



Ahoo Engineering Group

Software Engineering

by Nguyen Xuan Huy

Institute of Information Technology,

National Center for Natural Science and Technology

Email: nxhuy@ioit.ncst.ac.vn

Duration: 45 hours



Chapter 3 Software Design

Objectives

This chapter gives a general discussion of software design. The introduction considers the importance of design and its role in the software process. Some methods and techniques are presented for mastering design problems, that is, for decomposing systems into subsystems and subsystems into components, and for handling questions regarding the internal structure of system components. First we give an overview of various design techniques; then we explore some of those design techniques and their methodological foundation in detail and demonstrate their application with an example. In addition, we discuss certain design aspects such as the design of user interfaces and modularization criteria.

Contents

3.1 Design techniques

3.1.1 Top-down design

3.1.2 Bottom-up design

3.1.3 Systems design

3.1.4 Design decomposition

3.2 User interface design

3.3 Function-oriented design

3.4 Object-oriented design

3.4.1 The Abbott Method

3.4.2 Design of class hierarchies

3.4.3 Generalization

References and selected reading

3.1 Design techniques

- Specification design sits at the technical kernel of the software engineering process
- Specification design is applied regardless of the software process model that is used.
- Software design is the first of three technical activities – *design*, *code generation*, and *testing* – that are required to build and verify the software.

One of the most important principles for mastering the complexity of software systems is the *principle of abstraction*.



Two approaches:

- Top-down design, and
- Bottom-up design.

3.1.1 Top-down design

([Dijkstra 1969], [Wirth 1971])

- The design activity must begin with the analysis of the requirements definition and should not consider implementation details at first.
- A project is decomposed into subprojects, and this procedure is repeated until the subtasks have become so simple that an algorithm can be formulated as a solution.
- Top-down design is a successive concretization of abstractly described ideas for a solution.
- Abstraction proves to be the best medium to render complex systems comprehensible; the designer is involved only with those aspects of the system that are necessary for understanding before the next step or during the search for a solution.

3.1.2 Bottom-up design

The fundamental idea:

- To perceive the hardware and layer above it as an abstract machine.

Technique:

- Bottom-up design begins with the givens of a concrete machine and successively develops one abstract machine after the other by adding needed attributes, until a machine has been achieved that provides the functionality that the user requires.

An abstract machine is a set of basic operations that permits the modeling of a system.

3.1.3 Systems design

- In large-scale embedded system, the design process includes an element of systems design in which functions are partitioned into software and hardware functions.
- The advantage of implementing functionality in hardware is that a hardware component can deliver much better performance than the equivalent software unit. System bottlenecks can be identified and replaced by hardware components, thus freeing the software engineer from expensive software optimization.
- Providing performance in hardware means that the software design can be structured for adaptability and that performance considerations can take second place.

Parallelism



- A great many software systems, particularly embedded real-time systems, are structured as a set of parallel communicating processes which is an outline design of a simple control system.
- With a fast processor, it may not be necessary to implement an embedded system as a parallel process. A sequential system which uses polling to interrogate and control hardware components may provide adequate performance.
- The advantage of avoiding a parallel systems design is that sequential programs are easier to design, implement, verify and test than parallel systems. Time dependencies between processes are hard to formalize, control and verify.

Note:

- *There are some applications, such as vector processing, where a parallel approach is a completely natural one. If n -element vectors have to be processed with the same operation carried out on each element, the natural implementation is for a set of n -processes carrying out the same operation at the same time.*

Design process:

Two-stage activity:

1. Identify the logical design structure, namely the components of a system and their inter-relationships
2. Realize this structure in a form which can be executed. This latter stage is sometimes considered as detailed design and sometimes as programming.

3.1.4 Design decomposition

- The design process is influenced not only by the design approach but also by the criteria used to decompose a system.
- Numerous decomposition principles have been proposed.

