Andrzej Yatsko, Walery Susłow

Insight into Theoretical and Applied Informatics

Introduction to Information Technologies and Computer Science

Andrzej Yatsko, Walery Susłow

Insight into Theoretical and Applied Informatics

Introduction to Information Technologies and Computer Science

Managing Editor: Irmina Grzegorek

Associate Editor: Dalia Baziuke

Language Editor: Adam Tod Leverton

DE GRUYTER
OPEN

Published by De Gruyter Open Ltd, Warsaw/Berlin
Part of Walter de Gruyter GmbH, Berlin/Munich/Boston

ISBN 978-3-11-046987-5
e- ISBN 978-3-11-046988-2

Bibliographic information published by the Deutsche Nationalbibliothek.
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.dnb.de.

Managing Editor: Irmina Grzegorek
Associate Editor: Dalia Baziuke
Language Editor: Adam Tod Leverton

www.degruyteropen.com

Cover illustration: © Andrzej Yatsko, Walery Susłow, Michał Statkiewicz

Contents

Abstract

The book is addressed to young people interested in computer technologies and computer science. The objective of this book is to provide the reader with all the necessary elements to get him or her started in the modern field of informatics and to allow him or her to become aware of the relationship between key areas of computer science. The book is addressed not only to future software developers, but also to all who are interested in computing in a widely understood sense. The authors also expect that some computer professionals will want to review this book to lift themselves above the daily grind and to embrace the excellence of the whole field of computer science. Unlike existing books, this one bypasses issues concerning the construction of computers and focuses only on information processing. Recognizing the importance of the human factor in information processing, the authors intend to present the theoretical foundations of computer science, software development rules, and some business aspects of informatics in non-technocratic, humanistic terms.

1 Introduction to Informatics

1.1 Basics of Informatics

Informatics is a very young scientific discipline and academic field. The interpretation of the term (in the sense used in modern European scientific literature) has not yet been established and generally accepted [1]. The homeland of most modern computers and computer technologies is located in the United States of America which is why American terminology in informatics is intertwined with that of Europe. The American term computer science is considered a synonym of the term informatics, but these two terms have a different history, somewhat different meanings, and are the roots of concept trees filled with different terminology. While a specialist in the field of computer science is called a computer engineer, a practitioner of informatics may be called an informatician.

The history of the term computer science begins in the year 1959, when Louis Fein advocated the creation of the first Graduate School of Computer Science which would be similar to Harvard Business School. In justifying the name of the school, he referred to management science, which like computer science has an applied and interdisciplinary nature, and has characteristics of an academic discipline. Despite its name (computer science), most of the scientific areas related to the computer do not include the study of computers themselves. As a result, several alternative names have been proposed in the English speaking world, e.g., some faculties of major universities prefer the term *computing science* to emphasize the difference between the terms. Peter Naur suggested the Scandinavian term *datalogy*, to reflect the fact that the scientific discipline operates and handles data, although not necessarily with the use of computers. Karl Steinbuch introduced in 1957 the German term *informatik*, Philippe Dreyfus introduced in 1962 the French term *informatique*. The English term *informatics* was coined as a combination of two words: *information* and *automation*; originally, it described the science of automatic processing of information. The central notion of informatics was the *transformation of information*, which takes place through computation and communication by organisms and artifacts. Transformations of information enable its use for decision-making.

Q&A What is informatics?	**Technical definition:** Informatics involves the practice of information systems engineering, and of information processing. It studies the structure, behavior, and interactions of natural and artificial systems that collect, generate, store, process, transmit and present information. Informatics combines aspects of software engineering, human-computer interaction, and the study of organizations and information technology; one can say it studies computers and people. In Europe, the same term informatics is often used for computer science (which studies computers and computer technologies). **Business definition:** Informatics is a discipline that combines into one-field information technologies, computer science and business administration.

A number of experts in the field of computer science have argued that in computer science there are three distinct paradigms. According to Peter Wegner, these three paradigms are science, technology and mathematics. According to Peter J. Denning, these are theory, modeling and design. Amnon H. Eden described these as rationalist, technocratic, and scientific paradigms. In his opinion, in frames of rationalist paradigm the computer science is a branch of mathematics; mathematics dominates in theoretical computer science and mainly uses the logical conclusion. The technocratic paradigm is the most important in software engineering. In the frame of a scientific paradigm, computer science is a branch of empirical science, but differs in that it carries out experiments on artificial objects – software and hardware.

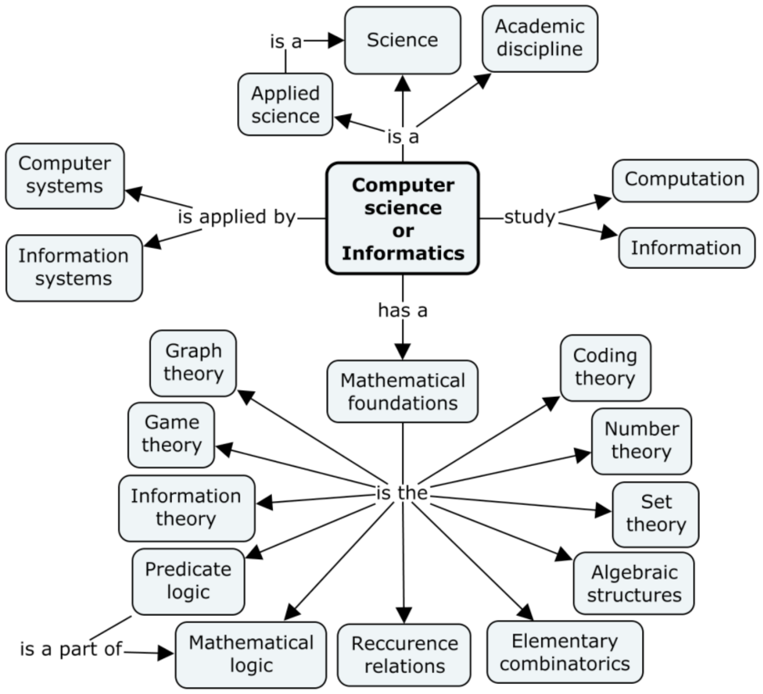

Figure 1: Foundations of computer science and informatics.

An overview of computer science can be borrowed from Charles Chen [2]. He referred to mathematical, theoretical, and practical (application) branches as key components. The mathematical branch is devoted to systems modeling and creating applications for solving math problems. Some related principles of study are classical and applied mathematics, linear algebra and number theory. The theoretical branch covers algorithms, languages, compilers and data structures. This branch is based on numerical and logical analyses, discrete mathematics, statistics, programming theory, software and hardware architecture. The practical branch contains operating systems, different apps, frameworks, and software engineering. Real world applications of computer science include cryptography, software development, project management, IT system support, artificial intelligence, and computer graphic.

The dominated meaning of the term *informatics* has changed in different periods of development of the science. At first, informatics was treated as a part of library science – the theory of scientific information activity. This means practical work on the collection, analytical and synthetic processing, storage, retrieval and provision of scientific information laid down by scientists and engineers in their documents. Informatics was a discipline that studied the "structure and general properties of scientific information, as well as the laws of its creation, transformation, transmission and use in various spheres of human activity". Later, it turned more in the direction of the science of computers and their applications and informatics came to mean a computing and programming discipline engaged in the development of the theory of programming and of application of computers [3]. The last meaning of the term as fundamental science of information processes in nature, society and technical systems was coined in the early 1990s. According to this interpretation informatics examines the properties, laws, methods and means of formation, transformation and dissemination of information in nature and society, including by means of technical systems. The latter change caused that besides informatics specializations like theoretical computer science or technical informatics there are also social informatics, biological informatics, and physical informatics. Nowadays we can observe the coexistence of all three definitions of the word "informatics". This fact complicates the study of this scientific direction.

Informatics has evolved in eight major areas:

1. Theoretical informatics
2. Cybernetics
3. Programming
4. Artificial Intelligence
5. Information systems
6. Computing equipment
7. Social informatics
8. Bioinformatics

1.1.1 Historical Background

The history of informatics [4], in the modern sense, begins in the middle of the XX century, due to the advent of computers and the beginning of the computer revolution. Informatics or computer science is only now entering the age of early maturity. Some historians date the beginning of the informatics era from the date of the creation of the first electronic computer. Others prefer to date it from the first electric (and even mechanical) calculators. As it always happens in such situations, there are many opinions and justifications for each point of view. If we take 1950s as the beginning of the history of modern informatics, all previous events (important in the context of information processing science) are considered prehistoric. There were some very important inventions and discoveries during prehistory, which allow us to trace the historical logic of the creation of modern information technologies.

To understand the roots of informatics, one should look at the history of computer technology [5], which includes a multitude of diverse devices and architectures. The abacus from ancient Babylon (300 BC) and China (500 BC) are the oldest known historical examples. The Jacquard loom invented by Joseph Marie Jacquard (1805) and the analytical engine invented by Charles Babbage (1834) are the first examples of zero generation (prehistoric) computers – mechanical machines designed to automate complex calculations. De facto, Babbage's engine was also the first multi-purpose programmable computing device. After him, George and Edvard Scheutz (1853) constructed a smaller machine that could process 15-digit numbers and calculate fourth-order differences. Ada Lovelace (1815-52) collaborated with Charles Babbage. She is said to be the first programmer. She saw the potential of the computer over a century before it was created. It took more than one century, until the end of the 1960s, when mechanical devices (e.g. the Marchant calculator) found widespread application in engineering and science.

First generation electronic computers (1937-1953) used electronic components based on vacuum tubes. This was the period when both digital and analog electric computers were developed in parallel. The first electronic digital computer was ABC, invented by John Vincent Atanasoff at Iowa State University. The second early electronic machine was Colossus, designed by Alan Turing (1943) for the British military. The first general purpose programmable electronic computer was ENIAC (1943-1945), built by J. Presper Eckert and John V. Mauchly at the University of Pennsylvania. In fact, as late as the 1960s, analog computers still were used to solve systems of finite difference equations. Nevertheless, digital computing devices won the competition, because they proved to be more useful when dealing with large-scale computations (more computing power, more scalable and economical). Software technology during this period practically did not exist; programs were written out in machine code. Only in the 1950s a symbolic notation known as assembly language, began to be used.

Second generation computers (1954-1962) were based on semiconductor elements (discrete diodes and transistors) with a very short switching time and randomly accessed magnetic memory elements. There was a new opportunity to perform calculations in the format of real numbers (floating point). High-level programming languages such as FORTRAN, ALGOL and COBOL were created during this time. These changes enabled the production of the first computers designed not only for science and the military, but also for commerce. The first two supercomputers (LARC and IBM

7030) were designed during the same period. These were machines that had much more calculating power than others and could perform parallel processing.

The key changes in the third generation of computers (1963-1972) concerned the use of integrated electronic circuits instead of discrete elements and the use of semiconductor memory instead of magnetic cores. Changes in computer architecture were associated with the spread of operating systems and parallel processing techniques. These developments led to a revolutionary increase in the speed of execution of calculations. The key figure for the development of computer technology in this period was Seymour Roger Cray who established the supercomputer industry through his new architecture approaches. His computers attained for the first time a computation rate of 10 million floating-point operations per second (10 Mflops). At the same time (1963) Cambridge and the University of London developed in cooperation Combined Programming Language (CPL), which became the prototype for a number of other major programming languages. Early implementations of the excellent operating system – UNIX were based on the language B, which is derived from the CPL.

The next (fourth) generation of computer systems (1972-1984) went through the use of large and very large scale integration of electronic components on chips. This has enabled an entire processor and even an entire simple computer to fit onto a single chip. That very important change allowed for a reduction in the time required to execute basic calculations. Most systems acquired a semiconductor main memory, which also increased the speed of information processing. There were two very important events at the beginning of the fourth generation: the development at Bell Labs of the C programming language and the UNIX operating system (written on C). In a short time, UNIX was ported to all significant computers; thousands of users were exempted from the obligation to learn a new operating system each time they changed computer hardware. The new trend in software methodology during the fourth generation was the introduction of a declarative style of programming (Prolog) next to an imperative style (C, FORTRAN).

The fifth generation of computer technology (1984-1990) was characterized mainly by the widespread adoption of parallel processing. This means that a group of processors can be working on different parts of the same program. At that time, fast semiconductor memories and vector processors became standard on all computers, this was made possible by rapid growth of the scale of integration; in 1990, it was possible to build integrated circuits with a million components in one chip. Thus, all the processors participating in parallel calculations may be located on the same PC board. Another new trend was the common use of computer networks; connected to the network, single-user workstations created the conditions for individual work at the computer in contrast to the previous style of work in a group.

The latest, i.e. the sixth generation of computers (1990-) is distinguished by the exponential growth of wide-area networking. Nowadays, there is a new category of computers (netbooks), which are useful only when using the Internet. Another category, mobile computing devices (smartphones, palmtops, tablet PCs) are strongly dependent on the network. The computer spread out from science, technology and industry to all areas of our lives.

1.1.2 Famous People in the History of Computing

Many people shaped Informatics into what it is now [6]. We all know those who have had an immense influence on the industry, like *Bill Gates* or *Steve Jobs*. However, let us take a look at some lesser known individuals.

Grace Murray Hopper is the author of term *debugging* (1947), which means detecting and removing errors from a computer program. He is one of the originators of the idea of machine-independent programming that led to the development of modern programming languages starting with COBOL.

Ken Thompson and *Dennis Ritchie*, two engineers at AT&T’s Bell Labs research invented OS UNIX (1969). From the beginning, it was a multitasking, multiuser computer operating system; all Apple’s personal computers still owe their success to this operating system.

Seymour Cray, who has already been mentioned, set the standard for modern supercomputing. Two important aspects to his design philosophy were removing heat, and ensuring the electrical length of every signal path on board. That is why all his computers were equipped with efficient built-in cooling systems, and all signals were precisely synchronized.

Marvin Lee Minsky, co-founder and supervisor of Artificial Intelligence Laboratory (AI Lab) at Massachusetts Institute of Technology (MIT) is the author of foundational works on the analysis of artificial neural networks. He has made many contributions to not only AI, but also to cognitive psychology, computational linguistics, robotics, and telepresence (the use of virtual reality technology for remote control and for apparent participation in distant events).

Douglas Carl Engelbart is generally well known for inventing the computer mouse (1960s). More specifically, he developed some of the key technologies used in computing today. First, he advanced the study of human-computer interaction that resulted in developing desktop user interface and other components of graphical user interfaces (GUI). His efforts led to the development of hypertext and networked computers too.

The originator of the term *artificial intelligence,* computer scientist *John McCarthy* developed the LISP (1958) – the second high-level programming language in the history of computing; to this day, programmers use Lisp dialects. In the 1960s, McCarthy intensively popularized sharing a computing resource among many users; which means the use of multi-tasking (concurrent processes) and multiprogramming approach to building software systems.

Software designer *Tim Paterson* is the original author (1980) of MS-DOS; de facto, Bill Gates only rebranded his operation system known at first as QDOS (Quick and Dirty Operating System). Paterson worked for Microsoft on MSX-DOS and on Visual Basic projects.

Daniel Singer and *Robert M. Frankston* are the creators of the first spreadsheet computer program *VisiCalc*. It was historically the first program that has transformed microcomputers from a plaything into the professional tools needed to perform specialized calculations.

