# 3 Computer Programming

## 3.1 Computer Programming Languages

A programming language is an artificial language used by programmers and understandable (not necessarily directly) for computers. It is an intermediary in the transfer instructions from the programmer to the computer which may be the compiler or interpreter. Before run, the programmer's instructions are usually translated into machine language and only then are executed by a computer. In contrast to natural human languages, computer programming language must be clear so that only a single meaning can be derived from its sentences. The main objective of the study of programming languages is to improve the use of programming languages. This means to increase the programmer's ability to develop effective programs by growing the vocabulary of useful programming constructs, but also to allow them a better choice of programming language in the context of the problems to be solved.

### 3.1.1 A Very Brief History of Languages and Programming Paradigms

To be executed, a computer program should reside in primary memory (RAM). To be understandable for processor, a program should be represented in memory as a set of binary numbers – machine instructions. The instruction pointer, which points to the next machine instruction to be executed, defines the actual state of the computer. The execution sequence of a group of machine instructions is called *flow of control*. One can say that a program running on a computer is simply a sequence of bytes. Professionally, this is referred to as *machine code*. Programs for the first computers were only written in machine code; this period lasted until the end of the 1940s, and it is known in informatics as the pre-lingual phase. Each instruction of machine code performs a task, which is specific for the computer design, i.e. is hardware dependent. Modern computers still perform numerical machine codes, but they are created through the compilation of original programs, written by programmers in a high-level language. Direct writing of numerical machine code is not often done nowadays, because it is a painstaking, labor-intensive and error inclined job. The writing of machine code has been facilitated by *assembly languages*, which are more palatable to programmers. An assembly language is a low-level programming language and is specific to particular computer architecture like a machine code, but it uses mnemonic technique to aid information retention by programmers. The ability to program in assembly language is considered to be an indicator of a high level of programming skills because when the program is written in assembly language the programmer is responsible of allocating memory and managing the use of processor registers and other memory resources.

Unauthenticated
Download Date | 9/19/17 5:42 PM

In the history of programming languages, the 1950s-1960s are known as a period of exploiting machine power. It was at this time the first *high-level compiled programming* languages (more abstract than assemblers) came about. These were autocodes like FORTRAN and COBOL. To be executed, the autocode program had to be compiled and assembled. Autocode versions written for different computer architectures were not necessarily similar to each other. The languages created during this period are often referred to as *algorithmic languages*. They were designed to express mathematical or symbolic computations (e.g., algebraic and logic operations in notation similar to scientific notation). These languages allow the use of subroutines to aid in the reusing of code. The most common algorithmic languages are FORTRAN, Algol, Lisp and C. Programs which are written in a high-level programming language, consist of English-like statements with very limited vocabulary. The statements are controlled by a strict syntax rules.

From today's point of view, the most important event in the development of programming languages turned out to be what is known as the 'software crisis'. It is a state of conflict between increasing customer demands and the impossibility of distributing in time new, useful, efficient and cheap software systems by software developers. Such a situation began for the first time in the 1960s, and in accordance with some opinion, it continues today. To address the problems of the software crisis the discipline of *software engineering* came into being. In response to this situation, computer scientists have developed new types [19] of programming languages. The first are languages for business problem solving like COBOL and SQL, which were designed to be more similar to English, so that both programmers and business people could read and understand code written in these languages. The second, but not the last are education-oriented languages like Basic, Pascal, Logo and Hypertalk, which were developed to expand the group of software developers, and provided easy and convenient tools for programming time-sharing computers and later personal computers. It is worth noting that they were built to simplify the existing professional languages and were intended to be easy to learn by novices. Yet one important solution for the software crisis turned out to be the problem complexity reduction. The struggle to manage the complexity of software systems gave us object oriented languages (C++, Ada, Java). They carry on a hierarchy of types (classes of objects) that inherit both methods (functions) and attributes (states) from base type.

Historically, computer scientists have favored four fundamental approaches to computer programming, which are called programming paradigms. They describe conceptually the process of computation as well as structuration and arrangement of tasks, which have to be processed by the computer. Here they are:

1. *Imperative*, this is a machine-model based programming, because programs should detail the procedure for obtaining results in terms of the base machine model;
2. *Functional*, programming equations and expression evaluation;
3. *Logical*, is based on first-order logic deduction;
4. *Object-oriented*, this is programming with data types.

Unauthenticated
Download Date | 9/19/17 5:42 PM

An unambiguous, strict, universally accepted definition of these paradigms does not exist, so the classification of programming languages by these paradigms is indistinct. However, it makes sense to study some points concerning this classification, especially because each paradigm induces the particular way of looking at the programming tasks. Some programming language were designed to support one of the paradigms (e.g., Haskell supports functional programming, Smalltalk – object-oriented programming), other programming languages can support multiple paradigms (e.g., C++, Lisp, Visual Basic, Object Pascal), and some of them can be supported partially. Languages that support *imperative programming* (e.g., C, C++, Java, PHP, and Python) should be able to describe the process of computation in terms of statements that will change a program state. In this case, a program is like a list of commands (or the language primitives) to be performed; one can say about such languages that they are not only machine-model based but also procedural. The imperative paradigm has been nuanced several times by new ideas. The historically most recognized style of imperative programming is *structured programming*, which suggests the extensive use of block structures and subroutines to improve the clarity of computer programs. A variation of structured programming has become *procedural programming*, which introduced a more logical structure of program by disallowing the unconditional transition operations.

Alternatively, languages that support a non-imperative programming style should be able to describe the logic (meaning) of computation, mostly in terms of the problem domain without reference to machine model. Non-imperative programs detail what has to be computed conceptually, leaving the data organization and instruction sequencing to the interpreter of code. This style clearly separates programmers' responsibility (the problem description) from the implementation decisions. The non-imperative style is referred to as *declarative programming*; this term is a meta-category that does not enter the list of underlying paradigms. In this style we can include functional programming languages (Lisp, Scheme, ML, and Haskel) and logic programming languages (Prolog, Datalog).

In accordance with a functional paradigm, a program treats computation as the evaluation of mathematical functions and does not deal with state changes; such a program may be perceived as a sequence of stateless function evaluations. In accordance with a logic paradigm, a program is a set of sentences in the logical sense, which are representing facts and rules of inference regarding the problem domain.

Object-oriented programming (OOP) uses special data structures, which encapsulate the data and procedures as one; such structures are referred to as *objects*. Objects are derived from *classes*, which are abstract data types. Key programming techniques of OOP are based on data encapsulation, abstraction, inheritance, modularity, and polymorphism. The origins of object-oriented programming refer to 1965, when Simula language was created. The extensive spread of this paradigm only started the early 1990s. Most modern languages support OOP.

Unauthenticated
Download Date | 9/19/17 5:42 PM

Based on these four fundamental programming paradigms a few programming models (styles) were created to solve specific programming problems. For example, to program the GUI (graphical user interfaces) service, an event-based programming is introduced, where the control flow is subject to the events generated by the user (e.g., mouse clicks or screen touches). Event-driven programs can detect events derived from sensors and external programs as well. Of course, the handling of such events consists in carrying out planned actions. While writing event-driven programs, a programmer can use at the same time OOP style; so, in practice, models of programming usually are mixed. Finally, a particular language may support and the programmer may use not only one paradigm or style of programming but many.

At the end of this section, an important remark about the two subclasses of programming languages:

1. *Scripting languages* (e.g., Bash in Unix-like systems, Visual Basic for Application in Microsoft Office, Perl, and Python) run in special run-time environments and are used by programmers to mediate between programs in order to produce data; this category is fuzzy but in general scripting languages are helpful to automate a manual task during the use of programs or systems.
2. *Markup languages* (e.g., HTML, MathML, and XML) describe structure of data and are needed to control data presentation, especially in case of document formatting. It is a common opinion that markup languages are different from scripting languages and can not be classified as programming languages.