Classification of decomposition methods

1. *Function-oriented decomposition.* ([Wirth 1971], [Yourdon 1979]).
 - A function-oriented system perspective forms the core of the design.
 - Based on the functional requirements contained in the requirements definition, a task-oriented decomposition of the overall system takes place.
2. *Data-oriented decomposition.* ([Jackson 1975], [Warnier 1974], [Rechenberg 1984a])



- The design process focuses on a data-oriented system perspective.
 - The design strategy orients itself to the data to be processed.
 - The decomposition of the system stems from the analysis of the data.
3. *Object-oriented decomposition.* ([Abbott 1983], [Meyer 1988], [Wirfs-Brock 1989], [Coad 1990], [Booch 1991], [Rumbaugh 1991])
- An object-oriented system perspective provides the focus of the design.
 - A software system is viewed as a collection of objects that communicate with one another. Each object has data structures that are invisible from outside and operations that can be executed on these structures.
 - The decomposition principle orients itself to the unity of data and operations and is based on Parnas' principle of information hiding [Parnas 1972] and the principle of inheritance derived from Dahl and Nygaard [Dahl 1966].

3.2 User interface design

- The design of the user interfaces is a subtask of the design phase.

Modes

- A program mode is a situation in which the user can only execute a limited number of operations.
- Windowing technology can provide valuable services for modes.
- With the help of windows, the user can process various subtasks simultaneously in different windows representing different modes.

Elementary operations

- The key to avoiding modes is the decomposition of the overall user dialog into elementary operations.

Example 3.1

In modern text editors, we can move a block of lines of a text to a new position in the text by four elementary operations:

- *Selection of the block of lines*
- *Menu command "Cut"*
- *Move the cursor to the new position*
- *Menu command "Paste"*

This *cut-copy-paste principle* permits the user a change of mind or corrections after each elementary operation.

The user first determines what is to be manipulated and only then what is to occur whereas in command languages the user first establishes the operation and then specifies the parameters.



Menus

- The requirement to decompose the user dialog into elementary operations also means that the input of commands should take place via a minimum of individual actions. Menu commands provide a means to achieve this goal.

Classification of menus

- **Pop-up menus** prove more efficient because they can appear at any position therefore they require less mouse movement.
- **Pull-down menus** permit better structuring of an extensive set of commands and are easier to use with a single-button mouse.

The user can click on the menu bar with the mouse to display all the commands belonging to a menu and can select a command, likewise with the mouse.

Classification of menu commands

- Immediately executable commands
- Commands with parameters
- Commands for switching modes

Directly executable commands including all menu commands that require no parameters or that operate on the current selection.

Example 3.2

Commands *Cut* and *Paste* are elementary operations.

With a simple mouse click the user causes the system to carry out an action that normally involves processing data.

Commands with parameters are similar in effect to those in the first class. They differ primarily in the user actions that are required to execute them.

Example 3.3

Text editor commands *Find* and *Find Next*: locate certain characters in a text.

Find has an implicit parameter, the position at which searching is to begin. The execution of the command prompts the user to input additional parameters. Input prompting is normally handled via a dialog window. The execution of such a command thus requires several sequential inputs from the user.

To simplify the repeated execution of a command with the same parameters, it can be useful to use a dedicated, immediately executable menu command (*Find Next*.)

Instead of manipulating data, the menu commands of the third class cause a change in mode that affects subsequent commands or the way in which data are displayed.

Example 3.4

Switching between *insert* and *overwrite mode* and the command *Show Controls* in a text editor to display normally invisible control characters.

- A frequently neglected task in the design of a menu system is the choice of appropriate wording for the menu commands. Apply the rule that the command should be as short as possible, yet still meaningful.



- In the design of a menu system, similar commands should be grouped together under the same menu.
- The more frequently a command is used, the higher in the menu it should be placed to avoid unnecessary mouse motion.