Robert Elliot Kahn and *Vinton Gray Cerf* are the inventors of two basic communication languages or protocols of the Internet: TCP (Transmission Control Protocol) and IP (Internet Protocol). Both, TCP and IP are currently in widespread use on commercial and private networks.

Niklaus Wirth is known as the chief designer of several programming languages, including Pascal. He has written very influential books on the theory and practice of structured programming and software engineering.

James Gosling invented (1995) the Java programming language, which become popular during the last decade. Compiled Java applications typically run on Java virtual machines regardless of computer architecture. Independence from the computer architecture has been achieved in such a way.

The inventor of the World Wide Web (WWW, an internet-based hypermedia initiative for global information sharing), *Timothy John Berners-Lee* implemented in 1989 the first successful client-server communication via the Internet using Hypertext Transfer Protocol (HTTP). Nowadays he is the director of the W3C – the main international standards organization for the WWW.

Linus Benedict Torvalds created Linux – the kernel of the operating system GNU/Linux, which is currently the most common of free operating systems. Currently, only about two percent of the current system kernel has been written by Torvalds, but he remains the decision maker on amending the official Linux kernel tree.
It is not possible to mention here all who have contributed to the development of modern computer technology. There have been, and will continue to be, many others.

1.1.3 Areas of Computer Science

Computer science combines a scientific and practical approach. On the one hand, computer scientists specialize in the design of computational and information systems, on the other hand they deal with the theory of computation and information. To understand the areas of interest of computer scientists, we must analyze from where they or their teachers came to computer science: from electronics, from mathematics, from physics and so on.

In 2008 ACM Special Interest Group on Algorithms and Computation Theory published its vision of theoretical computer science themes that have the potential for a major impact in the future of computing and which require long-term, fundamental research. These are the issues of algorithms, data structures, combinatorics, randomness, complexity, coding, logic, cryptography, distributed computing, networks among others. Of course, some central research areas were not represented here at all, but vision can help us to understand the field of computer science theory.

Let us browse theoretical and applied areas, which are relatively stable and are represented in computer science curricula. Generally, theoretical sciences form a subset of wide-ranging computer science and mathematics. This is why all of them focus on the mathematical aspects of computing and on the construction of mathematical models of different aspects of computation. Someone who loves mathematics, can find among these disciplines his/her preferred area of interests. Applied sciences are directly related to the practical use of computers. The following lists contain names of the most popular theoretical and applied disciplines belonging to area of computer science:

List 1. Theoretical computer science	**List 2. Applied computer science**
Algorithms and data structures	Artificial intelligence
Algorithmic number theory	Computer architecture and engineering
Computer algebra or Symbolic computation	Computer Performance Analysis
Formal methods	Computer graphics and visualization
Information and coding theory	Computer security and cryptography
Programming language theory	Computational science
Program semantics	Computer networks
The theory of computation	Concurrent, parallel and distributed systems
Automata theory	Databases
Computability theory	Health informatics
Computational complexity theory	Information science
Formal language theory	Software engineering

1.1.4 Theoretical and Applied Informatics

Before the end of 1970s, cybernetics was the dominant term among sciences related to the processing of information; respectively, theoretical informatics was named *mathematical (theoretical) cybernetics*. Briefly, theoretical informatics is a mathematical discipline, which uses methods of mathematics to construct, and study models of processing, transmission and use of information. Theoretical informatics creates the basis on which whole edifice of informatics is build. By its very nature, information tends to discrete representation. Data messages typically can be described as a discrete set. This is why theoretical informatics is close to discrete mathematics, and many models of theoretical informatics are borrowed from discrete mathematics. However, as a rule, these models are filled with content related to the particulars of information as an object of study.

1.1.4.1 Theoretical Informatics

Theoretical computer science itself is divided into a number of distinct disciplines. According to tasks clusterization, theoretical informatics can be divided into five areas:

1. Information disciplines based on mathematical logic. These develop methods to use the achievements of logic to analyze the processes of information processing using computers (e.g., *algorithms* or the *theory of parallel computing*). As well, they deal with methods by which it is possible to study the processes taking place in the computer during computation (e.g., *automata theory* or the *Petri net theory*).
2. *Theory of computation*. Previously, mathematicians do not care about the possibility of transferring their methods of solving problems in a form that allows the programming. The expansion of computers stimulated the development of special problem-solving techniques in mathematics, so disciplines lying at the boundary between discrete mathematics and theoretical computer science (e.g., *computational mathematics and geometry*) emerged.
3. *Theory of Information*. This means the study of information per se (i.e. in the form of an abstract object, devoid of specific content). Theory of information engages the general properties of information laws that govern its birth, development and destruction. This science is closely related to *coding theory*, whose mission is to study the forms in which the content of any particular information item (e.g. the message being sent) can be “cast”. In information theory, there is a section specifically dealing with theoretical issues, which concern transmission of information through various channels of communication.
4. *System Analysis*. Informatics has to deal with real and abstract objects. The information circulating in the real form is materialized in various physical processes, but from a scientific perspective, it acts as an abstraction. This shift causes the need to use on computers the special abstract (formal) model of the physical environment in which the information is perceived in the real world. The shift from real objects to models that can be implement in computers and used to study some problems, requires the development of special techniques. The study of these techniques is carried out by system analysis, which studies the structure of real objects and provides methods for their formalized description (virtualizations). *General systems theory* is a part of system analysis, which studies the nature of a variety of systems with the same approach. Systems analysis takes

a boundary position between theoretical informatics and cybernetics. The same boundary position is occupied by two more disciplines. *Simulation* is one of them; this science develops and uses special techniques for the reproduction of real processes on computers. The second science is *queuing theory,* which is the mathematical study of a special, but very broad class of models of information transmission and information processing, the so-called queuing system. Generally, models of queuing systems are constructed to predict queue lengths and waiting times.

5. *The Theory of Games and Special Behavior*. The last class of disciplines included in theoretical informatics focuses on the use of information for decision-making in a variety of situations encountered in the world. It primarily includes *decision theory*, which studies the general scheme used by people when choosing their solutions from a variety of alternatives. Such a choice is often the case in situations of conflict or confrontation; that is why models of this type are studied by *game theory*. A decision maker always wants to choose the best of all possible solutions. Problems which arise in a choice situation are studied in the discipline known as *mathematical optimization* also known as *mathematical programming*. To achieve goals, the decision making process must obey a single plan; the study of ways of building and using such plans is provided by another scientific discipline – *operations research*. If not individual, but team decision are made, there are many specific situations, e.g., the emergence of parties, coalitions, agreements and compromises. The problem of collective decision-making is examined in the *theory of collective behavior*.

1.1.4.2 Applied Informatics

Cybernetics, which has already been mentioned, can be seen as the first applied informatics discipline concerned with the development and use of automatic control systems of diverse complexity. Cybernetics originated in the late 40s, when Norbert Wiener first put forward the idea that the control system in living and non-living artificial systems share many similarities. The discovery of this analogy promised the foundation of a general theory of control, the models of which could be used in newly-designed systems. This hypothesis has not withstood the test of time, but principles concerning information management systems have greatly benefited. *Technical cybernetics* was the most fully developed. It includes *automatic control theory*, which became the theoretical foundation of automation.

As the second applied informatics discipline, *programming* relies completely on computers for its appearance. In the initial period of its development, programming lacked a strong theoretical base and resembled the work of the best craftsmen. With experience, programming has groped towards general ideas that underlie the construction of computer programs and programming arrangements themselves. This has resulted in the gradual establishment of theoretical programming, which now consists of multiple destinations. One of them is connected with the creation of a variety of programming languages designed to make human interaction with computers and information systems easy.

Artificial intelligence is the youngest discipline of applied informatics, but now it determines the strategic directions of the development of information sciences. Artificial intelligence is closely related to theoretical computer science, from which it has borrowed many models and methods, such as the active use of logical tools to convert knowledge. Equally strong is its relation to cybernetics. The main objective of work in the field of artificial intelligence is the desire to penetrate the secrets of creative activity of people, their ability to master skills, knowledge and abilities. To do this, you need to open those fundamental mechanisms by which a person is able to learn almost any kind of activity. Such a goal of researchers in the field of artificial intelligence closely associates them with the achievements of psychology. In psychology, there is now a new area actively developing – *cognitive psychology*, which focuses exactly on examining the laws and mechanisms that interest specialists in the field of artificial intelligence. The sphere of interests of experts in the field of artificial intelligence also includes linguistic studies. Mathematical and applied *linguistics* also work closely with research in the field of artificial systems designed for natural language communication. Artificial intelligence is not a purely theoretical science; it touches on the applied issues related to the construction of real existing intelligent systems, such as robots.

The significant practical importance of informatics manifests itself in the field of *information systems*. This trend was created by researchers in the field of *documentology* (the scientific study of documents, including the examination of their structure) and by the analysis of scientific and technical information that were conducted even before computers. However, true success of information systems was reached only when computers become part of their composition. Now within this area, a few basic problems are being solved. The analysis and forecasting of various information streams, the study of methods of information presentation and storage, the construction of procedures to automate the process of extracting information from documents, and the creation of information search systems. On the one hand, research in the field of information systems is based on applied linguistics, which creates languages for the operative saving of information and for quickly finding answers to incoming requests in data warehouses. On the other hand, the theory of information supplies this research by models and methods that are used to organize the circulation of information in data channels.

Computer engineering is a completely independent line of applied research that integrates several fields of electrical engineering and computer science required to develop computer hardware. Within this field many problems which are not directly related to informatics are solved. For example, numerous studies are conducted to improve the element base of computers. The progress of modern informatics is unthinkable without the evolution of computers – the main and the only tool for working with various information. Research in the area of artificial intelligence has had great influence on the development of new computers. As a result, new generations of computers are far more intelligent than their ancestors. There are two major specialties in computer engineering – computer software engineers develop software, computer hardware engineers develop computer equipment. In the same way, computer engineers specialize in narrow sectors, e.g. coding and information protection, operating systems, computer networks and distributed systems, architecture of computer systems, embedded systems and so on.

The world is now entering the epoch of the information society. Certainly, the distribution, storage and processing of information will play a huge role in this society. Information becomes a commodity of great value, and information technology is an increasingly influential factor in business and in everyday life. The preparations for the transition of an information society are causing many problems, not only of a technical, but also of a social and legal nature. All of these problems are a subject of study for psychologists, sociologists, philosophers and lawyers who work in the field of informatics. Already, automated training systems, workstations for various specialists, distributed banking systems and many other information systems, whose operations are based on the full gamut of informatics have been created. The information technology sector, which contains these activities, can be defined as a *social informatics*.

There is one other area, in which informatics has recently played the role of an important accelerator of the research processes, i.e. natural sciences like biology, chemistry or physics. The main objective of this trend is the study of information processes in natural systems, and the use of acquired knowledge for the organization and management of natural systems and for the creation of new technical systems. *Informatics in the natural sciences* has its own characteristics, depending on the natural discipline.

1.2 Relationship with Some Fundamental Sciences

In the quite short historical period of its development, informatics has become a foundation for and taken inspiration from other fundamental sciences. At the same time, informatics has giving advances and inspirations to other disciplines. Sometimes, informatics has even merged with the name of a discipline, as in bioinformatics, health informatics or medical informatics; it denotes the specialization of informatics to the data management and information processing in the named discipline. The amalgamation of informatic theories and methods with classical (fundamental) disciplines enriches them too. Let us see how these interactions work.

1.2.1 Informatics and Mathematics

The mathematics of current computer science is constructed entirely on *discrete mathematics*, especially on *combinatorics* and on *graph theory*. Discrete mathematics (in contrast with continuous mathematics) is a collective name for all branches of mathematics, which deal with the study of discrete structures, such as graphs or statements in logic. Discrete structures contain no more than countable sets of elements. In other words, discrete mathematics deals with objects that can assume only separated values. Besides combinatorics and graph theory, these branches include cryptography, game theory, linear programming, mathematical logic, matroid theory, and number theory, which are used by informaticians intensively. The advantage is that nontrivial real world problems can be quickly explored by using methods of these discrete disciplines. One can even say that discrete mathematics constitutes the mathematical language of informatics.

From the beginning, informatics has been very solidly based in mathematics (Figure 1). In addition, theoretical informatics could be considered a branch of mathematics. It is easy to notice that both share a conceptual apparatus. The more precisely circumscribed area of computer science definitely includes many things, which would not be considered mathematics, such as programming languages or computer architecture. The computerization of sciences, including mathematics, also stimulated those sciences as well. Questions from informatics inspired a great deal of interest in some mathematical branches, e.g. in discrete math, especially in combinatorics. Some mathematical challenges arise from problems in informatics (e.g. complexity theory); they demand the use of innovative techniques in other branches of math (e.g. topology). Fundamental problems of theoretical computer science, like the P versus NP problem, have obtained an appropriate importance as central problems of mathematics.

The two-way conversation between informaticians and mathematicians is profitable. Mathematicians consider computational aspects of their areas to facilitate construction of the appropriate virtual objects (or entities, in terms of software). Numerous techniques increase mathematical sophistication, which result in efficiently solving the most important computational problems. Informatics has activated many mathematical research areas like computational number theory, computational algebra and computational group theory. Furthermore, a diversity of computational models was drafted to explain and sometimes anticipate existing computer systems, or to develop new ones like online algorithms and competitive analysis, or parallel and distributed programming models. Here are some explanations for the exemplary models:

– An online algorithm has a special restriction; it does not receive its input data from the beginning as a whole, but in batches (rounds). After each round, the algorithm has to provide a partial answer. An example is an allocation of CPU time or memory (scheduling), because in general it is not known which processes will require resources, it is necessary to allocate resources only based on the current situation. Competitive analysis compares the performance of such algorithms to the performance of an optimal offline algorithm that can view the sequence of requests in advance.

– Distributed computation is a solution for the big data problem. Unfortunately, this is very difficult to program due to the many processes which are involved, like sending data to nodes, coordinating among nodes, recovering from node failure, optimizing for locality, debugging and so on. The MapReduce programming model is suggested which allows such data sets to be processed with a parallel, distributed algorithm on a computer cluster. The model was inspired by functional programming. Applications which have implement the MapReduce framework, achieve high scalability and fault-tolerance, which is obtained by optimizing the execution engine.

1.2.2 Informatics and Mathematical Logic

Mathematical logic or *symbolic logic* is a branch of mathematics that studies mathematical notation, formal systems, verifiable mathematical judgments, computability, and the nature of mathematical proof in general. More broadly, mathematical logic is regarded as a mathematized branch of formal logic or logic, developed with the help of mathematical methods. Topically, mathematical logic stands in close connection with meta-mathematics, the foundations of mathematics, and theoretical computer science. Logic in computer science is the direction of research, according to which logic is applied in computation technologies and artificial intelligence. Logic is very effective in these areas. However, one should not forget that some important research in logic were caused by the development of computer science, for example, applicative programming, computation theory and computational modeling.