*The Popularity of Programming Languages*

There are no strict criteria relating to the popularity of programming languages. However, novice programmers particularly want to know, which programming language should be studied first, which language is widely used, and which language gives the maximum chance for well-paid job. The TIOBE Programming Community gives an answer by presenting the annual ranking [20] of programming languages. A top-ten list of languages with their average positions for a period of 12 months is shown in the table for last 30 years. Other indexes of programming languages popularity (e.g. RedMonk [21], PYPL [22] or TrendySkills [23]) are compatible with the TIOBE at least in regards to the position of the most popular languages.

| Programming Language | 1985 | 1990 | 1995 | 2000 | 2005 | 2010 | 2015 |
|---|---|---|---|---|---|---|---|
| C | 1 | 1 | 2 | 1 | 1 | 2 | 1 |
| Java | - | - | - | 3 | 2 | 1 | 2 |
| Objective-C | - | - | - | - | 39 | 23 | 3 |
| C++ | 12 | 2 | 1 | 2 | 3 | 3 | 4 |
| C# | - | - | - | 8 | 8 | 5 | 5 |
| PHP | - | - | - | 30 | 4 | 4 | 6 |

Unauthenticated
Download Date | 9/19/17 5:42 PM

| Programming Language | 1985 | 1990 | 1995 | 2000 | 2005 | 2010 | 2015 |
|---|---|---|---|---|---|---|---|
| Python | - | - | 22 | 24 | 6 | 6 | 7 |
| JavaScript | - | - | - | 7 | 9 | 8 | 8 |
| Perl | - | 19 | 9 | 4 | 5 | 7 | 9 |
| Visual Basic .NET | - | - | - | - | - | - | 10 |
| Pascal | 5 | 20 | 3 | 12 | 77 | 13 | 16 |
| Lisp | 2 | 3 | 5 | 15 | 12 | 16 | 18 |
| Ada | 3 | 4 | 6 | 16 | 15 | 26 | 30 |

As to the question which of the languages should be learned first, it appears that concentrating on one or a few specific and popular languages will be an excellent plan. It is good to remember that some languages such as Pascal or Basic were at first created as didactical languages and they become standard programming languages later. However, the current usage of classical Basic and Pascal has some essential faults, because they are not supported by modern operating systems and because they do not support the concept of object-oriented programming.

### 3.1.2 Syntax and Semantics of Programming Languages

Both syntax and semantics are terms used to describe essential aspects of language. Syntax is associated with the grammatical structure of language. Semantics refers to the meaning of language statements arranged according to that structure.

*Programming Language Syntax*
Like natural human languages, a computer language has a set of rules that defines how to create well-formed sentences; these rules constitute a *language syntax* that is the basis for ordering structure and elements in language statements. Syntax refers to the spelling of the language's programs. Precise syntax guarantees the grammatically correctness of a computer program. It is about formal (structural) and not about semantic correctness, but such formal correctness is absolutely necessary to create a workable and useable program, because meaning can be given only to language expressions which have been properly created. From a practical point of view, the syntax of programming languages is a set of requirements that must be satisfied by any meaningful program written in this language. In computer science, three levels of syntax are distinguished [24]: lexical, concrete, and abstract. The first level determines the set of all basic symbols of the language, i.e. names, values, operators, etc. The second level involves the rules for writing language expressions, language statements and whole programs. The third concerns the internal representation of the program. Syntax is defined by grammar, which is a set of meta-language rules relative to programming languages; informaticians generally use the *Backus-Naur*

Unauthenticated
Download Date | 9/19/17 5:42 PM

*Form* (BNF) to define programming language syntax. BNF was used for the first time to define syntax of Algol 60.

For describing the grammatical rules, the following BNF meta-symbols are used:

- Angle brackets to delimit non-terminal symbols <...>;
- Group of characters *::=* to shorten the key phrase "is defined as", the right arrow character can be used instead;
- The character / to separate possible alternatives.

There can be also used brackets [...] to indicate an optional part of the rule, curly brackets {...} to denote a fragment that can be repeated many times, parentheses (...) to group the definition of alternative fragments. Using these notation, one can describe the grammar rules in the form of so-called production: <symbol> ::= <expression containing symbols>. For example, the unsigned integer can be defined as:

*unsigned Integer ::= Digit | Integer Digit*
*Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9*

Based on this rule, all legal unsigned integers can be easy derived step by step. Such recursive logic allows the creation of a parse tree that will be useful when one needs to check whether the given number is indeed an unsigned integer.
Here are some basic syntactic concepts of programming languages:

- A *character set* is the alphabet of the language. In programming, the most common are two character sets: *ASCII* (a 7-bit data transmission code that covers the set of digits, letters in the Roman alphabet, and typographic characters) and *Unicode* (8-, 16-, and 32-bits versions of this code has become standard for modern computing and Web browsing) .
- *Delimiters* are characters used to denote the beginning and the end of syntactic constructs.
- *Identifiers* are strings of letters or digits typically beginning with a letter; they may exist as the name of a variable, file, data stream, data set, and so on.
- *Reserved words* (terminal symbols) are fixed parts of the syntax of a statement; they have an unambiguous fixed meaning in the context of a given language.
- *Expressions* are instructions that lead to operation on data and to returning computed values.
- *Statements* are the sentences of the language; they depict a computational task to be executed; the statement contains expressions, reserved words and identifiers.

A computer converts a sequence of program's characters into a sequence of tokens before compiling and running a process. This stage is known as lexical analysis or scanning, which has to identify the tokens of the programming language, that is to say reserved words, identifiers, constants and other special symbols. For example,

Unauthenticated
Download Date | 9/19/17 5:42 PM

the statement *gross_margin = net_sales – cost_of_goods_sold + annual_sales_return;* contains the following tokens:

| **Lexemes** | **Tokens** |
|---|---|
| *gross_margin* | identifier |
| = | equal_sign |
| *net_sales* | identifier |
| – | minus_sign |
| *cost_of_goods_sold* | identifier |
| + | plus_sign |
| *annual_sales_return* | identifier |
| ; | delimiter |

Directly after the lexical analysis, the program has to pass the syntactic analysis. This activity is called *parsing*. At this stage, the computer has to determine the structure of the program; it should examine whether the program meets the requirements of language grammar.

At the end of this section, here is one more example of a syntax definition of a structural program with the use of induction:

1. Let *e* be an expression. Let the assignment operator *x = e;* is a program.
2. If *P1* and *P2* are programs, then *P1P2* also is a program.
3. If *P1* is a program and *T* is a test, then *if T then P;* also is a program.
4. If *P1* and *P2* are programs, and *T* is a test, then *if T then P1 else P2;* also is a program.
5. If *P1* is a program and *T* is a test, then *while T do P1;* also is a program.

*Programming language semantics*

Generally, semantics is the study of meaning in language. In computer programming, semantics is a discipline that studies the formalization of programming language constructs by building their formal mathematical models. For creating such models, mathematical logic, set theory, λ-calculus, category theory, universal algebra, and others mathematical means are used. The idea is that mathematically transparent models guarantee the "correct comprehension" of programs by computers and as a result – the correct calculations. The official semantics of the language allow a formal analysis of its properties and is required for valid compiler design. The formalization of semantics is used not only to describe language, but also for the purposes of formal verification of programs in this programming language. From a practical point of view, the semantics of a programming language is the set of rules that determine in what order a computer should perform when executing a grammatically correct program that was written in the language. Alas, semantics is much harder to formalize than syntax.

Unauthenticated
Download Date | 9/19/17 5:42 PM

There are three prevailing approaches to the creation of formal semantics:

- Operational semantics (also called intentional semantics) should describe the meaning of a program by analyzing the process of program execution on an abstract or real machine. In this case, the meaning will be described operationally. The meaning of language constructs will be expressed in terms of transitions varying in an abstract machine from one state to another. The meaning of a correctly written program will be traced through a series of computation steps recorded during the processing of the input data.
- Axiomatic semantics should describe the meaning through the creation of assertions about algorithmic states and by proving the properties of a program based on formal logic rules. In such semantics, the meanings of the program will be represented by observed properties. The language constructs will be rated in terms of a set of formulas describing the state of the program. The meaning of a correctly written program will be a logical proposition that states a claim about the input and output.
- Denotational semantics (also called extensional semantics) should describe the meaning by building a detailed mathematical model of each language construct. The meaning of language constructs will be represented in terms of mathematical functions that operate a state of the program. The meaning of a correctly written program will be a mathematical function that calculates output data using input data. The steps made to calculate the output data would be unimportant.

The table below shows an example of the axiomatic semantics of a factorial program. Unfortunately, an operational semantics cannot be briefly written, even for this trivial example, which calculates factorial of n (in math, it is symbolized by an exclamation mark, n!).

| **Program** | **Semantics** | **Comments** |
|---|---|---|
| *int Factorial (int n) {*<br>*int i := 1;*<br>*int f : = 1;*<br>*while (i < n) {*<br>*i := i + 1;*<br>*f := f * i;*<br>*}*<br>*return f;*<br>*}* | $\forall n.n \geq 0 \supset \exists f.f = n!$ | It is the propositional logic expression[6], where<br>$\forall$ – means “for all” or “for any”,<br>$\exists$ – means “there exists”,<br>$\supset$ – means “implies”,<br>. – means “such that”. |

6 Whole expression means: For all $n$ such that $n \geq 0$ there exists a $f$ such that $f = n!$

Unauthenticated
Download Date | 9/19/17 5:42 PM

To give an example of deliberation when building operational semantics we can refer to the definition of syntax of a structural program that was given at the end of the previous paragraph. Let state $\sigma$ be a partial mapping of variables in the subject domain; then the value of the variable $x$ in the state $\sigma$ can be written as $\sigma x$. State $\sigma$ can be called a state of program $P$ if $dom\ \sigma \supseteq Var\ (P)$[7], which means that the σ-state assigns specific values to each variable of the program $P$. Let the value of the expression $e$ on the state $\sigma$ denoted by $\sigma\ (e)$ is an element of the domain, which is defined as follows[8] [25]:

If $e \sim x$ ($x$ is a variable) then $\sigma(e) = \sigma x$.
If $e \sim c$ ($c$ is a constant) then $\sigma(e) = c$.
If $e \sim o(e_1,...,e_n)$ ($o$ is a operation and $\sigma(e_i) = v_i$ is a program variable) then $\sigma(e) = o(v_1,...,v_n)$.