The basic possibilities for handling the situation where a command is invoked in a mode where it cannot be executed are:

- **The command has no effect: *This variant is easiest to implement, but can cause confusion and frustration for the user when the command fails to function in certain situations.***
- *An error message is displayed.* This possibility is preferred over the first because the user can be advised why the execution of the command is impossible. If the same situation arises repeatedly, however, the recurring error message tends to irritate the user.
- *The command is disabled.* The best choice is to prevent the selection of a senseless command from the start. To achieve this goal, the command should not be removed from the menu, which would confuse an occasional user and cause a search in vain for the command. It is better to mark non-executable commands as disabled in some suitable form.

Dialog windows

- *Static text* is used for titles, label and explanations. Dialog windows should be structured to minimize static text. In particular, avoid long explanations by means of adequate labeling of the elements available to the user.
- As far as possible, use *editable text* only for inputting character strings. For the input of numbers, other user interface elements do a better job of eliminating errors.
- *Command buttons* trigger immediate action. Typically an input dialog window contains an OK button to confirm an input and a Cancel button to abort an action.
- *Checkboxes* permit the selection of Boolean values.
- *Radio buttons* allow the selection of one alternative from a set of mutually exclusive possibilities.
- *Pop-up menus* provide an alternative to radio buttons. A mouse click on a framed area presents a list of available possibilities from which the user can select. Compared to radio buttons, pop-up menus require less space and can be arbitrarily extended. That they normally do not display all available options proves to be a disadvantage.
- *Sliders* enable the input of continuous, variable values. Use sliders wherever precision plays no role, e.g., for adjusting brightness or volume.
- *Selection lists* serve a function similar to pop-up menus. Use them when the number of alternatives exceeds the capacity of a menu.
- *Direction buttons* allow the setting of values in discrete steps. Their function resembles setting a digital watch. Direction buttons make it easy to restrict the numeric range.



- **Icons (or pictograms) frequently provide a space-saving alternative to verbal description.**

Note: The design of icons represents an art; use only self-explanatory pictures, preferably ones that the user recognizes from other areas of daily life.

- **Frames and separators provide an optical structuring medium. They augment special ordering for structuring a dialog window in cohesive functional blocks. Whenever possible, avoid extensive dialog windows with numerous individual elements. They become time-consuming to use because the user must control every adjustment, even to set only a single field.**
- **Modern graphical user interfaces with windows and menus have come to dominate not only because of their user friendliness, but also for aesthetic reasons. No one disputes that working with an attractive screen is more fun and so is perceived as less of a burden than character-oriented communication with the computer. Designers of newer systems thus pay particular attention to achieving an appropriate look. The graphic capabilities of monitors are exploited, e.g., to give windows and buttons a plasticity like Three-dimensional objects. Use such design possibilities only if they are standard in the target environment. It would be wrong to sacrifice the uniformity of a user interface for purely aesthetic reasons.**

Color

Guidelines for the design of color user interfaces ([Apple 1987])

- About one person in twenty is colorblind in some way. To avoid excluding such persons from using a program, color must not serve as the sole source of information. For example, highlighting an error in a text with red effectively flags it for most people, but is scarcely recognizable, if at all, to a red-green colorblind user. Another form of highlighting (e.g., inversion of the text) should be preferred over color.
- Longitudinal studies have shown that black writing on a white background proves most readable for daily work. For a black background, amber proves the most comfortable to the eyes. Writing in red and blue proves least readable because these colors are at opposite ends of the visible spectrum for humans. Because of the extreme wave length of these colors, their focal point is not exactly on the retina of the eye, but in front of or behind it. Although the eye automatically adapts to reading red or blue text, the resulting strain causes early fatigue. Experts recommend avoiding red and blue to display text.
- Red serves as a danger signal. However, because the user's eye is drawn to red, it would be wrong to employ red to mark dangerous actions. Color "support" in such a case would have the reverse effect.
- Light blue on a white background is hard to read, especially for small objects or thin lines. Thus important things should never be displayed in light blue. Still, this effect can be exploited, for example, to set grid lines in the background as unimportant.



