP and JSD: The Jackson Approach to Software Development* (2nd ed.). IEEE Computer Society Press, 1989.

[CBB+03] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond.* SEI Series in Software Engineering. Addison-Wesley, 2003.

[CK94] S. R. Chimader and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20*(6):476–493, June 1994.

[Cle00] P. C. Clements. Rationalize your design. In P. C. Clements, editor, *Constructing Superior Software,* chapter 5, pp. 105–124. McMillan Technical Publishing, 2000.

[CP86] P. Clements and D. L. Parnas. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering, SE-12*(2):251–257, 1986.

[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.

[Fow03] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2003.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[IEE90] IEEE. IEEE standard glossary of software engineering terminology. Technical Report IEEE Std 610.12-1990, IEEE, New York, 1990.

[IEE98] IEEE. An american national standard—IEEE recommended practice for software design descriptions. Technical Report IEEE Std 1016-1998, IEEE, New York, 1998.

[IEE00] IEEE. Recommended practice for architectural description of software-intensive systems. Technical Report IEEE Std 1471-2000, IEEE, New York, 2000.

[ISO95] ISO/IEC. Information technology—software life cycle processes. Technical Report ISO/IEC Std 12207: 1995, ISO/IEC, 1995.

[ISO01] ISO/IEC. Software engineering—product quality—part I: Quality model. Technical Report ISO/IEC 9126.1-2001, ISO/IEC, 2001.

[Jac75] M. A. Jackson. *Principles of Program Design.* Academic Press, 1975.

[Jac83] M. A. Jackson. *System Development.* Prentice-Hall, 1983.

[JBP+91] Rumbaugh J., 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.

[JTM00] J.-M. Jézéquel, M. Train, and C. Mingins. *Design Patterns and Contracts.* Addison-Wesley, 2000.

[KB00] R. Kazman and L. Bass. Software architecture and quality. In P. C. Clements, editor, *Constructing Superior Software,* chapter 4, pages 83–104. McMillan Technical Publishing, 2000.

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

[Kru00] P. B. Kruchten. *The Rational Unified Process: An Introduction* (2nd ed.). Addison-Wesley, 2000.

[LG01] B. Liskov and J. Guttag. *Program Development in Java.* Addison-Wesley, 2001.

[Mac82] B. J. MacLennan. Values and objects in programming languages. *SIGPLAN Notices, 17*(12):70–79, 1982.

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

[MB89] T. McCabe and C. W. Butler. Design complexity measurement and testing. *CACM, 32*(12):1415–1425, 1989.

[McC97] C. McClure. *Software Reuse Techniques: Adding Reuse to the System Development Process.* Prentice-Hall, Inc., 1997.

[MEH01] M. W. Maier, D. Emery, and R. Hilliard. Software architecture: Introducing IEEE Standard 1471. *IEEE Computer, 34*(4):107–109, April 2001.

[Mey92] B. Meyer. Applying "design by contract". *IEEE Computer, 25*(10):40–51, 1992.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM, 15*(12):1053–1058, 1972.

[PJ88] M. Page-Jones. *The Practical Guide to Structured Systems Design* (2nd ed.). Prentice-Hall, 1988.

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

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

[Pre01] R. S. Pressman. *Software Engineering—A Practitioner's Approach* (5th ed.). McGraw-Hill, 2001.

[PT76] L. J. Peters and L. L. Tripp. Is software design "wicked"? *Datamation, 22*(5):176, 1976.

[PW85] D. L. Parnas and D. M. Weiss. Active design reviews: Principles and practices. In *Proceedings of the Eigth International Conference on Software Engineering,* pp. 215–222, 1985.

[RW84] H. W. J. Rittel and M. W. Webber. *Developments in Design Methodology,* pages 136–144. Wiley., 1984.

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

[SC93] R. C. Sharble and S. S. Cohen. The object-oriented brewery: A comparison of two object-oriented development methods. *ACM SIGSOFT Software Engineering Notes, 18*(2):60–73, 1993.

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

[SNH95] D. Soni, R. Nord, and C. Hofmeister. Software architecture in industrial applications. In *Proceedings of the 17th International Conference on Software Engineering,* pp. 196–207, Seattle, WA, 1995. ACM Press.

[TD02] R. H. Thayer and M. Dorman, editors. *Software Engineering—Volume 1: The Development Process* (2nd ed.). IEEE Computer Society Press, 2002.

[WBM03] R. Wirfs-Brock and A. McKean. *Object Design—Roles, Responsibilities, and Collaborations.* Addison-Wesley, 2003.

[WBW89] R. Wirfs-Brock and B. Wilkerson. Object-oriented design: A responsibility-driven approach. In *Proceedings of OOPSLA '89,* pp. 71–75, New Orleans, LA, 1989.

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

[Wir71] N. Wirth. Program development by stepwise refinement. *CACM, 14*(4):221–227, 1971.

[WK99] J. Warmer and A. Kleppe. *The Object Constraint Language-Precise Modeling with UML.* Addison-Wesley, 1999.

[WL99] D. J. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999.

[YC79] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Programming and Systems Development.* Prentice-Hall, 1979.

[You89]. E. Yourdon. *Modern Structured Analysis.* Yourdon Press, 1989.