In considering this style, we can proceed to write the formal semantics of the program:

1. Let $P \sim x$, then $P(\sigma) = \tau$, where $\tau$ is a state defined in this way[9]:
   $\tau = \{(y,v) \in \sigma : y \neg\sim x\} \cup \{(x, \sigma(e))\}$,
   it means that, for all variables $y$, other than $x$, $\tau y = \sigma y$, and $\tau x = \sigma\ (e)$.
2. For collection of programs $P \sim P_1P_2$, then for every $\sigma$ one can define $P(\sigma) = P_2(P_1(\sigma))$.
3. For a short conditional construct $P \sim if\ T\ then\ P_1;$, it is if $\sigma \models T$ then $P(\sigma) = P_1(\sigma)$ else $P(\sigma) = \sigma$.
4. For a longer conditional construct $P \sim if\ T\ then\ P_1\ else\ P_2;$, it is if $\sigma \models T$ then $P(\sigma) = P_1(\sigma)$ else $P(\sigma) = P_2(\sigma)$ .
5. For cyclic calculation, it is less simple. Let $P \sim while\ T\ do\ P_1;$, $P(\sigma)$ is determined only if there is a positive integer $n$ and a sequence of states $(\sigma^i)^n_{i=0}$ such that:

*a.* $\sigma^0 = \sigma;$
*b.* $P_1(\sigma^i)$ is defined for $i = 0,...,n-1$ and $\sigma^{i+1} = P_1(\sigma^i);$
*c.* $\sigma^i \models T$ then and only then, when $i < n$.

Substantially, programming language is a means of communication between a human being (the programmer) and a computer (the performer). Each programming language has its own lexical (the lowest level of syntax), syntactic and semantic rules that must be followed by programmers designing a computer program. Ultimately, only operational semantic rules determine the sequence of computer actions that should be performed while executing a program.

---

**7** Here „dom" means domain and ⊇ is the sign of the superset. One can read this expression: "Domain of $\sigma$ is the superset for variables of the program $P$.

**8** Here and further ~ is the sign of similarity. It can be read „is similar to".

**9** Here and further ¬ is the sign of logical negation, ∈ means "is an element of", and / is used to abbreviate "such that".

Unauthenticated
Download Date | 9/19/17 5:42 PM

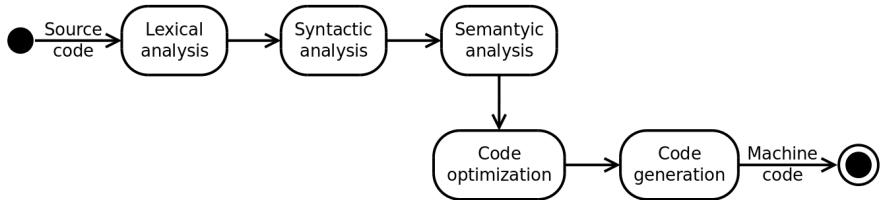


**Figure 8:** The compile process.

### 3.1.3 Design and Implementation

For *effective programming*, a programmer should understand four basic things about the language:

1. How the language is designed, e.g. what a program written in the language looks like (syntax);
2. What can program constructs mean (semantics);
3. How the language is implemented, e.g. how a program written in this language executes;
4. Whether or not, it is a suitable language to solve the problem.

*Language design* [26-27] is not easy work. The designer of a programming language is under pressure to fulfill the requirements of computability theory, of the base hardware, and of the programming community. In practice, until the early 1970s, programming languages were intuitively designed only to achieve efficient execution. This was due to the high cost of computers and the low cost of programmer labor. Currently, all manufactured software is constructed *efficiently*. Larger software systems are unthinkable without concern for work sharing in software teams, so the architecture of programs and their listings become a medium of communication and cooperation for team members. Executive efficiency is still an important goal in some applications, e.g. in the case of system programming in C or C++. To aid executive efficiency, the language designer should use some known technical features, e.g. static data types to allow efficient data allocation and access, or manual memory management to avoid an overload of the memory garbage collector, or simple semantics to simplify the structure of running programs. Furthermore, many other criteria can be mapped to the efficiency perspective, e.g. programming efficiency may be a metric for the ability of a language to express complex processes and structures.

The progress in theoretical informatics and software engineering created in the 1980s, the need for scientific principles of language design. Here are some principles that were developed for today's designers. Language design must:

– Be sufficiently general to embrace a wide range of common problems;
– Be easy to use to prove the correctness of solutions;

Unauthenticated
Download Date | 9/19/17 5:42 PM

- Guarantee that similar project items will look alike but different items will appear different;
- Allow a program solution to match the problem structure and be natural for the application. The program structure should reflect the logical structure of the problem. This is one of the key ideas of software engineering;
- Be expressive and allow for a direct, simple expression of conceptual abstractions. A program's data structures should be able to reflect the problem being solved;
- Allow for the international use of software systems;
- Be machine-independent, i.e. should be executable on any hardware. Programs written in Java are examples of this;
- Be internally consistent and consistent with commonly accepted notations. The language must be easy to understand and it must be easy to remember the language constructs;
- Guarantee that every combination of program features is meaningful. This language feature is called orthogonality;
- Be regular; do not allow strange, bizarre interactions, unusual restrictions, and so on;
- Be extensible; allow for the extension of language in a range of ways;
- Be precise in definitions to clearly answer possible programmer questions;
- Be standardized to increase the portability of programs. A very new language without experience and an old language with many incompatible versions cannot be standardized;
- It must not provoke large cost of use; the program creation, execution, translation, and maintenance should not require very special conditions.

Among existing programming languages, there is a lack of an ideal language that completely satisfies all of the above principles. Even in popular languages, one can point out some deficiencies, e.g. Pascal has no variable-length arrays – the length of the array is rigidly set in the declaration of array. Another example, C++ can convert from integer to float by simply assigning, but not vice versa. Declarations in C++ are not uniform because data declarations must be followed by a semicolon, function declarations must not. One more example, Java has no functions but has static methods. Simple data in Java are not objects, special primitive-wrapper classes exist for them.

*Programming environment*

Broadly speaking, a computer-programming environment supports the process of creation, modification, execution and debugging of programs. The programming environment is not an element of programming languages. It merely provides the tools that can help programmers in their job. Different programming environments provide various development tools, typically consisting of the source code editor, a

Unauthenticated
Download Date | 9/19/17 5:42 PM

compiler, a debugger, and help facilities. One of the main attributes of every good programming language is a *software development environment* [28] (SDE). This is the dedicated application program providing external support for the language, as often as not created by the authors of language themselves. Historically at first, the language-centered interactive environments supported a single programming language with its apparatus. Initially, this was a simple command based software. Later, together with the formation of the corresponding programming paradigm structure-oriented environments were developed that introduced syntax-directed code editors. After that, *toolkit environments* were introduced that provided a collection of language-independent tools. This was a very important modification, which lead to modern composite development environments. This allows programmers to change language without changing their working environment. Normally, toolkit environments have even more tools needed for team development, e.g. for tracking and controlling program changes (an activity referred to as *software configuration management*).

The commonly used at present the term IDE (integrated development environment, some would call "integrated design environment" or "interactive development environment") emphasizes, that all independent tools of programming environment cooperate with each other. Typically, in addition to a source code editor, compiler, and debugger, IDE provides a central interface and includes a runtime environment. This construction makes easy quick code changes, recompilations, and test runs of programs that are developed easy. Modern IDEs support so-called *visual programming* that helps programmers in code creation by visual modeling of programming building blocks, especially in the case of programming of graphical user interfaces. Professional IDEs contain also a mechanism of *intelligent code completion*, which suggests to programmers subsequent items in the code according to the current context and thus accelerates the encoding process. The IDE can be treated as an application for easy work with separate sets of development tools. One such set of tools is called SDK (software development kit). Usually, SDKs are dedicated to support a particular *development platform*, i.e. particular hardware (PSM SDK) or operating system (Android SDK or Ubuntu SDK). SDK may support code writing for very popular applications (Microsoft Office 365 SDK or LibreOffice SDK) as well.