- In general, color should only be used where the task expressly requires it, such as in graphic editors. In all other cases color should be used at best as a supplementary means for highlighting, but never for mere decoration of a user interface.

Sound

Guidelines for the design of sound user interfaces

- Sound is an excellent way to attract attention. Hence it should only be used when the user's attention is really required, but never as simple background for other actions. Appropriate applications of sound include error situations (e.g. paper exhausted while printing) and the occurrence of temporally uncertain events such as the end of an extended procedure (e.g. data transfer).
- Sound should only be used to underscore a message displayed on the screen, but never as the sole indicator of an exception situation. Otherwise an important message can easily be missed because the user is hearing-impaired, works in a loud environment, or has left the office.
- It might make sense to signal different events with different sound. However, observe that only a user with musical training or talent can discern signals with similar pitch.
- Avoid loud and shrill sounds, imagine a large office where every few minutes another computer trumpets an alarm. For the same reason, avoid sounds with more than three sequential tones; such signals become irritating when heard too often.
- Even in particularly pressing emergencies, never under any circumstances, employ a long, continuous sound or, even worse, an enduring staccato. Such sounds can create panic and provoke erroneous action from the user. It is better to use a short, uninterrupted sound and to repeat it at intervals of one minute until the user reacts.

Consistency

- Menu commands with similar functions should have the same nomenclature and the same location even in different programs,
- Shortcut keys for menu commands should always be consistent. Rather than introduce a nonstandard key combination, leave the menu command without a shortcut,
- Buttons with similar functions should have similar labels and the same relative position in different dialog windows,
- Dialog windows for the input of similar functions should have a consistent look.

A typical user works with various programs on the same computer. The acceptance of a new program depends significantly on whether the user encounters similar interface functionality as in other familiar programs. Every deviation as well as every omission can prove an irritation factor. For example, if the user is accustomed to interrupting any action by pressing a given key combination, then irritation results when this key has no function or a different function in a new program.



The greater the internal consistency of a user interface and its consistency with the user interfaces of other programs, the less effort is required for learning to use the new program. In many cases this can replace extensive user manuals. On the other hand, the obligation to maintain consistency, like any form of standardization, can also restrict possibilities for user interfaces.

Prototyping

- The dynamics of a program cannot be described statically. To avoid unpleasant surprises in the implementation phase, it is important to also define the behavior of the program as early as possible.
- A prototype serves as the best medium for specifying a user interface because it allows playing through the typical steps in working with the planned program. In such an experimentation phase, most questions regarding operation can be discussed and solved together with the client.
- Many software products are available today that support the interactive description of the user interface. Most of these tools generate the source code of a program that implements the user interface definition. The time required to implement a prototype thus shrinks from weeks to hours.

3.3 Function-oriented design

- The best-known design methods are function oriented, which means that they focus on the algorithm to solve the problem.

Example 3.5

Imagine the algorithm as a mathematical function that computes results on the basis of given parameters. At the start of the design stage the algorithm is still a black box whose contents are unknown. The most difficult task to be solved is its solution algorithm. Thus it make sense to proceed as with the modularization, i.e., to decompose the task into self-contained subtasks, whereby the algorithms for the solution of subtasks again should be viewed as black boxes. The resulting overall solution thus becomes a network of cooperating subalgorithms.

Two simple rules of the stepwise refinement principle of problem solving ([Wirth 1971]):

- Decompose the task into subtasks,
- Consider each subtask independently and decompose it again into subtasks.

Continue the ongoing decomposition of large tasks into smaller subtasks until the subtasks become so simple that they can readily be expressed with statements in the selected programming language.

Solving a task using above approach is characterized as follows:

- For each (sub)task, begin with the identification of its significant components. However, merely specifying the subtasks will not suffice; it must be clear whether and how the solutions to these subtasks can be assimilated into an overall solution. This means that the initial design of an algorithm certainly can contain control structures. The interconnection of the subsolutions can and should already be specified at this stage by means of sequential, conditional and repeated execution.