From the very beginning, informatics depends heavily on logic; do not forget that Boolean logic and algebra was used for the development of computer hardware. Logic is a part of information technology, for example, in relational data models, relational databases, relational algebra, and relational calculus. In addition, logic has provided fundamental concepts and ideas for informatics, which naturally can use formal logic. For example, this applies to the semantics of programming languages. Here are some very important applications of logic in the field of informatics:

– Formal methods and logic reasoning about concepts in semantic networks and semantic web;
– Problem solving and structured programming for application development and the creation of complex software systems;
– Probative programming – the technology of development of algorithms and programs with proof of the correctness of algorithms;

– The logic of knowledge and justified assumptions, e.g. in artificial intelligence;
– Prolog language and logic programming for creating of knowledge bases and expert systems and research in the field of artificial intelligence;
– Spatial and temporal logic to describe spatial position and movement;
– Abstract machine logic, especially code compilation and optimization;
– Objects transformation based on lambda calculus.

1.2.3 Informatics and Cybernetics

Cybernetics investigates abstract principles of organization of complex systems [7] (not only artificial, but also natural systems). It deals with the functioning of the systems, rather than their design. The fundamental contribution of cybernetics was its explanation of purposiveness of natural systems using key concepts of *information* and *control*. Cybernetics focuses on the use of information, models and the control actions by systems to achieve their goals and to prevent disorder at the same time. The historical relationship between computer science and cybernetics has already been mentioned in section 1.1. In the XX century, cybernetics was presented as one of the main areas of computer technology; theoretical informatics was named mathematical cybernetics; cybernetics was named the first applied informatics discipline.

Two disciplines born from cybernetics, computer science (informatics) and control engineering had become fully independent in the early 1970s. Informatics has taken over the terminology (conceptual base) of cybernetics in the areas of equipment control and information analysis. Some areas of modern informatics have their origins directly in cybernetics. The best-known areas are information theory, robotics, the study of systems, and the theory of computing machines. After this, the remaining cyberneticist have differentiated themselves from mechanistic approaches by emphasizing the phenomena of autonomy, self-organization, cognition, and the role of the observer in modeling a system. This movement became known as second-order cybernetics. Nowadays, cybernetics is not such a popular scientific discipline; modern theories have divided its topics.

1.2.4 Informatics and Electronics

Transistors are the smallest active electrical components of modern computers. They operate as elements of integrated circuits. For example, microprocessors are physically integrated circuits, which are produced for computers in the form of chips. Consisting of transistors, chips can perform arithmetical and logical operations, store data and communicate with other chips. Electronic computer circuits operate in digital mode; this means that all the transistors of the integrated circuits of computers can be only in specific states. Popular digital electronic circuits operate in binary mode, so these specific states are only two; they are referred to as *zero* and *one*. The theory of design and operation of the various electronic components like transistors is carried out by *electronics*, which is a part of *electrical engineering* and a branch of physics. To emphasize the type of basic materials used in modern electronics components and circuits, the terms *semiconductor electronics* or *solid state electronics* are used. To underline the digital operating mode of electronic circuits intended for the construction of computers, the term *digital electronics* is used.

Computer engineering fuses electronics with computer science to develop faster, smaller, cheaper, and smarter computing systems. Computer engineers design and implement hardware (computer architectures); design and implement systems software (operating systems and utility software); design processors; design and implement computer-computer and human-computer interface systems. One can say that computer engineers are engaged in analyzing and solving computer-oriented problems. Computer engineering also deals with the embedded computers that are parts of other machines and systems, as well as with computer networks, which are developed for data transfer.

It is an interesting fact that progress in micro-miniaturization of integrated electronic circuits has been and remains heavily supported by informatics. There have long been popular industrial software tools for designing electronic systems such as printed circuit boards and integrated circuits called ECAD (electronic design automation).

1.2.5 Informatics and Linguistics

Computational linguistics joins linguistics and informatics; it is one of the cognitive sciences, which partly overlaps with the field of artificial intelligence and concerns understanding natural language from a computational perspective. Theoretical computational linguistics deals with formal theories about language generation and understanding; the high complexity of these theories forces the computer management of them. One of the main goals of theoretical study [8] is the formulation of grammatical and semantic frameworks for characterizing languages enabling a computational approach to syntactic and semantic analysis. In addition, computational linguists discover processing techniques and learning principles that exploit both the structural and distributional properties of language. This kind of research is not possible without the answer to the question: how does language learning and processing work in the brain?

Applied computational linguistics, called *language engineering*, focuses on the methods, techniques, tools and applications in this area. Historically, computational linguistics was initiated to solve the practical problem of automatic written text translation. Machine translation was recognized as being far more difficult than had originally been assumed. In recent years, we can observe the growth of technology for analysis and synthesis of spoken language (i.e., speech understanding and speech generation). Besides designing interfaces to operate in a natural language, modern computational linguistics has considerable achievements in the field of document processing, information retrieval, and grammar and style checking.

Computational linguistics partially overlaps with *natural language processing*, which covers application methods of language description and language processing for computer systems generally. This entails:

- Creation of electronic text corpora;
- Creation of electronic dictionaries, thesauri, ontologies;
- Automatic translation of texts;
- Automatic extraction of facts from texts;
- Automatic text summarization;
- Creation of natural language question-answering systems;
- Information retrieval.

Computational linguistics also embraces topics of computer-assisted second language learning. It is not only about the use of computers as an aid in the preparation, presentation, and assessment of material to be learned. The modernization of educational methods, such as the utilization of multimedia, and web-based computer assisted learning is also sought.

1.2.6 Informatics vis-à-vis Psychology and Sociology

There are more reasons for social, behavioral, or cognitive scientists (psychologists and sociologists) to acquire a basic familiarity with informatics tools and techniques, which give them a new ability to provide, publish and evaluate their research. One can speak about a new phenomenon – computational social science [9]. Here is an incomplete list of what a new computational approach can bring to psychology and sociology:

- Web-based data collection methods;
- Mobile data collection method;
- Data manipulation and text mining;
- Computerized exploratory data analysis and visualization;
- Big Data and machine learning applications.

A new, widely unknown discipline *psychoinformatics* [10] already helps psychologists to mine data and to develop patterns based on relations among data; the pattern finally reflects specific psychological traits. Furthermore, the development of psychology and informatics is on such a stage, when psychology can try to model the information processes of human thought. Some cognitive scientists even define metaphorically psychology as the informatics of the human mind [11]. In their view, the self-organization structures in the brain leads to the arrangement of stable data objects (images), most of which have prototypes in the external world or in the human body. The aggregate of objects forms a working model – a personal image of the world. Supposedly, primary information objects are formed using the innate transformation of sensory signals; the same transformation can be applied recursively to the primal images, creating images in the next row that do not have any direct prototypes in the outside world, nor in the human body.

At the same time, informatics, especially artificial intelligence, can be used with a positive effect as psychological models of human behavior. For example, some mental problem solving techniques studied in psychology are used as prototypes in artificial intelligence. Artificial intelligence is implemented as a computer or other device that has been designed “to think like a human”. Software engineers also use psychological knowledge, especially to design human-computer interactions; there is a separate presentation layer of software, which is designed by specialists in graphical user interfaces. Sociological knowledge is very useful in the context of managing large IT project teams. Separately, we can note the existence of *social informatics* – an interdisciplinary research field that studies the information society.

1.2.7 Informatics and Economics

Intensive implementation of information technology in the economy has led to a new area in computer science called *business informatics*. Business informatics (from the German *Wirtschaftsinformatik* or organizational informatics) is an integrated applied discipline, based on the interdisciplinary links between informatics, economics, and mathematics. Economic information is the aggregate data reflecting the status and progress of business processes. Economic information circulates in the economic system and the accompanying processes of production, distribution, exchange, and consumption of goods and services. It should be considered as one of the types of management information. Business informatics deals with knowledge about information systems, which are used for preparation and decision-making in management, economics and business. This discipline includes information technology and significant elements oriented on construction and implementation of information systems. In contrast to information systems theory, business informatics focuses on solutions, not on empirically explaining phenomena of the real world.

Economics is a stable source of tasks for computer engineers; e.g., the stock market delivers huge data sets for computer analysis and prediction of events. At the same time, economics inspires theoretical computer science, because business needs more and more new algorithms, feeling the heat of competition. Historically, economic initiated one of the first grand accomplishments in the algorithm theory – the simplex method, which has proved to be very effective in decision-making problems.

The impact of the economy on IT can be seen in the formation of an agent-based programming paradigm. The idea for this paradigm came from a commercial brokerage model. A software agent is a kind of small independent program, which mediates the exchange of information between other programs or people. Agents must be prepared to receive incorrect data from another agent, or to receive no data at all; this means that agent-based systems should be prepared to function in conditions of uncertainty. Agent-based modeling promises new analytical possibilities not only for business, but also for social sciences. At present, computing scientists expect help from economists to solve the problems of optimization of software agents.

Completely new types of algorithmic problems from natural sciences are challenging theoretical informatics: namely, problems in which the required output is not well defined in advance. Typical data might be a picture, a sonogram, readings from the Hubble Space Telescope, stock-market share values, DNA sequences, neuron recordings of animals reacting to stimuli, or any other sample of "natural phenomena". The algorithm (like the scientist), is "trying to make sense" of the data, "explain it", "predict future values of it", etc. The models, problems and algorithms here fall into the research area of computational learning and its extensions.

1.3 Information Theory

The term information entered into scientific use long before the rapid development of electronic communications and computing. Information (from the Latin word *informatio*, which means clarification, presentation, interpretation) in the most common sense means details about something transmitted by people directly or indirectly. Initially, this concept was only associated with communicative activities in the community. The understanding of information to be messages sent by people (oral, written or otherwise, like conventional signals, technical facilities, etc.), remained until the mid 20-ies of XX century.

Gradually, the concept of information has gained a more and more universal meaning. Before the beginning of the 1920s, the information was treated on a qualitative level, any formal concepts, procedures and methods of quantifying were not used. The main focus was on the mechanisms of influence on receivers of information, to the ways of insurance its accuracy, completeness, adequacy, etc. The subsequent refinement of information's scientific meaning was done by scientists in different directions. At first, they tried to include it as part of the structures of other general concepts (e.g. probability, entropy, and diversity). R. S. Ingarden (1963) proved the failure of these attempts.

The development of the technical facilities of mass communication (the telephone, telegraph, radio, television, computer networks) led to a snowballing in the number of transmitted messages. This led to the need to evaluate various characteristics of information, and in particular its volume. With the development of cybernetics, information become one of the main categories of its conceptual apparatus, along with concepts such as management and communication.

Nowadays, the field of information theory partly belongs to mathematics, computer science, physics, electronics and electrical engineering. Important subfields of information theory are measures of information, source and channel coding, algorithmic complexity theory, information-theoretic security, and others. Information theory investigates the processes of storage, conversion and transmission of information. All this processes are based on a certain way of measuring the amount of information. A key measure of information is entropy, which is expressed by the average number of bits needed to store or communicate one symbol in a message. Entropy quantifies the uncertainty involved in predicting the value of a signal.

As it was first applied, mathematical information theory was developed by Claude E. Shannon (1948) to find fundamental limits on signal processing operations. It was based on probabilistic concepts about the nature of information. The area of the correct use of this theory is limited to three postulates: 1) only the transmission of information between technical systems is studied; 2) a message is treated as a limited set of characters; 3) the semantics of messages is excluded from the analysis. This theory gave engineers of data transmission systems the possibility of determining the capacity of the communication channels. It focuses on how to transmit data most efficiently and economically, and how to detect errors in its transmission and reception. Information theory is sometimes seen as the mathematical theory of information transmission systems, because the primary problems of this science arose from the domain of communication. It establishes the basic boundaries of data transmission systems, sets the basic principles of their development and practical implementation. Since its inception information theory has broadened to find applications into many other areas, e.g. data compression and channel coding.

Shannon's theory, which studies the transmission of information, is not engaged in the meaning (semantics) of the transmitted message. As an attempt to overcome its limitations, new versions of Shannon's mathematical theory of information appeared: topological, combinatorial, dynamic, algorithmic and others. However, they take into account only the symbolic structure of messages, and they can be attributed only to theories of syntactic, not semantic type. There is a later, complementary piece of information theory that draws attention to the content through the phenomenon of lossy compression of the subject of the message, using the criterion of accuracy. Unfortunately, the attempt made to connect a probabilistic-statistical (syntax) approach and semantic approach has not led yet to any constructive results. Nevertheless, it may be noted that the category information has infiltrated the relevant scientific fields, so it has obtained the status of a general scientific concept. To the original meaning of information were added:

– A measure of reducing the uncertainty as a result of the message;
– A collection of factual knowledge, e.g. circulating in the management process;

– The process of meaningful reflection of the diversity of natural objects, exchange of signals in the animal and plant world, the transmission of traits from cell a cell;
– The communication not only between machines, but also between man and machine, and between people.

1.3.1 Quantities of Information

Information is always presented as a message. The elementary units of messages are characters; characters assembled in a group create words. A message is issued in the form of words or characters is always transmitted in a material and energy form (e.g. an electric or light channel). When using structural measures of information only the discrete structure of the message, the amount of information elements it contains, the connections between them are taken into account. In a structural approach, geometric, combinatorial and additive measures of information are distinguished.
Geometric measure involves the measurement of the parameter of the geometrical model of the message (length, area, volume) in discrete units. A combinatorial measure of the amount of information is defined as the number of combinations of elements (characters). The possible amount of information coincides with the number of possible combinations, permutations and placement of elements. The most common is the additive measure (Hartley's measure) in accordance with it the amount of information is measured in binary units - bits.

1.3.1.1 Units for Measuring Computer Information

In computer technologies, the amount of information is the capacity of a data storage system or memory. Information capacity refers to a count of binary symbols stored in the memory; that is why it is a dimensionless quantity. In information theory, the units of information are used to measure the information contents (entropy) of sources of information and of communication channels. In both areas, the most common units of information are the bit and the byte (one byte is equivalent to eight bits). One-bit storage can save only two different symbols – 1 or 0 (logical *yes* or *no*). One-byte storage can save 256 different symbols – 0, 1, 2, 3…255. This means that a data storage system with 256 possible states can store up to $\log_2 256 = 8$ bits or 1 byte of information. The first, Ralph Hartley (1928), observed the logarithmic dependence of the amount of information that can be stored in a system on possible states of that system.
Multiples of information units are formed from bits (b) and bytes (B) with the SI[1]

1 International System of Units

or IEC[2] prefixes (see table 1). Units of information are not covered formally in the International System of Units, but computer professionals have historically used the same symbols and pronunciation for the binary series in the description of computer storage and memory. The simple reason was: *1 kB = 1024 B ≈ 1000 B*, 1MB = 1024 kB ≈ 1000 kB, but now a lot more information capacity is used and rounding errors become more noticeable, hence the necessity of the use of binary IEC units.

Table 1: Decimal and binary prefixes of information units.