Ready building blocks of code are distributed by means of SDK-program – program libraries and software frameworks. These are intended to achieve occasional code reuse. Typically, the *program library* file contains the archived definitions of all-purpose data types and subroutines that have been proven in previous implementations. The elements of the library can be called from the level of original code as if they were a normal part of the program. A programmer might add a program library to his application to achieve more functionality or to employ a computation procedure without programming it. The programmer should know the library used because the library's objects and methods will be instantiated and invoked by the

Unauthenticated
Download Date | 9/19/17 5:42 PM

custom application. The programmer needs to know precisely which objects to instantiate and which methods to call in order to achieve the desired effect. A program library may be static or dynamic. A *static library* is inserted into the program at link time in contrast to a *dynamic library*, which is loaded into memory of the running computational process on demand.

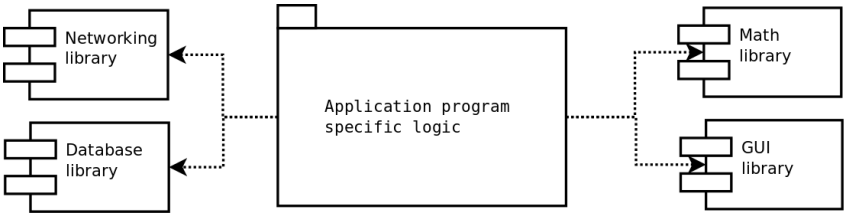


**Figure 9:** The library architecture of a program.

In contrast to a library, a *software framework* is a domain-specific programming environment that supplies reusable code for building a new functionality. In fact, frameworks are the skeletons of applications or in other words semi-finished products – application programs that can be completed by the configuration of ready software components. The programmer does not need to have detailed knowledge about the construction of the framework used. The objects and methods of the framework are unfamiliar to his application and the framework defines the flow of control for the application itself. Of course, the programmer can override the framework-implemented features and can customize the framework behavior. In this case, the programmer should be familiar with the framework's construction. Due to its construction, a framework can support the reuse of architecture and detailed design. An integrated set of framework components supports a family of applications with similar business goals. Often, frameworks aid the development process by providing ready functionality. For example, it may be complete service of a user management process or of database access. Such solutions help developers to focus on the more important custom details of software design.

Unauthenticated
Download Date | 9/19/17 5:42 PM

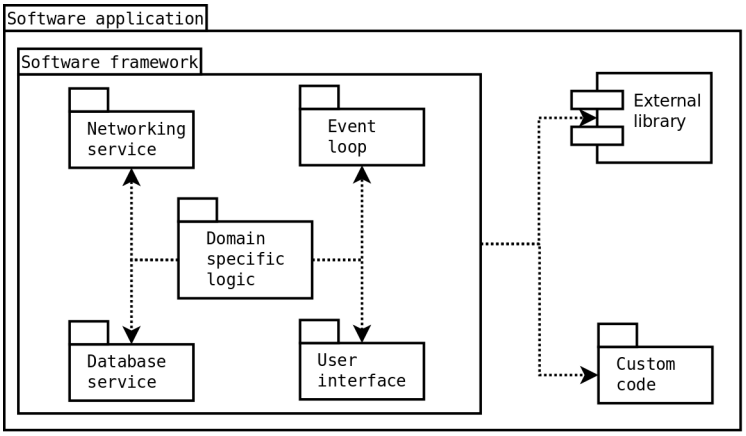


**Figure 10:** The framework architecture of a program.

## 3.2 Software Engineering

The term *software engineering* was introduced into public circulation for the first time at conferences organized by NATO in 1968 and 1969. The software crisis, that has been previously mentioned, was discussed. The term was introduced to clearly suggest that software developers should structure and manage a process of software development similar to others industrial sectors. It was a suggestion to move away from an art and get closer to the craft (business) of programming. An engineering approach to software development needs to reduce the costs of software development and lead to software that is more reliable [29]. Today, software engineering (SE) is a scientific discipline that deals with the development, deployment and maintenance of computer software. It is a part of computer science. SE promotes the methodical, disciplined, and quantifiable approach to the development of software, i.e. a well defined development process consisting of traceable actions and tasks with predictable size. Software engineers have to take into consideration a lot of circumstances, e.g. the type of hardware the software will be used on, operation systems by which the software will be handled, and also the specificity of the company in which the software will be deployed or the qualification level of the end users who will operate the software.

The primary objective of SE is the production of software systems in accordance with a requirement specification; it should be done on time, and within budget. One can say that software engineering is an instance of systems engineering (an interdisciplinary field). However, carrying out its own tasks, software engineering utilizes its own development processes and design methods for software.

Unauthenticated
Download Date | 9/19/17 5:42 PM

### 3.2.1 Software Development Process

Software development teams follow a specific theoretical life cycle in order to guarantee the best quality of created software product. This life cycle was build around the detailed study of the experiences of many software teams. It assumes similar knowledge from general engineering as well. Normally, the *life cycle of software* is comprised of the following phases:

1. Requirements gathering, analysis and specification
2. Software design
3. Implementation (coding)
4. Testing (validation) and integration
5. Deployment (installation)
6. Maintenance
7. Retirement (withdrawal)

Each phase produces deliverables required by the next phase in the life cycle, e.g. requirements are translated into software design, code is produced in accordance with blue prints, testing verifies the code taking into account the requirements, and so on. Even the withdrawal from circulation of old software generates useful information for the requirements and design phases of the next version of software.

In the first phase, business requirements are gathered. This means interviewing a stakeholder about the software that is planned and its business goals. The answers to the following questions will be significant: who is going to use the software, how will they use the software, what data will be inputted into the programs, what data should be outputted, and which action or data should be restricted? These answers constitute requirements. They are analyzed for their validity, cohesion and the possibility of realization in the planned software. The analysis ends with a document called software *requirements specification* (SRS). This document is an essential part of the contract for the software and should clearly determine "what the software should be."

The second phase is about blue prints. At the time, the software design is arranged in accordance with the requirement specification. One of the most important tasks is to determine the architecture of the planned of software. The designer should describe a high-level software architecture that outlines the functions, relationships, and interfaces for major components. This means that the designer should decide on modularity of software, on the possible need to use libraries, frameworks, design patterns, and the ready to use components. Based on these decisions, the third phase will be organized – coding (implementation). This is the longest period of the software development life cycle. Software design documents (blue prints) are guidelines for programmers in the process of writing code. Coding is done in collaboration with designers, because planned modules or other units of code may vary with respect to those documented in the preceding phase.

Unauthenticated
Download Date | 9/19/17 5:42 PM

In the fourth phase, ready-made units of code are integrated (combined one with another so that they become a whole) and tested against the requirements. Testers have to make sure that the product is really solving the needs formulated by stakeholders. At a further stage other tests are performed to ensure a high quality of product. After successful testing, the software can be delivered to the customers to be used. This fifths stage is not a simple as it may seem. A customer's hardware and infrastructure should be ready to implement the new software. The customer's personnel should be trained in the use of the new software. The customer's data and archives should be available for the new software. After resolving all the problems, software developers often have to adapt product to local requirements.

In the sixth phase, the customer operates the installed software. During the period when the customer is using the software, problems may arise and they need to be solved (occasionally very quickly, e.g. when they relate to information security). Developers solve such problems and update the product. This activity is called maintenance. With time there are so many fixes (patches) in the software that the developers decide to release a new optimized version of software. This is a software upgrade. The old version is withdrawn, and the new version is deployed. Sometimes, this means the complete withdrawal of software from circulation for business reasons, e.g. the production of a given software ending.

The modern software development process tends to run iteratively through some of the seven phases rather than linearly. This is the most common flow of work: detailed design, code construction, unit testing, integration, complete software testing and release. Iterative production of software allows for managing software projects of high complexity effectively. Successive iterations can be planned based on the priorities identified by the stakeholders in relation to the functions of the software. The most important functions should be scheduled for the first releases, and the less important – for later. The necessity of such planning usually results from the limited resources of software teams.

In implementing an operational business process methodology, software developers use different approaches to the organization of the development process. To visualize and then study the development processes, several models have been elaborated. These models and corresponding approaches are referred to as *software development process models*. The best-known classical approach is represented by the waterfall model (Herbert D. Benington, 1956). This was often used for the creation of large software systems; metaphorically, it looks like a cascade waterfalls. This is why some people call it a cascade model. This model describes a linear process in which the activities resulting from the software life cycle are executed one by one. The model contains the ability to go back to one of the preceding activities when conditions make this necessary or desirable. The main negative aspect of the waterfall model is the trouble of making convenient changes when the process is ongoing. One can say that this process is inflexible.