- Treat details as late as possible. This means, after the decompositions. This means, after the decomposition of a task into appropriate subtasks, waste no thought yet on what the solutions of these subtasks could look like. First clarify the interrelationship of the subtasks; only then tackle the solution of each subtask independently. This ensures that critical decisions are made only after information is available about the interrelationship of the subtasks. This makes it easier to avoid errors with grave consequences.
 - A continuous reduction of complexity accompanies the design process when using the principle of stepwise refinement. For example, if a task A is solved by the sequential solution of three subtasks B, C and D, there is a temptation to assess this decomposition as trivial and of little use.
 - The subtasks become increasingly concrete and their solution requires increasingly detailed information. This means stepwise refinement not only of the algorithms but also the data with which they work. The concretization of the data should also be reflected in the interfaces of the subsolutions. In the initial design step, we recommend working with abstract data structures or abstract or abstract data types whose concrete structure is defined only when the last subtasks can no longer be solved without knowledge thereof.
 - During stepwise refinement the designer must constantly check whether the current decomposition can be continued in the light of further refinement or whether it must be scrapped.
- The most important decomposition decisions are made at the very start, when the designer knows the least. Hence the designer is hardly able to apply stepwise refinement consistently.
 - If the design falters, the designer is tempted to save the situation by incorporating special cases into already developed subalgorithms.

3.4 Object-oriented design

[Booch 1991], [Coad 1990], [Heitz 1988], [Rumbaugh 1991], [Wasseman 1990], [Wilso 1990], [Wirfs-Brock 1989], [Wirfs-Brock 1990].

Function-oriented design and object-oriented design

- Function-oriented design focuses on the *verbs*
- Object-oriented design focuses on the *nouns*.
- Object-oriented design requires the designer to think differently than with function-oriented design.
- Since the focus is on the data, the algorithms are not considered at first; instead the objects and the relationships between them are studied.
- The fundamental problem in object-oriented design consists of finding suitable classes for the solution of the task.
- The next step is identifying the relationships of these classes and their objects to each other and defining the operations that should be possible with the individual objects.



3.4.1 The Abbott Method ([Abbot 1983])

- In object-oriented design we try quickly to identify the objects involved in task. These could be data (e.g. texts), forms (e.g. order forms), devices (e.g. robots) or organizational aids (e.g. archives).
- Verbal specifications can serve as the basis of the design.

Approach:

- Looking for various kinds of words (nouns, verbs) in the specifications and deriving the design from them.
- Forming abstract data structures.
- Defining the operations required on abstract data structures.

The approach can also be used for object-oriented design.

Working schema:

- *Filtering out nouns*

The relevance of a noun depends on its usage:

- *Generic terms* (e.g. person, automobile, traffic light) play a central role. Every such term is a candidate for a class.
- *Proper nouns* serve as identifiers for specific objects (e.g. Paris, War and Peace, Salvador Dali). They can indirectly contribute to forming classes since every proper noun has a corresponding generic term (e.g. city, book, painter).
- *Quantity and size units* (e.g., gallon) and material descriptor (e.g. wood) as well *collective terms* (e.g., government) usually prove unsuitable for forming classes.
- *Abstract terms* (e.g. work, envy, beauty) likewise prove unsuitable.
- *Gerunds* indicate actions. They are treated like the verbs from which they derive.

- *Seeking commonalties*

In real tasks descriptions many related nouns usually occur together, such as

“department manager” and “supervisor”, “author” and “write”, “auto” and “vehicle”,

“street” and “one-way street”. In such cases the analyst must first determine whether

the terms are really distinct, or whether they indicate the same class. The answer to the

question “Same or different?” generally emanates from the task description. For

example, for a traffic simulation it would be of no importance whether only cars or

arbitrary vehicles utilize a street. For a program to manage license plates and

registrations, this difference would certainly play a role. In any case, the criterion for



distinguishability is the question of whether the elements of the class in question have differing attributes.