Decimal (SI)			Binary (IEC)		
Value	Symbol	Name[3]	Value	Symbol	Name
1000	*kB*	kilobyte	1024	*KiB*	kibibyte
1000^2	*MB*	megabyte	1024^2	*MiB*	mebibyte
1000^3	*GB*	gigabyte	1024^3	*GiB*	gibibyte
1000^4	*TB*	terabyte	1024^4	*TiB*	tebibyte
1000^5	*PB*	petabyte	1024^5	*PiB*	pebibyte
1000^6	*EB*	exabyte	1024^6	*EiB*	exibyte

1.3.1.2 Quantities of Information in Information Theory

Q&A “What is information?”	**Technical definition:** Information is stimuli that have meaning in some context for its receiver. In computer science, information is a choice or entropy. The meaning of a message (in the human sense) is irrelevant. When information is entered into computer through interfaces and is stored in a computer memory, it is generally referred to as data. After processing (e.g. formatting and printing), output data can again be perceived as information. **Business definition:** Information is data that is accurate and timely, specific and organized for a purpose, presented within a context that gives it meaning and relevance, and can lead to an increase in understanding and decrease in uncertainty. Information is valuable because it can affect behavior, a decision, or an outcome. A piece of information is considered valueless if, after receiving it, things remain unchanged. For a technical definition of information, see information theory. http://www.businessdictionary.com

2 International Electrotechnical Commission

3 In the microelectronics industry, the names kilobyte (KB), megabyte (MB), and gigabyte (GB) are used for the value 1024 according to JEDEC standard, https://www.jedec.org/.

In information theory, the concept of information has been introduced as axiomatic (without explanation). Instead of answering the question „What is the information?“, the founders of information theory answer the question „How can we measure information?“ Harry Nyquist and Ralph Hartley (research leaders at Bell Labs when Shannon arrived in the early 1940s) were the most direct precursors of Shannon‘s theory. In considering the engineering of telegraph systems, Nyquist (1924) discusses *quantifying intelligence* – information transmitted by telegraph signals – and the highest *line speed* at which signals can be transmitted by a communication system. He gives the relation

$$W=K\log(m), \tag{1}$$

where W is the speed of transmission of intelligence, m is the number of different voltage levels to choose from at each time step, and K is a constant.
Hartley (1928) first used the term *transmission of information* in a technical sense, and makes explicitly clear, that information in this context was a measurable quantity. The amount of information he understood as the receiver’s ability to distinguish that the specific sequence of symbols had been intended by the sender rather than any other, and quantified as

$$H=\log (S^l), \tag{2}$$

where S was the number of possible symbols in the communication channel, and l – the length of message or in other words the number of symbols in the transmission. Intuitively, this definition makes sense: one symbol (e.g. one letter) has the information of $\log(S)$, then a sentence of length l should have l times more information.

The natural unit of information was therefore the decimal digit, much later renamed the hartley in his honor as a unit or scale or measure of information. The logarithm of total number of possibilities $H0$ is still used as a quantity for the Hartley information. This parameter characterizes the theoretical capacity of the information channel.

Shannon’s measure of information is the number of bits to represent the amount of uncertainty (randomness) in a data source, and is defined as entropy

$$H=-\sum_{i=1}^{n} p_i \log(p_i)\,. \tag{3}$$

Where there are n symbols $1, 2 \ldots n$, each with probability of occurrence of p_i.

Once again, it is worth recalling that Shannon was building a mathematical theory of communication. The amount of information in a Shannon information channel – is the number of random data at one site relative to another. Let x and y – the random variables defined on the corresponding sets X and Y. Then the amount of information x of y is the difference between a priori and a posteriori entropy:

$$I(x,y)=H(x)-H(x|y), \tag{4}$$

where

$$H(x)=\sum_{x}^{x\ in\ X} p(x)\log_2 p(x), \tag{5}$$

$$H(x\,|\,y)=-\sum_{y}^{y\ in\ Y} p(y)\sum_{y}^{y\ in\ X} p(x\,|\,y)\log_2 p(x\,|\,y). \tag{6}$$

Here, equation (5) describes the entropy. Equation (6) describes the conditional entropy, in communication theory it characterizes the noise in the communication channel.

1.3.2 Coding Theory

Coding theory is the study of the properties of codes and their fitness for a specific application. Codes are used for data compression, cryptography, error-correction and more recently also for network coding. Codes are studied by various scientific disciplines—such as information theory, electrical engineering, mathematics, and computer science—for the purpose of designing efficient and reliable data transmission methods. This typically involves the removal of redundancy and the correction (or detection) of errors in the transmitted data. There are essentially two aspects to coding theory:
1. Data compression (source coding)
2. Error correction (channel coding)

Source encoding attempts to compress the source data in order to transmit them more efficiently. This practice is found every day on the Internet where Zip data compression is used to reduce the network load and make files smaller.

Channel encoding, adds extra data bits to make the transmission of data more robust to disturbances present on the transmission channel. The ordinary user may not be aware of many applications using channel coding. A typical music CD uses the Reed-Solomon code to correct for scratches and dust. In this application, the transmission channel is the CD itself. Cell phones also use coding techniques to correct for the fading and noise of high frequency radio transmission. Data modems, telephone transmissions, and NASA all employ channel-coding techniques to get the bits through, for example the turbo code and LDPC codes.

Coding theory is a branch of applied mathematics concerned with transmitting data across noisy channels and recovering the original message. Coding theory is about making messages easy to read. Please, do not confuse it with cryptography which is the art of making messages hard to read.

Typically, the message in communication channels (such as a telephone line or CD stream) is in the form of binary digits or bits, strings of 0 or 1. The transmission of these bits takes place along a channel in which errors occur randomly, but at a predictable overall rate. To compensate for the errors we need to transmit more bits than there are in the original message.

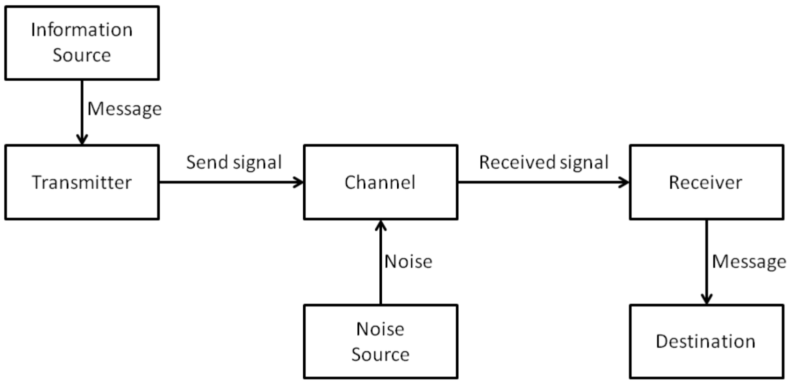

Figure 2: Shannon's model of communication.

The simplest method for detecting errors in binary data is the parity code, which transmits an extra "parity" bit after every 7 bits from the source message. However, this method can only detect errors; the only way to correct them is to ask for the data to be transmitted again.

A simple way to correct as well as detect errors is to repeat each bit a set number of times. The recipient sees which value, 0 or 1, occurs more often and assumes that that was the intended bit. The scheme can tolerate error rates up to one error in every two bits transmitted at the expense of increasing the amount of repetition.

1.3.3 Semiotics

Semiotics is the study of signs and symbols and their use or interpretation. It is the study of meaning making, the philosophical theory of signs and symbols. This includes the study of signs and sign processes (semiosis), indication, designation, likeness, analogy, metaphor, symbolism, signification, and communication. Semiotics is closely related to the field of linguistics, which, for its part, studies the structure and meaning of language more specifically. Where it differs from linguistics, however, is that semiotics also studies non-linguistic sign systems. Semiotics often is divided into three branches:

1. Semantics is the study of meaning, i.e. of the relationship of signs to what they stand for the relation between signs and the things (denotata) to which they refer.

2. Syntactics is the study of relations among signs in formal structures. Syntactical knowledge goes farther than what is grammatical; it also studies ambiguity, in which a sentence could have more than one meaning, and enables us to determine grammatical categories such as subject and direct object. It enables the use in sentences of terms, which we may not consciously be able to define.
3. Pragmatics is the study of the relation of signs to interpreters' sign-using agents; it is the interpretation of linguistic meaning in a given context. There can be a difference between linguistic context (the discourse that precedes a sentence to be interpreted) and situational context (the knowledge about the world). That is why comprehension the meaning of sentences can be a difficult task.

Semiotics is the science of communication and sign systems as well. In short, it is the science of the ways in which people understand phenomena and organize them mentally. It is also the science of the ways, in which people devise means for transmitting that understanding and for sharing it with others. Although natural and artificial languages are therefore central to semiotics, the field covers all non-verbal signaling and extends to domains whose communicative dimension is perceived only unconsciously or subliminally. Knowledge, meaning, intention and action are thus fundamental concepts in the semiotic investigation of phenomena.

Semiotics is often employed in the analysis of texts, which can exist in any medium and may be non-verbal. The term *text* usually refers to a message, which has been recorded in some way (e.g. writing, audio- and video-recording) so that it is physically independent of its sender or receiver. A text is an assemblage of signs (such as words, images, sounds, and gestures) constructed and interpreted with reference to the conventions associated with a genre and in a particular medium of communication.

The term *medium* is used in a variety of ways by different theorists, and may include such broad categories as speech and writing or print and broadcasting or relate to specific technical forms within the mass media (radio, television, newspapers, magazines, books, photographs, films and records) or the media of interpersonal communication (telephone, letter, fax, e-mail, video-conferencing, computer-based chat systems). Some theorists classify media according to the communication channels involved visual, auditory, tactile and so on. Human experience is inherently multisensory, and every representation of experience is subject to the constraints and allowances of the medium involved. Every medium is constrained by the channels, which it utilizes. Different media and genres provide different frameworks for representing experience, facilitating some forms of expression and inhibiting others. The more frequently and fluently a medium is used, the more transparent (invisible) to its users it tends to become.

Semiotics represents a range of studies in art, literature, anthropology and the mass media rather than an independent academic discipline. Those involved in semiotics include linguists, philosophers, psychologists, sociologists, anthropologists, literary, aesthetic and mass media theorists, psychoanalysts and educationalists. Beyond the most basic definition, there is considerable variation amongst leading semioticians as to what semiotics involves. It is not only concerned with (intentional) communication but also with our assignment of significance to anything in the world.

Semiotics teaches us not to take reality for granted as something having a purely objective existence, which is independent of human interpretation. It teaches us that reality is a system of signs. Studying semiotics can assist us to become more aware of reality as a construction and of the roles played by others and ourselves in constructing it. It can help us to realize that information or meaning is not contained in the world or in books, computers or audio-visual media. Meaning is not transmitted to us. We actively create it according to a complex interplay of codes or conventions of which we are normally unaware. According to semiotics, we live in a world of signs and we have no way of understanding anything except through signs and the codes into which they are organized.

1.3.3.1 Computational Semiotics

Computational semiotics is an interdisciplinary field that applies, conducts, and draws on research in logic, mathematics, the theory and practice of computation, formal and natural language studies, the cognitive sciences generally, and semiotics proper. A common theme of this work is the adoption of a sign-theoretic perspective on issues of artificial intelligence and knowledge representation. Many of its applications lay in the field of human-computer interaction (HCI) and fundamental devices of recognition.

Computational Semiotics refers to the attempt of emulating the semiosis cycle within a computer. It refers to the study of the computer as a medium or distinct sign system and its central concern is the extent to which the information inside a computer can be considered semiotic or can be usefully analyzed in semiotic terms. Among other things, this is done for the purpose of constructing autonomous intelligent systems able to perform intelligent behavior, what includes perception, world modeling, value judgment and behavior generation. There is a claim that most intelligent behavior should be due to semiotic processing within autonomous systems, in the sense that an intelligent system should be comparable to a semiotic system. Mathematically modeling such semiotic systems is being currently the target for a group of researchers studying the interactions encountered between semiotics and intelligent systems.

The key issue in this study is the discovery of elementary or minimum units of intelligence, and their relation to semiotics. Some attempts have been made to determine such elementary units of intelligence, i.e., a minimum set of operators that would be responsible for building intelligent behavior within intelligent systems. Within computational semiotics, we try to depict the basic elements composing an intelligent system, in terms of its semiotic understanding. We do this by the definition of a knowledge unit, from which we derive a whole taxonomy of knowledge. Different types of knowledge units are mathematically described and used as atomic components for an intelligent system. They are at the same time, containers of information and active agents in the processing of such information.

Computational semiotics is a branch of one, which deals with the study and application of logic, from computation to formal, to natural language in terms of cognition and signs. One part of this field, known as algebraic semiotics, combines aspects of algebraic specification and social semiotics, and has been applied to user interface design and to the representation of mathematical proofs.

Semiotics deals with the basic ingredients of intelligence [12] (signs or representations, object or phenomenon, interpretants or knowledge) and their relationships, the triple (sign, object, interpretant) represents a signic process. To perform semiosis is to extract meaning of an object or phenomenon. In the process of extracting meaning, it essentially studies the basic aspects of cognition and communication. Cognition means to deal with and to comprehend phenomena that occur in an environment. Communication means how a comprehended phenomenon can be transmitted between intelligent beings.

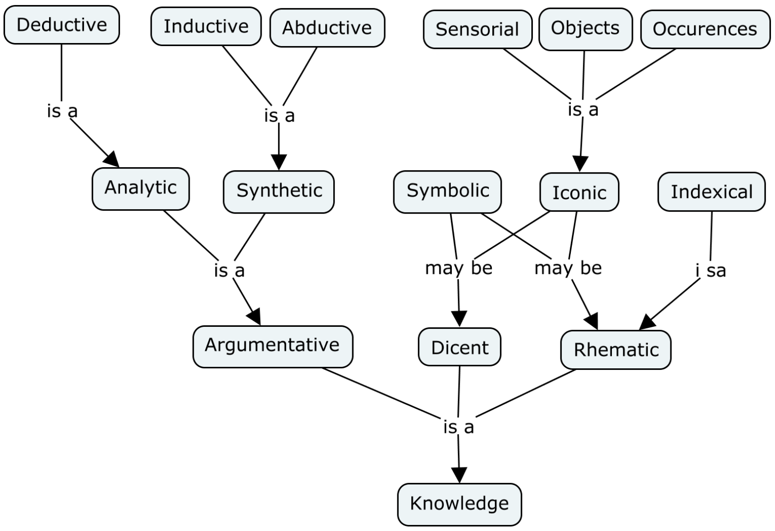

Figure 3: The map[4] of knowledge unit concepts classified by its nature (in accordance with Ricardo R. Gudwin [13]).

4 The map of concept (concept map) includes concepts, enclosed in boxes with rounded corners, and relationships between concepts or propositions. Relationships are indicated by a connecting line and linking words, which specify the relationships between concepts.

Signs are the basic units of analysis in semiotics; they are representation, which are either internal or external to a cognitive system, which can evoke internal representations called interpretants that represent presumable objects in the world. Interpretants, too can act as signs evoking other interpretants. The taxonomy of signs provides three distinct kinds of knowledge, relates to the experience of an object, to experience of a sign of an object and to the other experience of the mapping between these two concepts. A sign can be a *mere quality*, an *actual existent* or a *general law*; a sign interpretant represents it as a sign of possibility (*rheme*), fact (*dicent*) or reason (*argument*).

Basic rhematic knowledge[5] can be:

- Symbolic – is knowledge of arbitrary names like 'dog' or any symbol which has a conventional and arbitrary relationship to its referent;
- Indexical – is knowledge of indices – signs that are not conventionalized in the way symbols are but are indicative of something in the way that smoke is indicative of fire;
- Iconic – is knowledge of signs that resemble their referents or provide direct models of phenomena (as such, icons unlike symbols are not arbitrarily related to their referents.