Unauthenticated
Download Date | 9/19/17 5:42 PM

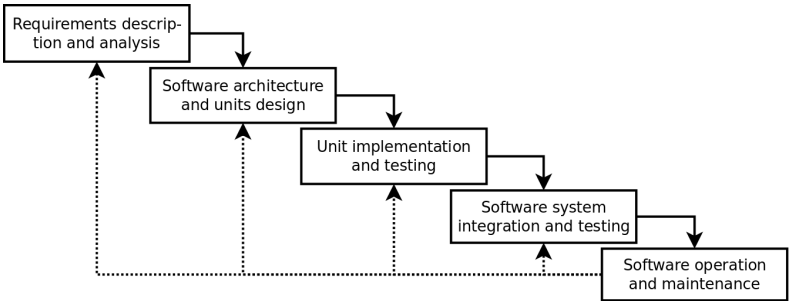


**Figure 11:** The waterfall model of software development process.

The idea of a more flexible process shows another classical model of software development process – the spiral model (Barry Boehm, 1986). Here, the process is represented as a set of loops – each loop leads the team to the final product, and the current loop execution phase constitutes an indicator of the progress of the stage. This model does not establish fixed phases of the life cycle, for instance requirements, design and implementation. Loops are selected depending on what is required. The spiral model provides an explicit assessment of the risk of failure, so managers can respond quickly to changes in the project. One can say that it represents the process driven by risk management.

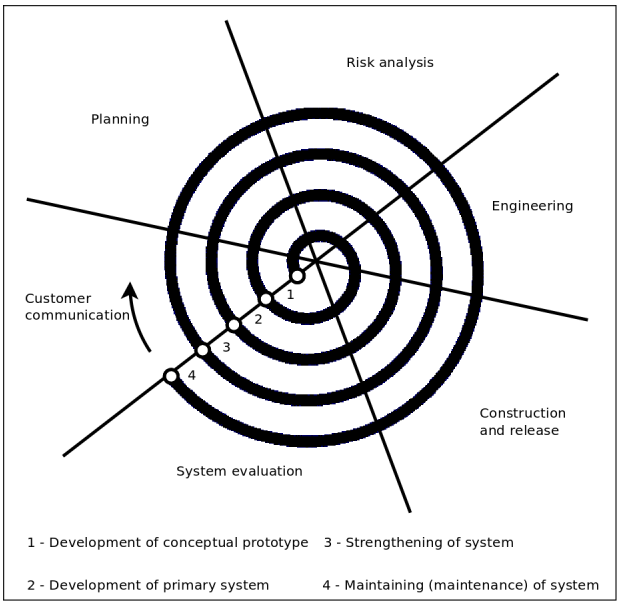


**Figure 12:** The spiral model of software development process.

Unauthenticated
Download Date | 9/19/17 5:42 PM

Both of these models now have a generally historical significance. New experiences and new knowledge have helped expand the idea of software production. The most influential idea has turned out to be agile software development. This was implemented in the methods called *Scrum* (Jeff Sutherland and Ken Schwaber, 1995) or *extreme programming* (Ken Beck, 1999) and others (which fall between them conceptually). An agile paradigm has changed priorities in software development. More attention should be paid to people and interactions than to procedures and tools, to functioning software than to complete documentation, to customer collaboration that to contract negotiation, to responding to changes than to following the plan. All agile methods take seriously active user involvement in software development. Customer representatives are regular participants in all activities of the project team. Development works through small increments of functionality (frequent releases with completed features). This allows the evolution of requirements and timescale fixing simultaneously.

An important attribute of agile methods is that software teams are empowered to make decisions. One can say that agile methods actively use the *human factor*. In this context, the aforementioned Scrum is an agile development method, which uses the concept of a team-based development environment. It emphasizes team self-management, which in conjunction with experiential feedback allows building the correctly tested product gains within short intervals. At the same time, the necessary role of product owner in Scrum team ensures the representation of the client's interests. In contrast to Scrum, extreme programming (XP) is a more radical agile method. It concentrates more on the development and test phases of the software engineering process. XP teams use a simplified structure of planning and monitoring activities to choose what should be done next and to forecast project finalization. An XP team usually delivers very small increments of functionality. It is focused on continuous code improvement based on user involvement in the development process. An unusual feature of this method is so-called pair-wise programming. Two programmers, sitting side by side, at the same machine, create production software (one can say this is continuous reviewing of code).

In conclusion, software development processes combines all activities associated with the production and evaluation of software. To understand the development process abstract representation, a model of this process, which shows the structure (organization) of typical activities (e.g. specification, design, testing, implementation and installation) is used. There is no possibility to closely classify all processes and their models, because they were created in a spontaneous manner, in the context of competition. To understand the essence of software engineering, one can look at some historical examples (the best example is the Rational Unified Process). A good idea might be to analyze several international standards on this issue, such as ISO/IEC 12207 "Systems and software engineering – Software life cycle processes" or ISO/IEC 15504 "Information technology – Process assessment".

Unauthenticated
Download Date | 9/19/17 5:42 PM

*The Rational Unified Process*

Rational Unified Process was developed in the second half of the 1990s. This new model of the software engineering process was distributed as a native key component of professional software (development kit, set of tools) targeted at software developers. It was a very important event in the history of object-based software systems. Instead of providing a large number of paper documents, the Unified Process concentrates on the development and maintenance of semantically rich models [30], which must represent the software system being developed. Rational software tools delivers full support for such models, using the Universal Modeling Language (UML). The modeling of large information systems with the use of this language is described in the next chapter. RUP is not a constant process. Developers can fit it to the scale of the project and adjust it to customer requirements. The Rational Unified Process has adopted the best-known practices of modern software development and assists developers in implementation of these practices.

RUP is an iterative development process. This means that a whole software project is divided into several mini projects (iterations) planned one by one. It helps to recognize the changing requirements and to organize early risk attenuation. After each iteration, developers can correct errors in software and be more accurate with their plans. Each RUP iteration increments the software functionality. This means that the software is evolving to be the result of cumulative effort. Such an evolutionary approach to developing software gives developers a chance to deal with the reliability, stability, and usability of the product.

RUP's standard sets out four phases of the project. The first phase, called inception, refers to capturing the initial requirements, to analyzing initial risks and cost benefits, and to defining the project scope. At this stage, developers arrange an initial architecture of the software system. Then they build a prototype which is not reusable and begin creating the key models – a use case model and a domain model. The first model is used to analyze the required functionality of the software, while the second one – to analyze the problem itself with its surroundings. The results of this analysis will represent the problem in the design of software. One can say that this phase deals most with business modeling and project management, not with the design of software.

The second phase, called elaboration, continues capturing and analyzing the requirements, modeling use cases and problem domain. During this phase, scenarios of user activities are developed, a glossary model of the project is created and prototypes of user interfaces are fleshed out. It is important that at this early stage, even before the meticulous, detailed design, the customer (users) will be able to do some work by using the real views (graphical user interfaces) of the future application. As a result of such interactions with trial software, serious semantic errors are usually detected that were overlooked during requirement gathering. One can say that this phase is concentrated on requirements and on design of software.

Unauthenticated
Download Date | 9/19/17 5:42 PM

The third phase, called construction, focuses on implementation of the design and on testing the created software units. The application features, remaining after the elaboration phase, are developed and integrated into the complete product. At this time, the software architecture is subjected to continuous validation; the same applies to data repositories. All planned user activities and all sequences of object interactions are modeled and validated as well. It is a typical manufacturing process with resource management and operation monitoring, which must optimize costs, schedules, and quality of software development. This phase begins a conversion from intellectual property (created in the previous phases) into the deployable product.

The fourth phase, called transition, relies on the relocation of the software to the users' groups. It includes making software available for users (e.g. via websites), and in some cases the installation, training of end users, and technical support. Once users have begun to use software, problems begin arise. The software team is required correcting mistakes and preparing new releases. At the same time, developers have to work out features that were postponed.

A static perspective of the Rational Unified Process shows development process activities, artifacts and workers in context of workflows. This perspective has to describe the behavior and responsibilities of team members using the template "who is doing what, how, and when". The same team member can be planned as a few workers (to play several roles), e.g. as a designer he or she may deal with object design and as a use-case author he or she may detail a use-cases. During the development process various artifacts are produced – standalone pieces of information like source code or architecture documents; workers produce or modify or use them performing their activities. Finally, a workflow is a meaningful sequence of those activities, which produces a result (a value like a document or program unit). RUP suggests nine core process workflows, which arrange all workers and activities. These workflows are divided into six core engineering workflows (business modeling, requirements, analysis and design, implementation, testing, and deployment) and three core supporting workflows (project management, configuration and change management, and environment).