If we find that two related terms designate different things, then we need to clarify what relation they have to one another. Often we find a subset relation that finds expression in sentences like: “Every auto is a vehicle”.

- *Finding the relevant verbs*

Verbs denote action with objects. In contrast to the function-oriented approach, here these actions are not viewed in isolation, but always in connection with objects of certain classes. For example, the verb “toggle” belongs to a traffic light, and “accelerate” is associated with a vehicle. Assignment to a certain class proves easy for intransitive verbs because they are bound only to the subject of the sentence (e.g., wait). Transitive verbs and sentence structures that connect multiple nouns present more problems. For example, the statement: “The auto is to be displayed on the screen” does not indicate whether the action “display” should be assigned to the design. A usual solution to the problem is to assign the action to both classes in question, and then during implementation to base one of the two operations on the other.

3.4.2 Design of the class hierarchies

A particular characteristic of object-oriented programming is that the classes (data types) are hierarchically structured. This approach has the organizational advantage that similar things are recognized and combined under a common superordinate term. This produces a tree structure in which the general (abstract) terms are near the root and the leaves represent specialized (concrete) terms.

The class `MotorVehicle` determines the general attributes of the classes derived from it. For example, independent of the type of vehicle, every vehicle has a brand name, a model, a series number, a power rating for the motor, and many other attributes. Every auto (since it is also a vehicle) also has these attributes. In addition, it has further attributes such as the number of passengers, the number of doors, etc. This scheme can be continued to any extent. Each specialization stage adds further attributes to the general ones.



Another characteristic of a class hierarchy is the extension of functionality with increasing specialization. A convertible, for example, permits all actions that are possible with a car. In addition, its roof can be folded down.

The most important relation in a class library is the *is-a* relation. Since the class Car is a subset of the class Auto, every car (object of class Car) is an auto. This relation represents the first criterion for designing class hierarchies. Often, however, there are multiple possibilities for assignment. For example, the class StationWagon could also be a subset of commercial vehicle for the transport of goods as reflected in the class Truck. Such ambiguities can impede the search for an adequate common superordinate term.

Such a class hierarchy has two consequences for implementation:

- *Inheritance*: Attributes and operations defined for a general class automatically apply to the more specialized classes derived from it. For example, if Auto has a function that computes the top speed from the wind resistance value and the engine data, then this function is also valid for all objects of derived classes (e.g., Car, StationWagon).
- *Polymorphism*: A program fragment that functions for objects of a class K can also work with objects of a class derived from K even though such objects have different attributes (structure, behavior) from K objects. This is possible because derived classes only *add* attributes, but can never *remove* any. This results in a flexibility that can be achieved only with arduous effort with conventional programming.

Often specifications employ terms that are similar in many ways yet differ significantly in certain details. Such an example arises in a program to display rectangles and circles. These geometric shapes have attributes that could be handled collectively (e.g. line thickness, fill pattern, movement on the screen), yet they cannot be combined fruitfully in an *is-a* relation. In such cases a common superordinate class, or superclass, must be created, e.g., *GeomFigure*. The attributes common to a circle and a rectangle (e.g., line thickness, color, fill pattern) are then transferred to the common superclass, and only the specialized attributes and operations (center and radius, width and height, drawing operations) are implemented in the derived classes. The common super-class represents an abstraction of the specialized classes. In this context we speak of *abstract classes*.

The availability of a collection of prefabricated classes (a class library) can save a great deal of implementation and even design work). A little luck might even produce an exact match, or at least a more general class from which the desired class can be derived by inheritance and specialization. In this case two activities in opposing directions characterize the design procedure: top-down analysis and bottom-up synthesis.

The Abbott Method is a *top-down analytical method* because we begin with the design of the application-oriented classes starting with the task description and progress to the implementation-specific classes. This method gives no consideration to any classes that might already be available. If new classes are derived exclusively on the basis of classes in the class library, this amounts to a *bottom-up constructive approach*. The truth lies somewhere between. For specification of higher complexity, a designer can scarcely find problem-oriented classes in the class library. Thus it makes sense in any case to establish the initial classes analytically.