Dicent knowledge employs truth-values to link sensorial, object or occurrence knowledge to world entities. It has two types: iconic (truth-values of iconic proposition are derived directly from iconic rhematic knowledge) and symbolic (truth-values of symbolic proposition match those of their associated propositions, because symbolic propositions are names for other propositions, which may be either iconic or symbolic).

Argumentative knowledge is knowledge used to generate new knowledge through inference or reasoning. There are three types: deductive (is categorized as analytic, meaning it does require knowledge of the word), inductive (synthetic, involves inference from a large number of consistent examples and a lack of counter examples) and abductive (synthetic, valid inferences do not contradict previous facts).

5 Pertaining to the formation of words or to the rheme of a sentence.

2 Algorithmics

2.1 The Science of Algorithms

The *algorithm* is a fundamental notion to mathematics and informatics; it was created far before the invention of modern computers. Originally, an algorithm referred to a procedure of arithmetic operations on decimal numbers; later the same term began to be used to designate any series of actions that leads to a solution for a problem. In the field of informatics, an algorithm is understood to be the exact and finite list of instructions that defines the content and the order of enforcement actions, which are done by the executor on certain objects (source and in-between data sets) in order to obtain the expected result (final data sets).

Any algorithm depends on the executor; the description of the algorithm is performed using the executor's commands; objects which can be acted on by executor must belong to its environment, so that the input and output data of the algorithm must belong to the environment of a particular executor. The meaning of the word algorithm is similar to the meaning of words such as rule, method, technique or the way. In contrast to rule and methods, it is necessary that an algorithm have the following characteristics:

– *Discreteness* (discontinuity). An algorithm consists of a limited number of finite actions (steps); only after finishing the current step, the executor can proceed to the next step. The executor identifies each successive action solely based on statements recorded in the algorithm; such instruction is called a command.
– *Determinacy*. The way to solve the problem is unequivocally defined as a sequence of steps; this means that the algorithm used several times for the same input data will always result in the same set of output data.
– *Understandability*. An algorithm should not contain ambiguous instructions and prescriptions; the executor should not undertake any independent decisions.
– *Effectiveness*. If each step of an algorithm is executed precisely, the computation should be completed in a real time by delivering a solution to the problem; one possible solution is no result (an empty set of result data).
– *Mass character*. An algorithm should work correctly for some types of problems (not for a single problem); an algorithm's usefulness area includes a number of tasks belonging to the defined category of the problem.

There is a fundamental difference between executing and developing algorithms. To execute an algorithm, you have to have an executor (a performing machine), for which the algorithm has been developed and saved. To develop the algorithm, you have to have a storage medium, onto which the contents of the algorithm can be saved. There are several indirect forms of describing the same algorithm; you can describe it:

- In text form, using a natural language or a specially defined algorithmic language;
- In the graphic form, e.g. using a flow chart;
- In analytical form as a sequence of formulas;
- In the form of a computer program, more precisely, in a programming language.

Regardless of the form of representation, algorithms can be translated into common control and data structures provided by most high-level programming languages. Different forms of presentation of algorithms are preferred to precisely analyze the temporal and spatial requirements. Regardless of the form of description, their structure comprises steps connected into serial-chained, branched or cyclic fragments. Edsger W. Dijkstra has shown that these three structural units are the only ones needed to write any algorithm.

The science of algorithms (sometimes called *algorithmics*) is classified as a branch of computer science (in a historical perspective it was a branch of cybernetics) and occurs as a sector in the majority of natural sciences, economics and technology. The algorithmization or the art of building algorithms is called *algorithm design*; the result of algorithmization is a procedure (defined process), which solves efficiently a class of problem. Also within the scope of algorithmics, there is the study of the difficulty of solved problems; *algorithmic complexity theory* deals with this. A branch of algorithmics, called *algorithm analysis*, studies the properties of solved problem. Analysis defines resources needed by the algorithm to solve this problem.

2.1.1 Algorithm Design

From a practical point of view, an algorithm is a set of steps to be adopted by the computer code to reach specific informational goals; it is based on the idea of a solution to a problem. Therefore, the development of algorithms is the most important structural component of programming; the algorithm should not depend on the syntax of programming languages and the specifics of a particular computer, to be reusable. One can say that a program is a specific implementation of an algorithm, when the algorithm is the idea of the program. In a certain sense, the creation of algorithms is not engineering, because this activity contains elements of art; nevertheless, there are several different engineering approaches for algorithm design [14]. Sometimes, this general approaches are called *algorithmic paradigms*; they were formulated to constructing of efficient solutions to solving a broad range of diverse problems.

Operational Approach

Approaches and requirements for the development of algorithms have changed significantly during the evolution of computers. During the first generations, when computer time was expensive, and their ability was modest in terms of today's

achievements, the basic requirement for algorithms was narrowly understood as their effectiveness. The criteria were:
- Use the smallest number of memory cells when the program is executed;
- Achieve minimum execution time or the minimum number of operations.

In this case, the processor has executed the program commands nearly directly; the most frequent commands were an assignment statement, simple arithmetic operations, comparisons of numbers, unconditional and conditional jumps, subroutine calls. Such an approach to the creation of algorithms and to programming, focused on an operation directly executed by a computer is called an *operational approach*. Let us consider the basic steps of algorithms that are performed by a computer assuming the operational approach.

An *assignment statement* copies the value into a variable memory cell; this cell may belong to the main computer's memory or be one of the processor's registers. After an assignment, specified value is stored in the memory cell, where it is located until it will be replaced by another assignment. The memory cell, which houses the value, is indicated in the computer program by a name (identifier). As the variables and their values can be of different types, and values of these types are coded and represented in computer memory in different ways, they must match each other.

A set of *simple arithmetic operations* allows us to record arithmetic expressions using the numeric constants and variable names.

Comparison of numbers operations are actually reduced to the determination of the sign of the difference, which is displayed by a special memory (flag of the result) of a computing device and can be used in the performance of *conditional jumps* between the step of an algorithm. A conditional jump changes the order of command execution depending on some condition, most often the conditions of a comparison of the numeric types. In contrast, an *unconditional jump* changes the order of the commands independent from any conditions.

A *subroutine call* operation interrupt the normal order of execution steps and jumps into a separate sequence of program instructions (steps) to perform a specific task; this separate sequence is called subroutine and is packaged as a unit of code.
The use of an operational approach provokes certain drawbacks in the resulting code; the misuse of conditional and unconditional transitions often leads to a confusing structure of the program. A large number of jumps in combination with treatments (to increase the efficiency of the code) lead to the fact that the code may become incomprehensible, very difficult to develop and to maintain.

Structural Approach
Since the mid-1960s, computer professionals have obtained a deeper understanding of the role of subroutines as a means of abstraction and as units of code [15]. New programming languages have been developed to support a variety of mechanisms of parameter transmission. These have laid the foundation of *structural* and *procedural*

programming. Structural programming is a software development methodology, based on the idea of program architecture as a hierarchical structure of units or blocks (this idea was formulated by Edsger W. Dijkstra and Niklaus Wirth). This new programming paradigm was able to:

- Provide the programming discipline that programmers impose themselves in the process of developing of software;
- Improve the understandability of programs;
- Increase the effectiveness of programs;
- Improve the reliability of programs;
- Reduce the time and cost of software elaboration.

The structural approach methodology enabled the development of large and complex software systems. At the same time, it gave birth to structure mechanisms for algorithms and program codes, and a control mechanism for proving the correctness of calculations. A routines mechanism was implemented in form of procedures and functions, which are powerful programming tools.
There are four basic principles of structural methodology:

1. *Formalization* of the development process guarantees the adherence to a strict methodological approach; usually, programming should be engineering, not art.
2. The hierarchy of levels of *abstraction* requires building an algorithm divided into units, which differ in the degree of detail and approximation to the problem being solved.
3. In accordance with the maxim "*Divide and conquer*", the splitting of a whole problem into separate sub-problems, allowing the independent creation of smaller algorithms is a method for complex projects. The fragments of solution code can be compiling, debugging and testing separate as well.
4. The *hierarchical ordering* (hierarchical structuring) should cover relationships between units and modules of software package; this means respect for the hierarchical approach up to the largest modules of the software system.

A structural approach can be applied step by step, detailing of parts of an algorithm, until they are quite simple and easily programmable. Another way is the developing of lower-level units, and further combining them into units with higher levels of abstraction. In practice, both methods are used.

Modular Approach

In computer programming, a module is a set of related subroutines together with the data that these subroutines are treated. The terminology of different programming paradigms may provide another name for subroutine; it may be called a subprogram, a routine, a procedure, a function or a method. Gradually, software engineers formed the concept of a *module* as a mechanism of abstraction; a dedicated syntax was developed for modules to be used on par with subprograms. The construction of modules hides the implementation details of subprograms, as opposed to subroutines, because they are under the effect of global variables. The prohibition of access to data from outside the module has a positive effect on a program; it prevents accidental changes and therefore a violation of the program. To ensure the cooperation of modules, a programmer needs only to consider the construction of an interface and a style of interaction for the designated modules in the whole program.

Any subroutine or group of subroutines can constitute the module: the ability to determine the structure and functions of the program modules depends on the qualification and the level of training of the programmer; in larger projects such decisions are taken by a more experienced specialist, called *software architect*. According to recommendations, a module should implement only one aspect of the desired functionality. Module code should have a header, which will include comments explaining its assignment, the assignment of variables passed to the module and out of it, the names of modules that call it and modules that are called from it.

Not all of the popular programming languages such as Pascal, C, C++, and PHP originally had mechanisms to support modular programming. It is worth noting that a modular approach can be performed even if the programming language lacks the explicit support capabilities for named modules. Modules can be based on data structures, function, libraries, classes, services, and other program units that implement desired functionality and provide an interface to it. Modularity is often a means to simplify the process of software design and to distribute task between developers.

The choice of a design approach is an important aspect of algorithm production. More than one technique could be valid for a specific problem; frequently an algorithm constructed by a certain approach is definitely better than alternative solutions.

2.1.2 Algorithmic Complexity Theory

Many algorithms are easy to understand, but there are some which are very difficult to follow. However, when professionals argue about the complexity of algorithms, they usually have something else in mind. When an algorithm is implemented as a program and the program is executed, it consumes computational resources; the most important resources are processor time and random access memory (RAM) space. These two parameters characterize the complexity of an algorithm in the conventional sense today. The time and memory consumed in the solution of the problem, are called the *time complexity* and the *space complexity*.

The development of industrial technology has led to cheap and compact RAM. Nowadays, RAM is not a critical resource; generally, there is enough memory to solve a problem. Therefore, most often the complexity of the algorithm should be understood as time complexity. Ordinary time T (seconds) is not suitable to be the measure for the complexity of the algorithms; the same program at the same input data may be performed in different time on different hardware platforms. It is easy to understand that the increase in the number of input data will always cause an increase in the execution time of the program. This is why the time complexity is particularly important for applications that provide an interactive mode of the program, or for complex control tasks in real time.

A time complexity should not be dependent on the technical specification of a computer. It should be measured in relative units; typically, the number of performed operations estimates the time complexity. Here are some examples of basic operations: one arithmetic operation, one assignment, one test (e.g., x>0), one reading or writing of a primitive type. However, the number of elementary operations is often a useless measure for algorithm complexity; not always is clear which operations should be considered as elementary. In addition, different operations may require for their performance a different amount of time, and furthermore, the transfer of operations used in the description of the algorithm into the operations performed by a computer depends on the properties of the compiler and on such unpredictable factors as programmer skill. The assertion that a certain algorithm requires *N* operations does not mean anything in practice. Practitioners are most interested in the answer to the question of how the program execution time will increase with the increase in the number of input data.

Consider an algorithm that adds the *n* numbers $x_1, x_2, \ldots, x_n$. Here it is:

```
Sum = 0;
for i=1 to n do
  { Sum = Sum + x[i] };
```

If the numbers are not large and their sum can be calculated using standard addition, it is possible to measure the input data volume as the number *n* of summands. Assuming that the basic operation is addition, we can conclude that the complexity of the algorithm is equal to n. The time for addition is linear in the number of items (summands); if the problem size doubles, the number of operations also doubles.
Consider another example – a sequential search algorithm. It is looking for an element *y* in the *n*-elements set *x*, scanning sequentially all the elements of the set $x_1, x_2, \ldots, x_n$. The search result is the value of the variable *k;* if the desired element is found $y = x_i$, it is equal *i*, otherwise $k = n + 1$.

```
k = n + 1;
for i = 1 to n do
  { if y = x[i] then
     { k = i }
  };
```

Here, the basic operation is a conditional statement *if*. The number of comparisons made in the best case will be equal to *1*, and in the worst case *n*. It may be noted that on average there will be about n/2 basic operations made. In practice, the two most common measures of complexity of algorithms that are used are the complexity in the worse case and the average complexity.

Here are a few comments on the practical treatment of the complexity of algorithms. Firstly, a programmer should often find a compromise between computation accuracy and execution time; generally, the increase of accuracy requires an increase in time (an increase of time complexity of algorithm). Secondly, spatial complexity of the program becomes critical when the volume of data to be processed reaches the limit of the amount of RAM of a computer. On modern computers the intensity of this problem is reduced both by increasing the volume of random access memory, and by the efficient use of multilevel systems of storage devices (swapping, caching and buffering processes). Thirdly, the time complexity of the algorithm may be dependent both on the number of data and of their values. If we denote the value of the time complexity of the algorithm α by symbol T_α, and designate by V a numerical parameter that characterizes the original data, the time complexity can be represented formally as a function $T_\alpha(V)$. Selecting V will depend on the problem or the type of algorithm used for solving this problem. Here is an example of calculating the factorial of a positive integer *n*:

```
f = 1;
for i = 2 to n do
  { f = f * i };
  factorial =  f;
```

In this case, it can be said that the time complexity depends linearly on the data parameter *n* – the value of the factorial function argument; here the basic operation is multiplication.

In computer science, the characteristic of an algorithm which relates to the amount of resources used by the algorithm, is called *algorithmic efficiency*; it can be considered as an equivalent to engineering productivity for repeating or continuous processes; for maximum efficiency, one should to minimize resource usage. However, processor time and memory space as different resources cannot be compared directly, so which of the algorithms when compared is considered more efficient often depends on which measure of efficiency is being considered as the more important. In practice, efficiency of an algorithm or piece of code covers not only CPU time usage and RAM usage; it may include the usage of lots of resources, e.g. disks, network.

Computational Complexity Theory

In a wider perspective, the concept of the complexity of the algorithm can be used to characterize computational problems, namely the parametric evaluation of their intrinsic difficulty. A branch of the theory of computation, called *computational complexity theory*, focuses on classifying problems according to their difficulty (by means of computational complexity). This work assumes that the more difficult problem will have the more complex algorithm describing its solution. Here is a five-step process to solving computational problems:

1. Build a clear declaration of the problem.
2. Find a mathematical and logical model for the problem; usually, it is associated with making some simplifications to shift from the problem to the model.
3. Using the model, settle upon a method for solving the problem on a computer; remember what a computer can and cannot do.
4. Implement the method (carry it out with a computer).
5. Assess the implementation; estimate obtained answers to see if they make sense, and try to identify errors or unjustified assumptions. In case of a relatively large disparity between the correct and the obtained result, reassess the problem and try the process again.