### 3.2.2 Software Design Methods

Software design is a core (major) phase of the life cycle of software and at the same time, it is the most "mysterious" process of converting the requirement specification into an executable software system. The practice of software teams is not as spectacular as it may seem. Here, design means developing a blueprint (a plan) for a procedure and mechanisms that will perform the required tasks. It is different from software programming – design should be made before programming and programming (or in other words coding) is the realization of design. Software design is the most critical phase affecting the quality of the created software. Developers use particular design

Unauthenticated
Download Date | 9/19/17 5:42 PM

methods to make decisions about software design – about the structure of code, data, and interfaces along with their internal and external relationships.

A modern economy needs programming support for systems with global scale. This means that software designers have a high degree of professional responsibility. Software design methodology provides a logical and structured approach to the design process; it establishes the necessary sequence of dedicated activities using both text and graphical notation. It is particularly important for complex software projects (e.g. in case of so-called software intensive systems) to be carried out in accordance with the appropriate method. Historically, different approaches have been taken to develop high-quality software solutions for dissimilar problems. In modern practice, these different approaches are sometimes combined in a logical manner to get a complex method for a real problem. Names such as data-oriented design, function-oriented design, object-oriented design, responsibility-driven design, and domain-driven design correspond to different priorities given to the software designers. It is worth noting that these different approaches require different knowledge about the problem domain, which may substantially affect the work that must be performed at the analysis stage. In all cases, the software design process should comprise some typical activities: architectural design, abstract specification, interface design, component design, data structure design, and algorithm design. A further, vitally important activity is architectural design.

Architectural design [31] has to establish the overall structure of a software system, i.e. its units like sub-system, framework, module, interface, and other components as well as their interconnections and interactions. A designer has to identify these units, because before architectural design they are simply unknown for developers. There is no single best-known software architecture even for projects with the same subject. Almost every software project is unique in this respect. It is possible, that this is a result of the specificity of information engineering in comparison with other areas, e.g. with the field of civil engineering. At the same time, there are some architectural patterns in software architecture, and software product lines usually share a common architecture. Architecture is a conceptual thing (it is very fundamental) and exists in some context. For this reason, architecture design is important as the earliest set of design decisions used to negotiate with stakeholders about such essential things as the organization structure associated with a software system or as quality attributes of software.

A software architect may prefer some architectural styles, which are specifications of architectural component and connector types and constitute patterns of their data transfer and runtime control. If software architects find themselves in a particular context with a specified problem, then for some particular reasons they may decide to apply a special pattern. For example, they can choose architectural style of repository to manage richly structured business information, a layered style to create a system for effective management of the hierarchical structure of users, or model-view-controller style to separate the user interface from the application.

Unauthenticated
Download Date | 9/19/17 5:42 PM

An abstract specification means the formal registering of a very high level of design, i.e. there are documented terms, definitions and information models, along with general behavior of software. It is made after reaching a consensus between team members and stakeholders on software architecture and on other general things at the same level of abstraction, e.g. on services that will be produced by software. Usually, the abstract specification is written in formal language with precise, unambiguous semantics. Natural language is unsuitable because of the ambiguity of its statements.

Interface design activity embraces planning of the connection and communication between software modules. It concerns the communication of a software module with hardware and a software module with a user as well. Modules are planned as independent units of the software system, so the interface design should support such independence, while ensuring reliable transmission of data and control signals. Not all interfaces need to be controlled, but the critical interfaces of a software system should be specified at the design phase. The appropriate document is called the interface control document (ICD). The ICD can be modified only under specific, previously defined conditions because it is used by developers for software unit implementation. The specification of some interface characteristics may require an agreement between stakeholders and developers.

A special case is user interface design [32]. This activity concerns the visual, aural and tactile communication between the user and software. Each of the three communication channels has its own specificity. As in the general case, the ICD is the right document to record the findings. Here, the specification is written in two different languages – a presentation language for computer-to-human and an action language for human-to computer transmission. Developers use several conceptual types of user interfaces: natural language, question and answer, a menu, form-fill, command language, and finally graphical user interfaces (GUI); the last one is the currently the most popular.

### 3.2.3 Computer-Aided Software Engineering

Help in the development and maintenance of applications by means of professional software is called *computer-aided software engineering* (CASE). This concept was introduced in the 1970s. In fact, it refers to the idea of automating the most time-consuming tasks of the software development process by introducing a collection of useful tools, an organized layout, and craftsperson. CASE allows for *rapid software development*, which answers to the needs of businesses that have to change dynamically their offers in response to market demands. The main goal of CASE-technology is the distinction of the design process of software products from the encoding stage and the following stages of development, to automate the development stages to be less creative and more predictable. The term CASE is also used as a collective name for a new generation of developer tools that apply precise engineering principles to software projects.

Unauthenticated
Download Date | 9/19/17 5:42 PM

Software engineers need *analytical tools* which are useful in software development (e.g. for *top-down design* and for *cost-benefit analysis*), and *products tools* that assist the software engineers in developing and software maintenance. There are two known categories of CASE tools: upper CASE and lower CASE. The first is focused on automation of planning, requirements, specification, and design activities (mostly conceptual), whilst the other on the automation of implementation, integration, and maintenance of software (mostly practical and applied). There is also the category of integrated CASE tools (I-CASE), which assist the full software life cycle. In addition to CASE tools, one can distinguish some CASE building blocks, for example an integration framework, which allows tools to communicate between themselves. Another example may be portability services, which allow tools and their integration framework easy migration across different hardware platforms and various operating systems.

Here are some popular categories of CASE tools with a very short annotation to understand the scale of CASE workshop:

- *Business process engineering tools*, which are useful for presenting business data objects, their relationships, and flow of data between business sectors of corporation. Business process modeling activity is valuable in the case of software projects for complex and complicated institutions.
- *Project planning tools*, which are needed for software project scheduling, allocation of effort (task sharing), and costs estimation.
- *Risk analysis tools*, which aid project managers in the recognition and detailed examination of factors that may potentially hinder and even interrupt successful completion of the project.
- *Requirements tracing tools*, which are used to provide methodical requirement specification and lifetime documentation of a requirement.
- *Interface design and development tools*, these are useful for rapid prototyping of graphical user interfaces and for the detailed development of typical GUI modules.
- *Program generator*, this is a computer program that is used by developers to create other computer programs. It is not automatic programming in the full sense of the word – it only means that a software engineer can write a code at very high abstraction level.
- *Integration and testing tools*, these aid developers in combining the program units as groups and testing them, such activities can expose problems with interface design before entire applications have been formed.
- *Reengineering tools*, these are needed for the controlled restructuring or rewriting a part of code (*code refactoring*) without changing its functionality; usually they are utilized for improving the quality of software.
- *Reverse engineering tools*, these can recreate the physical and conceptual models from source code and aid software actualization (modernization).

Unauthenticated
Download Date | 9/19/17 5:42 PM

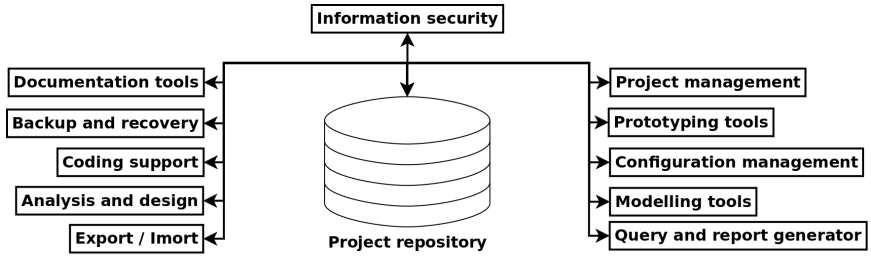


**Figure 13:** The structure of a typical CASE environment.

CASE tools aid various sectors of software projects: the modeling of business aspects of software, the design and development of code and interfaces, the verification and validation of software units, configuration management, and even project management. In terms of complexity, there are dedicated workbenches that integrate some CASE tools to assist specified activities of the software process, and environments that integrate CASE tools and workbenches to assist the whole software process. By using software tools, developers improve software quality, reduce time and effort, but there are some problems, e.g. CASE tools bring inflexibility in the documentation of a project, while it is generally known that completeness and syntactic correctness does not necessarily equal compliance with requirements. In addition, commercial CASE tools themselves are very expensive, and this increases the cost of software produced.

The selection of CASE tools should be carried out with careful consideration not only for technical requirements but also management necessities. A structured set of CASE tool characteristics is defined by international standard ISO/IEC 14102:2008. These characteristics are related to life cycle process functionality, to CASE tool usage functionality, and to general characteristics related and not related to the quality of software. Just after the publication of this standard, CASE-tools began to be considered as software tools supporting the processes of the whole software life cycle.