3.4.3 Generalization



➤ Reusability plays a central role in object-oriented programming.

Type of reusability:

- Reuse within a program
- Reuse between programs.

Within a program reuse is realized with inheritance. High reuse results almost as a side effect of the design of the class hierarchy by the introduction of abstraction levels. A class library as the basis of implementation is the most important medium for achieving high reuse between programs. Usually such a class library is provided by the compiler manufacturer. If similar tasks are to be solved frequently, similar subtasks crystallize repeatedly that can be solved in the same way in different programs. This suggests striving for as much generality as possible in every class in order to assure high reusability in later projects. Above all the following measures prove useful toward this goal.

- *Far-sighted extensions*: In the development of a new class, more should be implemented than just the attributes absolutely necessary for the solution of the immediate task. Developers should also plan ahead and give consideration to the contexts in which the class might also be used.
- *Factoring*: When it is predictable that a class will later have alternatives with similar behavior, abstract classes should be planned from which the future classes can be derived. Especially when there is a large abstraction gap between existing (base) class and the new class (e.g., between graphic elements and editable text), intermediate classes should be introduced (e.g., non-editable text).
- *Simple functions*: Since existing functions that are not precisely suitable must be reimplemented (overwritten) in the derivation of new classes, complex operations should be avoided. It is better to decompose complex operations to smaller ones during their design: the simpler operations can then be overwritten individually as needed.

The highest possible generality can be achieved most easily if the design of reusable classes is handled by independent design teams that are not directly involved in the current project. Still, only actual reuse can determine whether this goal has been achieved.

References and selected reading

- | | |
|-----------------|--|
| [Abbott 1983] | Abbott R. J., Program De6si6gn by Informal English Descriptions, <i>CACM</i> , Vol. 26, No. 11, 1983 |
| [Apple 1987] | Apple Computer Inc., <i>Human Interface Guidelines: The Apple Desktop Interface</i> , Addison-Wesley, 1987 |
| [Booch 1991] | Booch G., <i>Object-Oriented Design with Applications</i> , Benjamin/Cummings, 1991 |
| [Coad 1990] | Coad P., Yourdon E., <i>Object-Oriented Analysis</i> , Prentice Hall, 1990 |
| [Dahl 1966] | Dahl O., Nygaard K., Simula – An Algol-based Simulation Language, <i>CACM</i> , Vol., 9, No. 9, 1966 |
| [Dijkstra 1969] | Dijkstra E. W., Structured Programming, Software Engineering Technique, <i>Report on a Conference</i> , Rome, 1969 |
| [Jackson 1975] | Jackson M. A., <i>Principle of Program Design</i> , Acad. Press, 1975 |



- [Meyer 1988] Meyer B., *Object-Oriented Software Construction*, Prentice Hall, 1988
- [Parnas 1972] Parnas D. L., On the Criteria to be Used in Decomposing Systems into Modules, *CACM*, Vol. 15, No. 12, 1972
- [Rechenberg 1984a] Rechenberg Predicate, Attributierte Grammatiken als Methode der Softwaretechnik, *Elektronische Rechenanlagen* 26, 1984
- [Rumbaugh 1991] Rumbaugh J., et al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991
- [Warnier 1974] Warnier J. D., *Logical Construction of Programs*, Van Nostrand Reinhold, 1974
- [Wirfs-Brock 1989] Wirfs-Brock R., Wilkerson B., Object-Oriented Design: A Responsibility-Driven Approach, *Proceedings of OOPSLA '89*, 1989
- [Wirth 1971] Wirth N., Program Development by Stepwise Refinement, *CACM*, Vol. 14, No. 4, 1971
- [Yourdon 1979] Yourdon E., Constantine L., *Structured Design*, Prentice Hall, 1979