The essential parts of the problem solving process should include understanding the problem and good intuition. Informaticians should deal with multidisciplinary problems; as computer professionals, they must be able to take not theoretical but real problems that involve specific information from the different areas (physics, chemistry, biology, psychology, politics, etc.) and find solutions, which will be finally rendered as instructions operating on data. Obviously, computational complexity theory studies also its own problems, but there are primary problems, which engineers and scientists from other disciplines might be interested in. Here are some examples of such problems:

– A decision problem, which is one that can be formulated as a polar question (i.e., with “yes” or “no” answer) in some formal system.
– A search problem, which is not only about existence of solution but about finding the actual solution as well.
– Counting problems, which are requests for the number of solutions of a given instance.
– An optimization problem, which has more solutions but requires the best one.

The notion of efficiency is one of the most important things in computational complexity theory. According to convention, computation is efficient if its execution time is bounded by some polynomial function of the size of input; it is said that a calculating machine runs in polynomial time. This is possible when the algorithm uses some vital knowledge about the structure of the problem; it will result in proceeding with much better efficiency than “brute force”, which corresponds to the exponential time.

The theory of algorithms introduces *classes of complexity*; these are sets of calculation tasks characterized by approximately the same height of resource requirements. In every class, there is a category of problems, which are most difficult; this means that any problem in the class can be mapped to those problems. Such difficult problems are called *complete problems* for this class; the most famous are NP-complete problems. Complete problems are used to prove the equality of classes; instead of comparing sets of tasks just compare the complete problems belonging to these sets. The previously mentioned NP-complete problems, in some sense, form a subset of the typical problems in NP class; if for some of them a polynomial fast solution algorithm is found, then any other task in the NP class can be solved in the same fast tempo. Here, NP class can be defined as containing tasks that can be solved in real time on the non-deterministic Turing machine. Unfortunately, a more detailed presentation of this issue requires a very specialized knowledge of mathematics and computer science especially in the area of theoretical modeling of computer systems, which is used in automata theory.

2.1.3 Algorithm Analysis

Analysis of algorithms is a branch of applied informatics that provides tools to examine the efficiency of different methods of problem solutions. The analysis of the algorithm is intended to discover its properties, which enable the evaluation of its suitability for solving various problems; by analyzing different algorithms for the same application area, they can be compared with each other (ranking of algorithms). The practical goal of *algorithm analysis* is to foresee the efficiency of diverse algorithms in order to conduct a software project. In addition, an analysis of the algorithm may be the reason for its reengineering and improvements to simplify a solution, shorten code, and improve its readability.

A naive approach to comparison of algorithms suggests implementing these algorithms in the programming language, compiling codes and running them to compare their time requirements. In fact, this approach suggests comparing the programs instead of the algorithms; such an approach has some difficulties. Firstly, implementations are sensitive to programming style that may obscure the matter of which algorithm is more effective. Secondly, the efficiency of the algorithms will be dependent on a particular computer. Lastly, the analysis will be dependent on specific data used throughout the runtime. That is why, to analyze algorithms, one should utilize mathematical techniques that will analyze algorithms independently of detailed implementations, hardware platform, or data. This explanation has repeated evidence that was previously mentioned while explaining the complexity of algorithms. This is not a coincidence, because a professional approach to the analysis of algorithms provides an assessment of their complexity, effectiveness and efficiency through objective and independent determination of their categories or class.

Indeed, software engineers measure such parameters as the actual time and space requirements of a program, the frequency and the duration of subroutine calls, or even the usage of specified instructions. The results of these measurements serve to aid program (not algorithm) optimization; this activity is called *program profiling* and is a form of *dynamic program analysis*.

According to the strict language of mathematics, informaticians express the complexity of algorithms using big-*O* notation [16] that is a symbolism used in mathematics to describe the asymptotic behavior of functions. Generally, it tells us how fast a function grows. The letter *O* is used because the rate of growth of a function is also called its *order*. Formally, a function $T_\alpha(N)$ is $O(F(N))$ if for some constant *c* and for all values of *N* greater than some value n_0: $T_\alpha(N) \leq c{*}F(N)$. For a problem of size *N* a constant-time method is order 1: $O(1)$ and a linear-time method is order *N*: $O(N)$. Of course, $T(N)$ is the precise complexity of algorithm as a function of the problem size *N*, and selected *growth function F(N)* is an upper limit of that complexity. Algorithms can be categorized based on their efficiency; the following categories are often used in order of decrease of efficiency and increase of complexity:

Complexity	**Growth function**	**Comments**
O(1)	Constant	It is irrespective of size of data
O(log n)	Logarithmic	Binary search in a sorted array
O($\log^2$ n)	Log-squared	
O(n)	Linear	Finding an item in an unsorted array
O(n log n)	Linearithmic	Popular sorting algorithms
O(n^2)	Quadratic	Polynomial time algorithms
O(n^3)	Cubic	
O(a^n)	Exponential	Generally, brute-force algorithms
O(n!)	Factorial	

Let us see how to analyze an example algorithm. In the beginning, we need to count the number of important operations in a particular solution to assess its efficiency. Each operation in an algorithm has a cost (take a certain amount of time) c_i. The total cost of a sequence of operations is $C=c_1+c_2+...+c_n$; loop statements cost should be calculated from the cost of a block of operations multiple times, for all iterations:

Nr	**Instruction**	**Cost**
00	// Sum of sequential numbers from 1 to n	
01	i = 1;	c_1
02	sum = 0;	c_2
03	while (i <= n) {	$c_3\ (n + 1)$
04	i = i + 1;	$c_4\ n$
05	sum = sum + i;	$c_5\ n$
06	}	

Total cost $C = (c_3 + c_4 + c_5)\ n + (c_1 + c_2 + c_3)$

It is easy to notice that the time required for this algorithm is proportional to n. If a nested loop is added to this algorithm to change the logic of sum counting, the time required for a new algorithm will be proportional to n^2:

Nr	Instruction	Cost
00	// Double sum of sequential numbers from 1 to n	
01	i = 1;	c_1
02	sum = 0;	c_2
03	while (i <= n) {	$c_3\ (n + 1)$
04	j = 1;	$c_4\ n$
05	while (j <= n) {	$c_5\ (n + 1)$
06	sum = sum + i;	$c_6\ n^2$
07	j = j + 1;	$c_7\ n^2$
08	}	
09	i = i + 1;	$c_8\ n$
10	} //	

Total cost $C = (c_6 + c_7)\ n^2 + (c_3 + c_4 + c_5 + c_8)\ n + (c_1+c_2+c_3+c_5)$

When determining the cost of the operation, it should be taken into account that most arithmetic and indexing operations are constant time (runtime does not depend on the magnitude of the operands). Addition and subtraction are the shortest time operations, multiplication usually takes longer, and division takes even longer than they do. Sometimes the operands are very large integers; in such a situation, the runtime increases with the number of digits. If the algorithm contains conditional statement (e.g., *if condition then s1 else s2*), the cost of this fragment should be counted by adding the condition test running time and the larger running time of *s1* and *s2*. In general, the built-in functions tend to be faster than user-defined functions because they are implemented more efficiently.

After counting the number of important operations, we should express the efficiency of the complete algorithm using growth functions. It is important to know how rapidly the algorithm's time requirement grows as a function of the problem size. The efficiency of two algorithms can be compared via juxtaposition of their growth rates. Typically, two algorithms with the same growth rate are dissimilar only according to constant coefficients, but the different growth rate explicitly indicates a good algorithm and a bad algorithm. Informaticians who care about performance of algorithms often have an aversion to this kind of conclusion; sometimes the coefficients and the non-leading conditions make a real difference (e.g., the details of the hardware, the programming language, and the characteristics of the input). Furthermore, for small problems, asymptotic approximation is irrelevant; it is useful at least for big computational problems.

2.2 Data Science (Datalogy)

Data are the basis for any analytical or explorative business. Data science is a new interdisciplinary area in which informatics plays an essential role beside mathematics and statistics. Data science deals with large amounts of data and facilitates the acquisition of knowledge from such large collections. The morphology and semantic of terms *data science* and *datalogy* suggests that both can be used interchangeably. Historically, in connection with the central role of data in the construction of a computing system, Peter Naur has suggested the name datalogy for the whole of computer science. The design technique, data driven design, which was very popular in the last century, was evidence of the advantage of using this term. The rapid development of technology and the theory of relational databases also pointed to the dominant role of data in computer systems. In the new century, the vast change of the idea of large data centers into the new idea of distributed computing network systems caused the need of new data structures and new methods of data analysis. Currently, the demand for services related to the processing of huge data sets is increasing incredibly; professionals need theoretical support in this area – they need new models and methodologies as well.

2.2.1 Raw Data

We will call *raw data* (sometime, unprocessed data) every kind of data (e.g., commercial data, experimental data, statistical data), which have been collected in the process of observation, research, measuring, and monitoring, but have not been transformed or structured artificially (this means, they are not processed and behave like a natural structure). In order for such data to be available to computers, they must be digitized; a digital data source (generally) can be any of database, computer file, or even a live data feed like a network data stream. Digital data are stored on hard drives, on CD (DVD) or on flash memory; to be processed, they always should get to RAM. The data might be located on the same computer as the software that processes them, or on another computer (server) anywhere on a network.

Data does not magically come into view – somebody must have gathered it. Information and knowledge comes from data through processing them; the place where data are obtained from, is called the *data source*. The *original source*, which supplies *primary data*, is distinguished from the *secondary source*, which supplies secondary or *indirect data*. Usually, primary data are gathered (some time even developed or generated) by the informatics-analyst exclusively for the research project. The advantages of such data include the control over the process of data collection, the formal specification of data, and the dynamic management of data collection based on the previously stored values. Building an original data source requires a lot of time; very often in practice, the original source will be smaller than the secondary sources. Therefore, analysts often use secondary data sources; they contain data that has previously been gathered by somebody else and/or for some other purpose. One can obtain from secondary sources a huge series of data; compared to primary data gathering this approach is usually less expensive. The disadvantage of indirect data may be its lower quality. It may be inaccurate, not current or biased.

To be workable, data sources should hold not only raw data themselves but also information on *metadata* and *connectivity settings* such as the server address, table name, login, etc. Metadata is a data about data; there are catalogs, directories, registry, which contain the information about the composition of data, content, status, source, location, quality, format and presentation, access conditions, the acquisition and use, copyright, property and related rights to data. Metadata are the information that describes the content of the database. Metadata inform users when a piece of data has been updated for the last time, its format and what is supposed to apply it. In a practical sense, this information can serve as a reference for a user while working with the data source, and helps him to understand the meaning and context of data. Metadata are used to improve the quality of search results; a data source enables relatively complex operations for simultaneous search and filtering by informing the computer about the relationships that exist between its data elements. This approach (called *knowledge representation*) is in the interests of artificial intelligence and the semantic web. IT professionals call a structured metadata the *ontology* or the *schema* (for example, XML-schema). A metadata schema announces a group of terms, their definitions and relationships; the terms are often referred to as *elements*, *attributes* and *qualifiers*, the definitions make available the semantics of data elements. High-quality schema makes the data source understandable both for machine and for human through presentation of ontology.

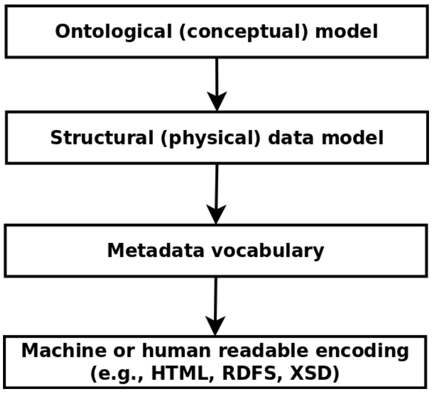
Ontological (conceptual) model
Structural (physical) data model
Metadata vocabulary
Machine or human readable encoding (e.g., HTML, RDFS, XSD)

Figure 4: The role of metadata in data set management.

Very often, the big volume of obtainable data, the variety of their formats and the high velocity of data actualization makes it more difficult to carry out large analytical projects. IT professionals use a special technical process, called *data integration*, to combine disparate sources into one significant and valuable data source. Data integration provides standardized access to data collected from heterogeneous and distributed data sources; it includes locating, monitoring, purifying, transformation and delivery of data from dissimilar data sources.

2.2.2 Data Structures

In a computer memory, all data are represented by binary codes, which cannot be distinguished directly. To recognize the different types of data (like text and numbers or like vectors and scalars), computer programmers use data type classification. Before using the data, a programmer should declare their types; after declaration, in order to ensure the faultless results of calculations, data types should be referenced and used properly. Data types serve to explicitly process and store data in a particular way; understanding data types allows programmers to design programs efficiently. Unfortunately, for historical reasons, such declarations and the list of possible data types are dependent on the programming language. However, there are some common notions and senses; all programmers make use of technical terms *primitive*, *built-in* and *compound* data types.

Some data types are primitive in the sense that they are basic and cannot be decomposed into simpler data types. From these basic types, one can define new compound types. Common examples of such basic data types are integer numbers, floating-point numbers, characters, pointers, and Booleans. Some categories of compound types are arrays, structures, unions, objects and alphanumeric strings. A built-in type is distinguished by the fact that the programming language (compiler) provides built-in support for it. In general, all primitive data should be supported by programming languages, not only when they fall within a group of built-in data types within a given programming language. For example, the alphanumeric string can be regarded as built-in data in some programming languages, in some others it is regarded as array of characters.

As there are only a few very important ways of organizing non-primitive data and data sets in computer memory or in storage, let us analyze them. The first approach is based on the declaration of *data structures* – the collection of elements (datum) of possibly different types. The architecture of a specific kind of data structure is matched to the solved problem, and the data items contained within such a structure are provided to be accessible and secure. The process of creating and using data structures is based on pointers – basic computer data that keep the memory addresses of structures; instead of handling the entire complicated data structure, it is often sufficient to perform operations only on these primitive pointers. In informatics, certain sorts of data structures that have similar behavior are represented by a mathematical model called an *abstract data type* (ADT), which is used by programmers to manage data during calculation. Data items have both a logical and a physical form – the meaning of the data item within an ADT, and the implementation of the data item within a data structure respectively.

The oldest data structures are *arrays*, they hold a specified number of homogeneous elements (data of the same data type); each element of the array is identified by an array index (pointer) and can be accessed individually. The entire advantage the arrays show in mass calculation, e.g. in loops. The most clear in a linear data structures, into which all elements have an order (each element has a predecessor and a successor); in the linear data structure one element is first, and one is last. The examples of *linear data structure* are a list (an ordered collection of nodes), a stack (last-in-first-out data structure), and a queue (first-in-first-out data structure). *Hierarchical data structures* are more complicated. They are organized like a tree, in which the first element (the root) can have more than one successor (the leave), and all leaves can have one or many successors as well. A completely disordered collection might seem to be a *graph data structure*; its nodes may have "many to many" relationships with other nodes of data sets, and there are no constraints on the numbers of predecessors or successors.

An alternative approach to the management of data items is provided by an object-oriented paradigm. In this approach, data are attributes of *object*, they can be represented by primitive or compound data types, or by another objects. Objects consists of attributes and data and are responsible for themselves – the attributes (data) have to know what state the object is in, the methods (operations and functions possible on this data) have to work on data properly. By packing of data and functions into a single component, data are encapsulated (closed in the separate memory area); that means, programmers can effectively manage them.