## 3.3 Information Technology Project Management

As discussed in the previous section software development is a key activity and occupation in the wider area of *information technology* (IT). Software has become a core component, which enables people to use all forms of machinery useful in creation, storage, exchanging, and consumption of information in its various forms. Information technologies have grown in the last half of century at a very high tempo. Generally, this growth has takes place in the organizational form of IT projects. The term *IT project* is used as a collective name for all ventures associated with

Unauthenticated
Download Date | 9/19/17 5:42 PM

development, implementation and deployment of computer software and hardware in any sector of the economy, as well as other projects. It has an assigned goals, a start and end date, detailed milestones and so on. This is usually a temporary attempt to create a matchless product, service or environment, e.g. to develop a new ecommerce site, or to automate some tasks previously performed manually, or to replace an aged information system.

Generally, an IT project is carried out in order to obtain a unique product or service. The business which has ordered the project is the first recipient of the project results, because it expects to increase its profits and competitiveness through the implementation of the latest IT solutions. An IT project is a complex machinery – it has a unique purpose, it has to be prepared and monitored all the time. It should have a primary customer or sponsor. It is developed using progressive elaboration. It may be terminated. It ends when its objectives have been reached. It requires a lot of resources. Some larger companies have their own IT departments for managing such projects. Other companies prefer to cooperate with external independent contractors – firms specializing in the organization and implementation of IT projects. It is natural that orderly IT project management has to be inherent to the IT strategy of an organization and is usually under the leadership of the Chief Information Officer, because the CIO is responsible for information technology and computer systems that support enterprise goals.

The Standish Group, independent analysts of IT project performance publish the CHAOS report [33] from which it follows that in 1995 only 16.2 percent of IT projects were successful in meeting scope, time, and cost goals. In 2004 [34], the percentage of successful projects only increased to 29. In 1995 over 31 percent of IT projects were canceled before completion. In 2004 the percent of failed IT projects decreased to 18. It was found that larger projects have the lowest success rate and appear to be more risky. It has been observed that in the last decade computer technologies, business models and markets change so rapidly that a project that takes more than one year has a chance to be outdated even before completion.

The CHAOS reports have shown the most important factors of IT project success. Among these are user involvement, clear business objectives and statement of requirements, executive management support, an experienced project manager and project management expertise, and finally, a formal methodology of project management. At the same time the main reasons for project challenges have been formulated (generally, lack of user input, incomplete or changing requirements, unclear objectives, and unrealistic time frames and expectations). The reasons for project failure include: incomplete requirements, lack of user involvement or lack of resources and executive support, and lack of planning. Based on analysis of results from thousands of projects, IT experts provide recommendations on how to increase IT project success, and quantify the relative contribution of a scheme of factors that will affect the project's success. Such knowledge is generalized within the discipline called *information technology project management* [35].

Unauthenticated
Download Date | 9/19/17 5:42 PM

### 3.3.1 The IT Project Management Stages

Founded in 1969, a professional membership association for the project, program and portfolio management profession – the Project Management Institute [36] (PMI) – defines *project management* as the application of knowledge, skills, tools and techniques to project activities to meet project requirements. IT project management is organized and conducted by project managers; especially at the levels close to the software development team. These are informaticians trained to plan, delegate, and monitor responsibilities (tasks) between project team members. The other important category of people associated with the project is called project stakeholders. These are the people involved in or affected by project activities, e.g. project team members, the project sponsor, support staff, customers, users, and suppliers. One of the most important responsibilities of a project manager is to establish cooperation with all stakeholders. Another key responsibility is to establish and supervise the project scope (as shown in Figure 14). The project has been successful when it achieves the objectives according to their acceptance criteria, within an agreed timescale and budget [37]. To define a project scope, one must first identify the project objectives, goals, sub-phases, tasks, resources, budget, and schedule. After establishing these things, it is necessary to make clear all the limitations and plainly identify any aspects that do not have to be integrated in the project. The project scope must be clear to the team members, stakeholders, and senior management.

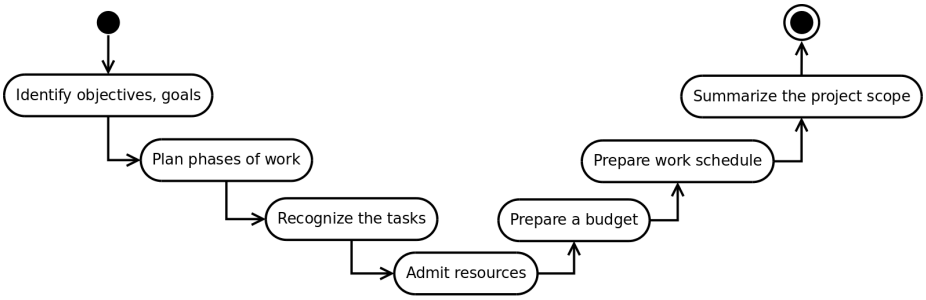


**Figure 14:** Defining the scope of IT project.

IT project management combines the knowledge and methodology of traditional project management and software engineering to increase the effectiveness of information technology projects. According to the PMI, the project management process is traditionally guided through five general stages: initiation, planning, executing, controlling and closing (as shown in Figure 15).

Unauthenticated
Download Date | 9/19/17 5:42 PM

Here is what happens during the next stages of the project life cycle:

- During the first, initiation stage, a conception (idea) for a project is examined to determine whether a new software system will benefit the organization; here, decision makers have to recognize if the project can be completed.
- During the second, planning stage, such important management documents as the *project plan* with a defined *project scope*, and a *project charter* may be developed in a written and approved. This means that the work to be performed becomes outlined. Within the planning period, a team should prioritize the project tasks, calculate a budget and create a schedule, and determine what resources and when they will be needed. Among other things, a scheduling of the software life cycle is completed at this stage as well.
- During the third, execution stage, the project is launched, project team members are informed of responsibilities and resources are distributed. The development team starts to work.
- The fourth stage is about project performance and control. At this time, the project condition and growth will be compared to the approved project plan, and the use of resources will be compared with the approved schedule. Throughout this period, there may be needed schedule adjustment or resources reallocation. This period ends with the final delivery of the software system.
- During the fifth stage, the project is closed; after project tasks are completed and the client has approved the IT product. An evaluation is necessary to highlight project success and learn from project history.

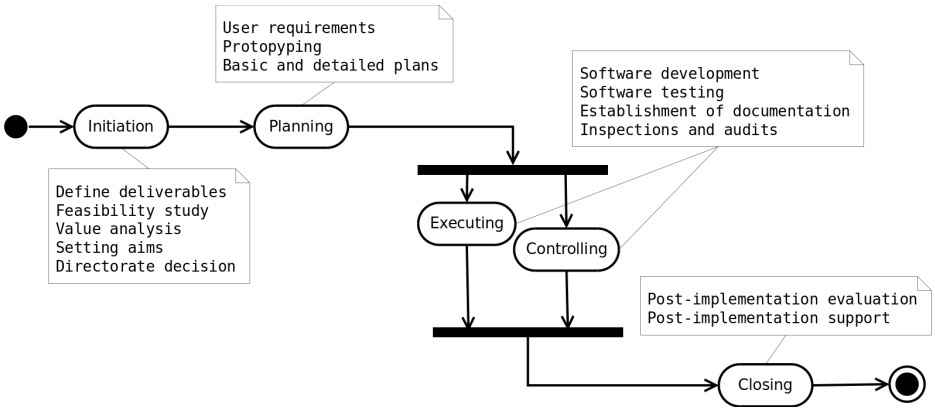


**Figure 15:** Crucial stages in an IT project life cycle.

Unauthenticated
Download Date | 9/19/17 5:42 PM

The life cycle of software is connected to the stages of project management. The software lifecycle is one of the most important of several other engineering activities. Besides software development activities, IT projects may comprise the erection of information infrastructures, the creation of telecommunication channels, data repositories and knowledge base creation and development or even multimedia production. It is not difficult to see that the IT project management process looks formally, as if it was a technological process developed for machines, not for the people. The main advantage of using such formal management is an improved control of all resources (financial, physical, and human as well), and thereby improved productivity and customer relations. Practice has shown that formal management to some extent improves project efficiency – reduces project costs, shortens development times, and improves the quality of software products.

### 3.3.2 Approaches to Managing IT Project Activities

An intuitive approach to IT project management should be based on standardized knowledge. Professional project management requires specific knowledge, which is empirical. This means that such knowledge is based on the best practices. Project managers confirm their level of expertise in this field by obtaining the relevant certificates. One such certificate issued by the Project Management Institute; PMP credential (Project Management Professional) is very popular among IT project managers. Here are examples of other respected organizations that carry out the training and certification of project managers: the International Project Management Association (IPMA), the Association for Project Management (APM). It should be remembered that the set of best practices used by these organizations are necessary, but not sufficient to develop the right approach to an IT management project. This knowledge does not guarantee the success of the project, even when conducted by a certified manager. These best practices are neither legally binding nor is one forced to adopt them. For example, PMI does not undertake to guarantee of the performance of any individual manufacturer of products or services by virtue of its standards.