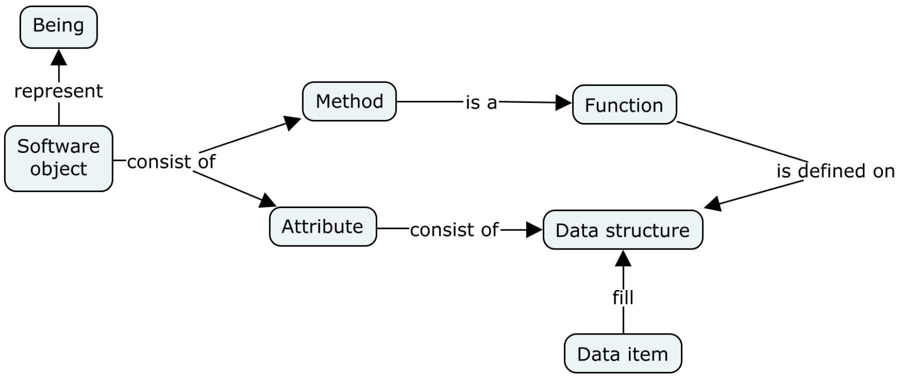

Figure 5: Software object idea (concept map).

Database specialists prefer a slightly different approach; they offer *records* (also called row or tuple) as a basic data structure for all storage medium, including the main memory of computers. Records are a single, implicitly structured complete set of information; the main goal of this specific data structure is the ease and speed of search and retrieval of data representing of the entity. Other database components are fields and files; a record is a collection of related fields, a file is a collection of records. The database is stored logically as a collection of files.

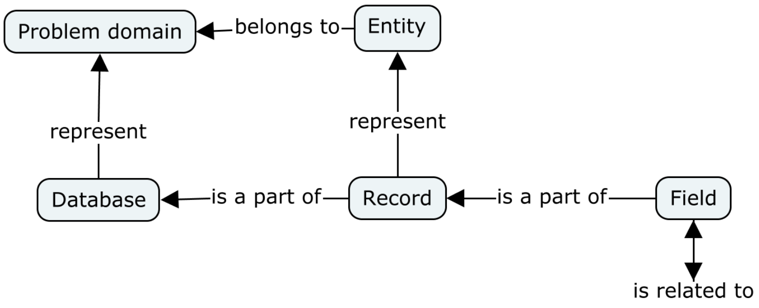

Figure 6: The record as a set of information (concept map).

2.2.3 Data Analysis (Data Analytics)

Data do not have their own meaning; data are facts (details), but the informatician-analyst has to see truth (regularity) behind facts by discovering rules hidden in the data. Even the greatest amount of the best quality data means nothing if they have not been analyzed. The primary activity of the analyst is to turn raw data into useful information. The aim of data analysis should be to search for the answers to questions that are asked by software engineers and domain specialists. Then the information and knowledge gained from the data will be useful in the context of the development of new software and will contribute to progress in the domains that are problematic for informaticians. It should be noted that the relatively new term *data analysis* is not successful, since the word "analysis" in mathematics has an established meaning and is the name of many of the classical sections (e.g., mathematical analysis, functional analysis, complex analysis, discrete analysis). In data analysis, there is not the study of the mathematical apparatus, which is based on some fundamental results. There is not a finite set of basic facts, from which follows how to solve the data problems especially that many such problems are needed to develop an individual mathematical apparatus. The science that studies raw data in order to draw conclusions about the information contained in them is called data analytics.

Data analysis is a field of informatics, engaged in the design and study of the most common computational algorithms (of course, based on mathematical and statistical methods) for extracting knowledge from raw data. In practice, data analyzing means transformation, filtering, mapping, and modeling of data in order to extract useful information and to aid decision-making. There are three traditional data analysis approaches: classical, exploratory, and Bayesian. They all deal with engineering and science problems. They gather big collections of relevant data; the difference lies in the sequence and focal point of their milestones. In the case of classical analysis, data collection is followed by modeling; on the next stage, the analysis is focused on the parameters of created model. For an exploratory analysis [17], data collection is followed directly by analysis. The goal of analysis is to arrive at a model, which would be appropriate. In the last case, that of a Bayesian analysis, data-independent distributions are imposed on the parameters of the selected model. The analysis consists of formally combining both the prior distribution on the parameters and the collected data, to draw conclusions about the model parameters. The sequences of activities shown in the table are performed in the context of these approaches. In addition, the following paragraphs provide more details of these approaches.

Table 2: Model of the data analysis process.

Data analysis	The sequence of activity
Classical	Problem → Data → Model → Analysis → Conclusions
Exploratory	Problem → Data → Analysis → Model → Conclusions
Bayesian	Problem → Data → Model → Prior distribution → Analysis → Conclusions

The classical approach is inherently quantitative and deals with mathematical models that are imposed on the data collections. Deterministic and probabilistic models which are built as a part of this approach can include regression analysis and analysis of variance, Student‘s t-test, chi-square tests, and F-tests. The outcomes of deterministic models are precisely determined through known relationships, and can be used to generate predicted values. Based on available data, probabilistic models are used to estimate the probability of a data related event or state occurring again. Research based on classical techniques is generally very sensitive, but depends on assumptions. Unfortunately, the proper assumptions may be unknown or impossible to verify.

The exploratory data analysis approach is based generally on graphical techniques, which allows the data set to suggest acceptable models (those that best match the data). For a visual assessment of the situation, histograms, scatter plots, box plots, probability plots, partial residual plots, and mean plots are created. Unfortunately, it follows that exploratory techniques may depend on subjective interpretation. Nevertheless, the advantage is that this approach makes use of all of the available data, without mapping the data into a few estimates as a classical approach does.

Bayesian data analysis is based on so-called subjective probability. It enables reasoning with hypotheses whose truth or falsity is uncertain. The difference between classical and Bayesian interpretation of probability plays an important role in practical statistics. For example, when comparing two hypotheses (models) on the same data, the theory of testing statistical hypotheses based on the classical interpretation, allows the rejection of a potentially adequate model. Bayesian techniques, in contrast, depending on the input data will give a posteriori probability chance to be adequate for each hypotheses-model. In fact, Bayesian theory allows the adaptation of existing probabilities to the newly obtained experimental data. It is useful for building intelligent information filters, such as a spam filter for email.

Besides the previous approaches for data analysis, there are a number of methods and styles tailored to the requirements of a particular sector of engineering and science. For example, biological and medical data analysis, social data analysis, business analytics and so on. Particularly well known for its extensive research is *business intelligence*. Most often, only final products fall under this concept – the software created to help managers to analyze information about the company and its environment. However, there is a wider understanding of term business intelligence as the methods and tools used for the transformation, storage, analysis, modeling, delivery and tracking of information for the period of work on tasks related to decision-making based on actual data. Business intelligence technology allows the analysis of large amounts of information, guiding the user's attention only to their effectiveness by simulating the outcome of various options for action, and tracking the results of the adoption of certain decisions. Data mining and data warehousing are typical tools in this area and are directly related to data analysis.

2.2.4 Data Mining

Nowadays, data are produced at a phenomenal rate. The continuous growth of the economy means that more and more data are generated by business transactions, scientific experiments, publishing, monitoring, etc.; more and more data is captured because of faster and cheaper data storage technologies, yet the capabilities of storages are limited. These trends lead to the phenomenon of the data flood. According to Moore's law, computer CPU speed doubles every 1.5 years, and total storage doubles twice as fast; in consequence, only a small part of gathered data will be looked at by a human. Society needs some kind of knowledge discovery in data [18] to make sense of data collected.

Data mining is the analytical process which is used to search large amounts of business or market related data (typically known as *big data*) for patterns. The final goal of data mining is the prediction or anticipation of live commercial data streams using those patterns. Data mining is associated generally with knowledge discovery in business. By uncovering hidden information, data mining performs a central role in the knowledge discovery process. The potential result of data mining is meta-information that may not be obvious when looking at raw data. It should be noted that the term data mining is a bit fuzzy in the public eye. Even experts cannot define the strict boundaries between data mining and exploratory data analysis or data-driven discovery because certain terms for similar activities are sometimes misleading. Here are several alternative names for data mining: data pattern analysis, data archeology, knowledge extraction, business intelligence, and knowledge discovery in databases (KDD).

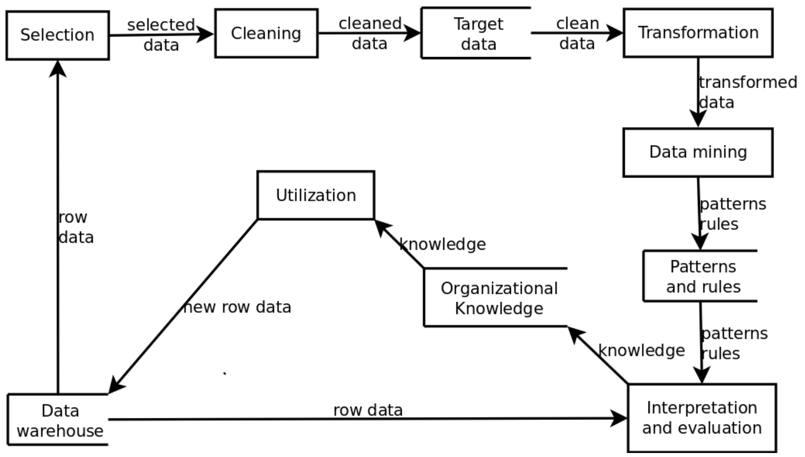

Figure 7: Knowledge discovery process (data flow diagram notation).

Here are the seven most important data mining tasks:
1. Visualize a data set to facilitate data regularity discovery by a human.
2. Clustering: finding natural explicit groups in data collection.
3. Classification: predicting an item class based on a learned method from pre-labeled instances.
4. Uncover association rules (in the form of if-then statements) that will support data relationships identification.
5. Link analysis: finding relationships between data based on uncovered earlier association rules.
6. Estimation: predicting a continuous data value based on historical records.
7. Deviation detection, this means finding irregular changes in data.

One can notice the similarity between database processing and data mining. Both activities are based on the formulation of queries to the collection of data. Data mining queries are not as well defined. In addition, data mining does not have precise-defined query languages. The result is that data mining processing cannot output the subset of a database, but only some fuzzy information e.g., about data clusters or about association rules between them. One important note, it makes sense to apply data mining techniques only for sufficiently large databases; patterns that have been found in small data collections may be accidental.

A giant amount of data like terabytes will disqualify traditional data analysis. Big data algorithms are highly scalable to handle such collections. High-dimensionality and high complexity of data are the reason to choose data mining processing before data analysis as well, especially in case of such dynamic data sources as data streams and sensor data. Contrary to expectation, many commercial and non-commercial organizations such as pharmacies, retail chains, international airports, and universities only hold large amounts of data and do nothing with it. These organizations will need data analysts specializing in data mining.

Some application areas for data mining solutions are advertising, bioinformatics, customer relationship management, database marketing, fraud detection, ecommerce, health care, investment, manufacturing, and process control. These are separate areas in engineering and science, where specialists deal with big or unstructured data. The most familiar is *text mining*.

Text Data Mining
Text data mining (also called text mining or *text analytics*) is the process of deriving information from collections of texts, e.g., from business or legal documents, newspaper or scientific articles and all kind of books. This is an area in artificial intelligence, the purpose of which is to obtain information from text sources based on machine learning and natural language processing. There is a complete similarity of goals between text data mining and data mining in approaches to information processing and applications. A difference appears only in the final methods that are used. Key groups of tasks for text data mining are text categorization, information extraction and information retrieval, processing changes in the collections of texts, as well as the development of tools for presenting information to the end user. The text is not a random collection of characters, so a text mining procedure should distinguish the words, phrases and sentences in special way to become aware of the meaning. These pieces of text are then encoded in the form of numeric variables, which are subjected to applied statistical methods and data mining in order to discover the relationship between them, and indirectly between the meanings hidden in the text.

Machine Learning
Machine learning is an area of artificial intelligence which is the study of algorithms that can learn from data. The main objective of machine learning is the practical application of such algorithms in automated systems so that they are able to know how to improve themselves by means of accumulated experience. Learning (the training of an algorithm) can be seen in this context as a concretization of the

algorithm, i.e. an optimal selection of algorithm parameters called knowledge or skill. The criterion for this selection may be increasing efficiency, productivity, uptime and reduction of the costs of an automated system. There are a few approaches to training an automated system; e.g., training on precedents provide an inductive learning based on identifying patterns in empirical data gathered by an automated system. Alternatively, deductive learning involves formalized expert knowledge in the form of rules. Machine learning technologies contribute to the field of natural language processing, stock market analysis, computer vision, brain-machine interfaces, speech and handwriting recognition, robot locomotion and others.

The Use of Data Mining

One might become familiar with the areas of application of data mining from examples of big data sources. Here are some popular web based collections:

– The European union open data portal, http://open-data.europa.eu/en/data/
– The home of the U.S. Government's open data, http://data.gov
– A federal government website managed by the U.S. Department of Health & Human Services, https://www.healthdata.gov/
– A community-curated database of well-known people, places, and things, http://www.freebase.com/
– The New York Times articles, http://developer.nytimes.com/docs
– Amazon public data sets, http://aws.amazon.com/datasets
– Google Trends, http://www.google.com/trends/explore
– DBpedia, http://wiki.dbpedia.org

Data mining means also using specialized, complex computer software; there are many commercial and non-commercial programs for data miners. Here are some examples of such tools:

Name	App Description	URL
Cubist	The tool analyzes data and generates rule-based piecewise linear models – collections of rules, each with an associated linear expression for computing a target value	https://www.rulequest.com/
Mercer's Internal Labor Market Mapping Tool	The tool constructs classification models in the form of rules, which represent knowledge about relations hidden in data	http://www.imercer.com/
Magnum Opus	The data mining software tool for association discovery finds association rules providing competitive advantage by revealing underlying interactions between factors within the data.	http://www.giwebb.com/

Name	App Description	URL
Orange	This is an open source data visualization and analysis tool, which realize data mining through visual programming or Python scripting. It has components for machine learning, add-ons for bioinformatics and text mining, and it is packed with features for data analytics.	http://orange.biolab.si/
Weka	This is a collection of machine learning algorithms for data mining tasks.	http://sourceforge.net/projects/weka/
RapidMiner	It provides an integrated environment for machine learning, data mining, text mining, predictive analytics and business analytics.	https://rapidminer.com/
Apache Mahout	This is an Apache project to produce free implementations of distributed or otherwise scalable machine learning algorithms on the Hadoop platform.	https://mahout.apache.org/
KNIME	This is a user friendly, intelligible and comprehensive open-source data integration, processing, analysis, and exploration platform.	http://www.knime.org/
SCaVis	This is an environment for scientific computation, data analysis and data visualization designed for scientists, engineers and students. The program incorporates many open-source software packages into a coherent interface using the concept of dynamic scripting.	http://jwork.org/scavis/
Rattle	It presents statistical and visual summaries of data, transforms data into forms that can be readily modeled, builds both unsupervised and supervised models from the data, presents the performance of models graphically, and scores new datasets.	https://code.google.com/p/rattle/
TANAGRA	This is free data mining software for academic and research purposes. It proposes several data mining methods from exploratory data analysis, statistical learning, machine learning and databases area.	http://eric.univ-lyon2.fr/~ricco/tanagra/

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.