A methodological order and systematized approach to IT project management should be based on a standard method. PRINCE2 delivers the most common generic methodology, especially among large scale IT projects. The name PRINCE2 is an acronym for PRojects IN Controlled Environments. This is a de facto process-based method for effective project management [38]. It defines what, when, how, and who can arrange activities of the continuous process. The PRINCE2 method is in the public domain, and offers a non-proprietorial best practice guidance on project management. One of the fundamental principles of PRINCE2 is a focus on an unceasing business justification which means that it is reactive and adaptive. It is characterized by a flexibility which can be applied at a level appropriate to the IT project. Besides, the method uses the product-based planning approach, which is very important in

Unauthenticated
Download Date | 9/19/17 5:42 PM

case of software. The fact that PRINCE2 is compatible with PMP is important as well. Unluckily, PRINCE2 does not provide specific guidance to the IT area and it has a gap in transitioning project outputs to business.

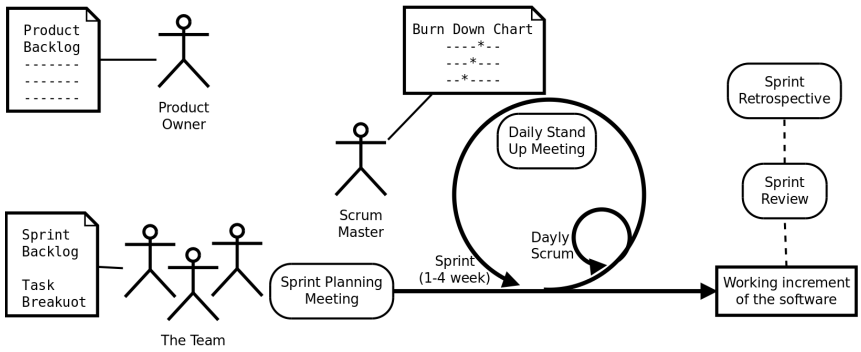


**Figure 16:** Scrum process (simplified view).

In the tradition of computer science, lies the handling of organizational and technological problems by providing a specialized framework, which provides both rules and tools. Generally, a framework means a base on which one can build something. This is an essential supporting structure or a basic structure underlying a system. Scrum [39] is a framework that originally was designed to support or underlay software development projects. However, it works for other complex and complicated undertakings as well. In accordance with Scrum ideology, product development takes place in small portions at a time. This style enables teams to respond quickly to changes in requirements and build exactly and only what is needed. Among other things, the team is able to respond quickly to their own failure in the development of the previous portion of the product. One can say that Scrum provides a work structure allowing teams to deal with the difficulty of IT projects. Scrum's popularity results from the simplicity of its main process (as shown in Figure 16) and from the transparency of its key roles – product owner, developers, and Scrum master. The product owner decides what has to be developed in the next 30-day sprint. The sprint period can be shortened, e.g. to 2 weeks, and the product owner's decision is limited to choosing the next task from the agreed backlogs. During the sprint developers create new functionality and demonstrate the product to the product owner, who has to choose the next task from the backlog. The Scrum Master oversees the process and helps developers to achieve high quality of product. The Scrum framework is consistent with the values of the Agile Manifesto therefore, it is sometimes called an Agile framework [40].

Unauthenticated
Download Date | 9/19/17 5:42 PM

Another framework is ITIL [41]. This is the broadly accepted approach to high quality IT service delivery and management. ITIL describes best practice for IT service management and most importantly, provides a common language with well-defined terms. IT services deliver values to the customer, i.e. the customer does not have to have own servers, software and staff, and can simply use ready information outcomes. ITIL provides its own service lifecycle with five stages: service strategy, service design, service transition, service operation, and continual service improvement. The lifecycle has a process organization (as a structured set of activities). The key roles in the ITIL framework are process owner, process manager, service owner, and service manager. As shown in Figure 17, ITIL closes the gap between business and technology. A business perspective is necessary to deal with the requirements of the business organization to develop and deliver appropriate IT services. ICT infrastructure management guarantees a stable infrastructure for communications and delivering IT services. Application management directly concerns the software development lifecycle. Development activities are required to create and test new IT services. Service management covers service delivery to the customer and service support, e.g. monitoring of customer access to the proper services and problem management). When planning to implement service management it is necessary to explain organizational benefits from the use of ITIL. Finally, security management protects and insures all the activities supported by the ITIL framework.

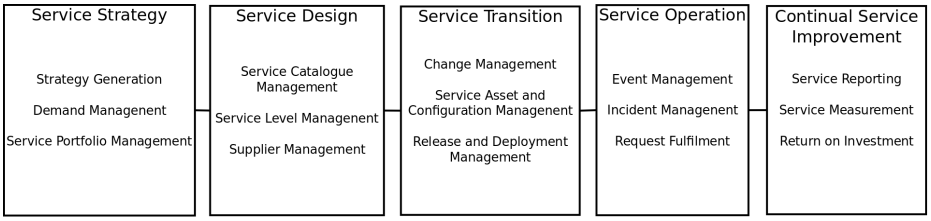


**Figure 17:** ITIL Framework (simplified view).

### 3.3.3 IT Team and IT Managers

A project's success depends largely on the team. Within an IT team, one will find a mixture of different professionals with different assignments. One can think about an IT team, as if it was comprised of sub-teams and at the same time, some of the sub-teams were provisional or transitional. In almost all IT projects, there are typical sub-teams with clear responsibilities like the leadership team, the solution architecture team, the application development team, the technology infrastructure team, and the information security team. In the real world, other configurations with less clear specialization and responsibility are possible, for example, the business team, the technical team and the data team. The degree of arrangement of obligations of the

Unauthenticated
Download Date | 9/19/17 5:42 PM

whole team depends on the policy of the company, and on traditions, which are supported by the owners and senior management.

As a rule, each team member plays a number of roles during the IT project life cycle; some of them may be performed simultaneously. The modern IT team does not necessarily have a hierarchical structure. The best circumstance is when all team members recognize that everyone else also provides significant value and contributions. Formally, inside an IT team there is always an organizational structure which sets task allocation, coordination and supervision. To visualize an organizational structure, an organization chart is shown in Figure 18.

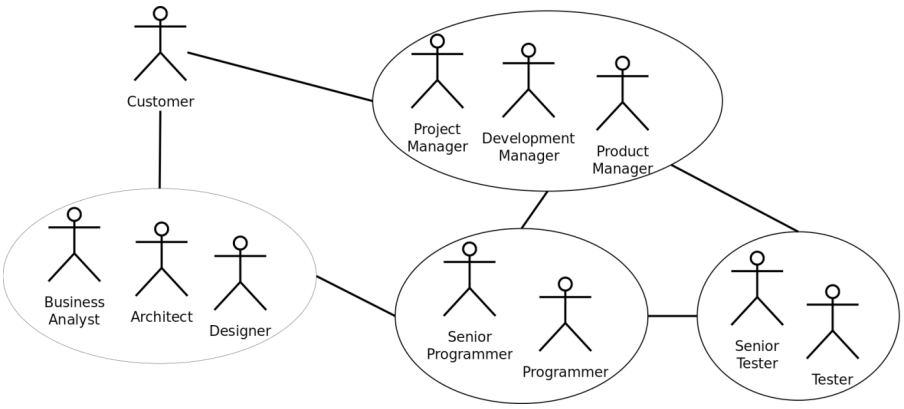


**Figure 18:** IT team organizational chart [42].

When the software team is the core of an IT project, it usually retains a structure appropriate to the software life cycle that is carried out. Other professionals make up the support groups. These may include business analysts, consultants, hardware specialists, security experts, sellers, schedulers and so on. However, this configuration is not mandatory in IT projects. Software development can be a secondary activity, e.g. in relation to constructing equipment or to business intelligence. It is important that regardless of the internal priorities, an IT project must lead to business success. To ensure this, an IT team is saturated with managerial staff, who aim to organize close cooperation of all substantive participants in the IT project.

IT managers are responsible for implementing and maintaining an organization's information technology infrastructure. At present, companies cannot function without information processing systems, which support well-organized business data management and communication. The IT manager has to keep an eye on the organization's operational necessities while continuously developing the organization's information system to aid the achievement of business goals. The

Unauthenticated
Download Date | 9/19/17 5:42 PM

specificity of the IT manager's profession lies in the fact that a managed system usually runs 24 hours a day and seven days a week. It is worth noting that not all developers (programmers, designers) are suited to the role of IT project manager even if they have a lot of experience. Not everybody has perfect communicational, organizational, team building and leadership skills. Not everyone is able to diagnose organizational problems in messy situations and to prescribe solutions to these problems in real time.

Unauthenticated
Download Date | 9/19/17 5:42 PM