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.

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.

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

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*

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, 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.

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

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

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 σ be a partial mapping of variables in the subject domain; then the value of the variable x in the state σ can be written as σx. State σ 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 σ 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 $e \sim o(e_1,\ldots,e_n)$ (o is a operation and $\sigma(e_i) = v_i$ is a program variable) then $\sigma(e) = o(v_1,\ldots,v_n)$.

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 τ 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 σ 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 $\sigma \mid= T$ 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 $\sigma \mid= T$ 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 $i = 0,\ldots,n\text{-}1$ and $\sigma^{i+1} = P_1(\sigma^i);$
c. $\sigma^i \mid= T$ 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 $\supseteq$ is the sign of the superset. One can read this expression: "Domain of σ 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".

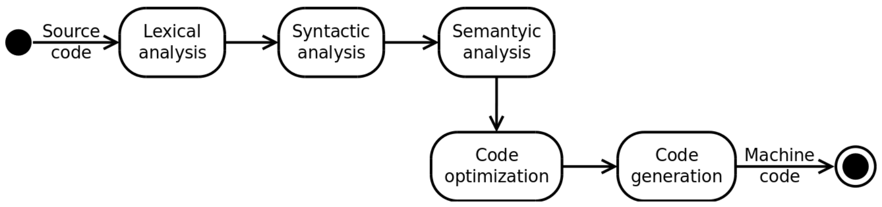

Figure 8: The compile process.

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;

- 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 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 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.

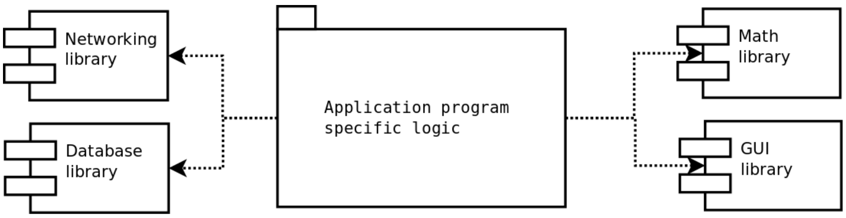

Figure 9: The library architecture of a program.

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.

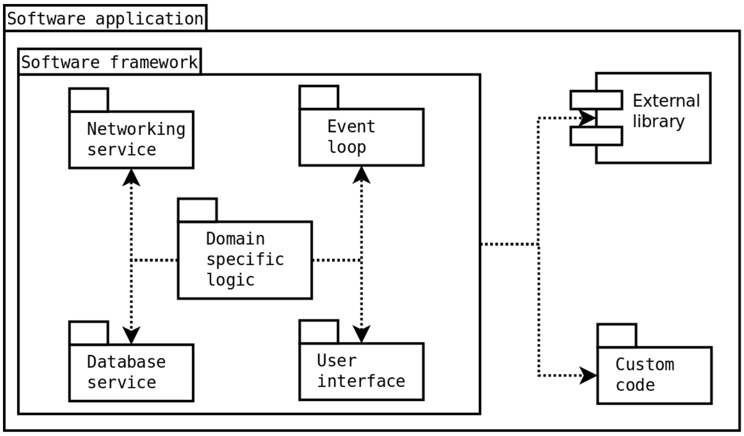

Figure 10: The framework architecture of a program.

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.

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.

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.

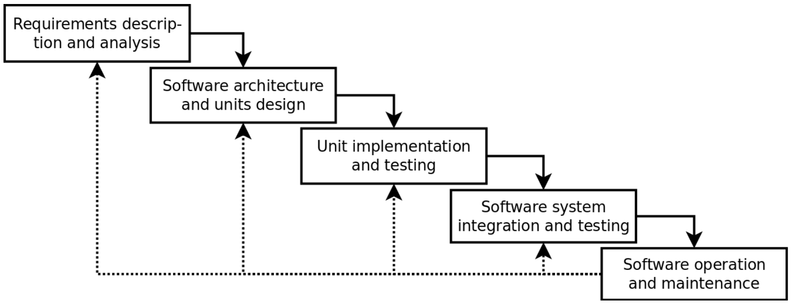
Requirements description and analysis
Software architecture and units design
Unit implementation and testing
Software system integration and testing
Software operation and maintenance

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.

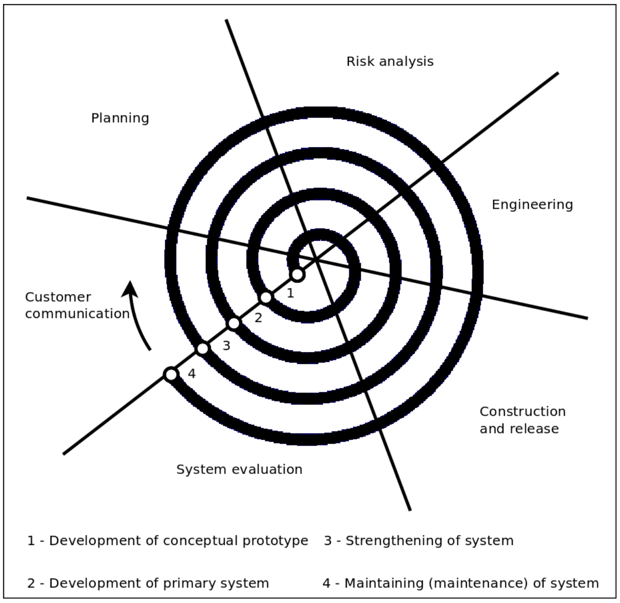
Risk analysis
Planning
Engineering
Customer communication
1
2
3
4
Construction and release
System evaluation
1 - Development of conceptual prototype
3 - Strengthening of system
2 - Development of primary system
4 - Maintaining (maintenance) of system

Figure 12: The spiral model of software development process.

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".

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.

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 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.

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.

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).

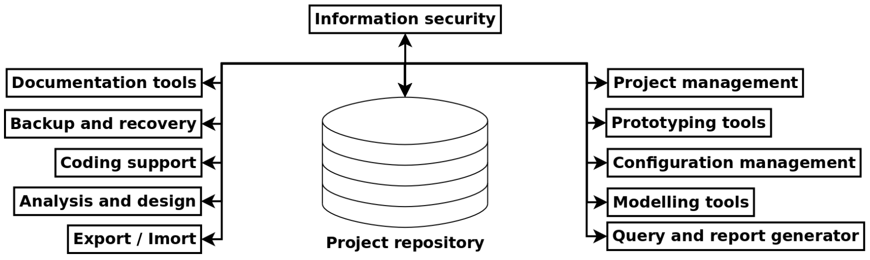

Figure 13: The structure of a typical CASE environment.

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 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].

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.

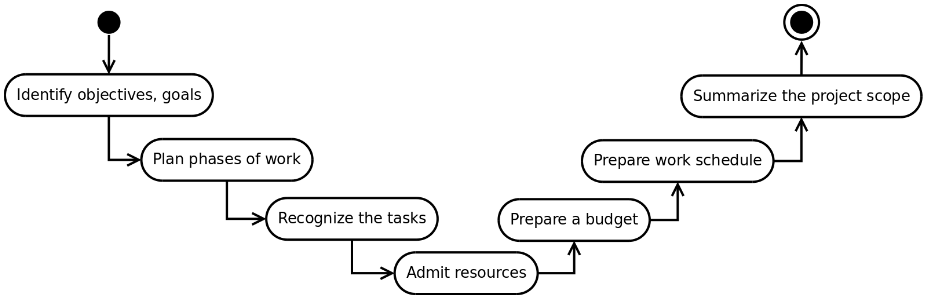

Figure 14: Defining the scope of IT project.

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).

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.

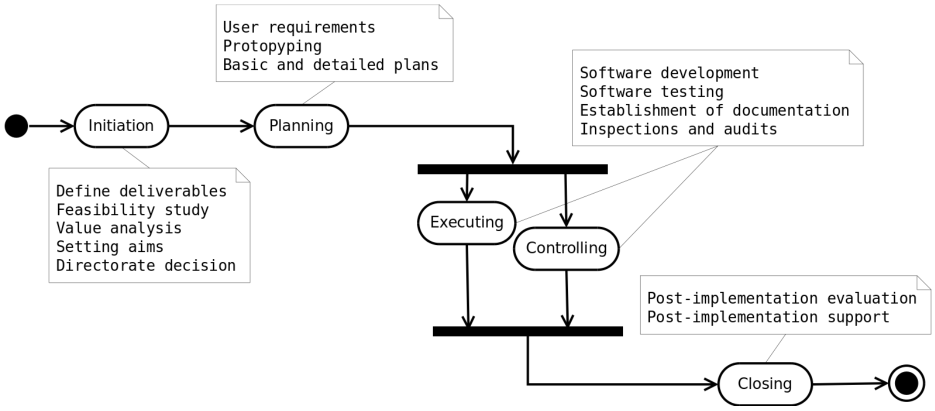

Figure 15: Crucial stages in an IT project life cycle.

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 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.

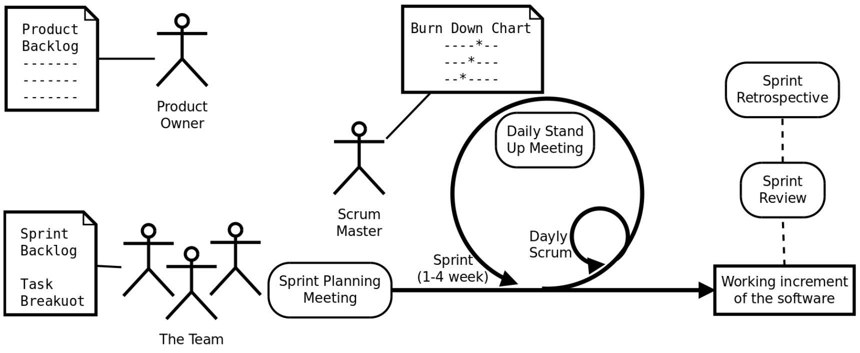

Figure 16: Scrum process (simplified view).

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].

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.

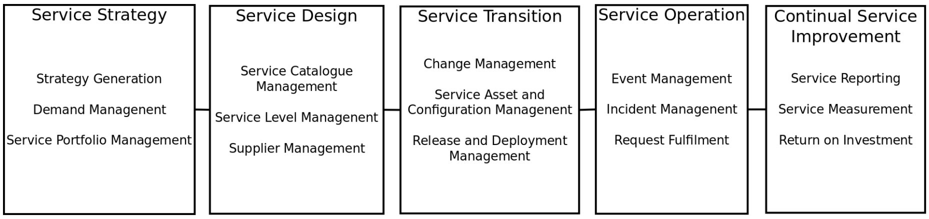

Figure 17: ITIL Framework (simplified view).

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 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.

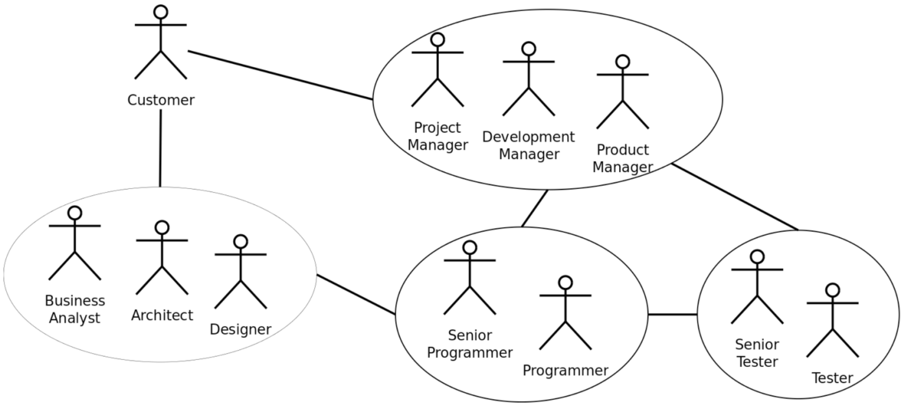

Figure 18: IT team organizational chart [42].

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 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.

4 Information Systems

4.1 Information Systems Basic

An *organization*, such as a company, firm, corporation, institution, agency, is a system focused on the achievement of specific objectives and business goals. Each organization has official structure, employees, material and technical equipment, and management. Deliberately inflicting the functioning of the organization is not possible without extensive internal and external information flows. The *information system* of the organization is a deliberate juxtaposition of elements, e.g. people, documents, data, processes, methods of communications, network infrastructure, and computer equipment, that have to work together to enable the daily functioning of the organization. Information systems (IS) include all the *information flows* within the organization, as well as all elements related to the processing and transfer of information, namely: data sources, methods of transmission, collection points and transformation processes. An information system is a subsystem in relation to the organization. Information systems of modern organizations (especially business and industrial) make intensive use of information technologies. The technical bases of such information systems are computer hardware and software, network infrastructure and telecommunication channels.

Information flows are not evenly spread throughout the structure of the organization. The most intensive exchange of information takes place in the management subsystem, which covers a large set of informative and communicative activities, e.g. analyzing, planning, organizing, controlling, communicating, motivating, and so on. The management subsystem is a set of actions and measures for finding, collecting, storage, transmission and processing of information in order to be able to make decisions about managing the organization. Because of the importance of the processed *managerial information*, IS imposes certain requirements on the quality it has; managerial information should be:

- Honesty and reliable, information must accurately describe the organization's processes and economic conditions.
- Selective, information have to be selected in the context of the economic problem.
- Addressable, the method of delivery and presentation of information have to be consistent with the requirements of each individual manager.
- Aptness and relevance, distributed information has to be consistent with the specific managerial demand.
- Topicality and timeliness, information has to be available on request.

The engineering of computerized information systems consists of *information engineering*, *knowledge engineering* and software engineering. It focuses primarily on

the data that are stored and maintained by computers, and the information that is derived from these data. The main purposes of engineering information are:
- Important organizational data have to be located in the processing center by providing repositories or databases.
- For these datasets, there is a logical way to present and interpret them.
- Types and structures of the data used in the organization should not vary; only little changes (evolution) over time may be acceptable.
- In a stable organization, data values are relatively constant, but the processes that use these data may vary freely.

The main task of the designer of information systems is to provide a subset of *organizational knowledge* by means of known information. For knowledge representation in information systems, technologies of knowledge engineering and *knowledge bases* are used. Knowledge engineering is a discipline dedicated to creating effective systems that represent knowledge (methods of mapping reality) in expert systems and decision support systems.

The *primary data* collected in a computer's information system represent the *raw facts* about the organization and its activities, e.g. about financial and non-financial transactions. Deliberately structured and filtered data, having a specific meaning, can deliver new information, e.g. about trends. When this information is practical and useful, we can interpret it (e.g. for making decision). Such interpretation is seen as a phenomenon of obtaining knowledge. In conclusion, we can say that a computerized information system has to processes data into useful information and can generate knowledge (in other words extract it from the data). Such systems are sometimes called intelligent information systems or more precisely *intelligent decision support system*.

4.1.1 Organized Collection of Data (Databases)

To create a computerized information system, it is essential to have the hardware and software to support the accumulation and handling of large data collections belonging to the organization. A core technology needed to create such data repositories is database technology. It is useful not only for data storing, but also for data analysis and processing, for information management, and for data visualization and presentation in practically all business and industrial contexts. A typical database presents its own data to the user as an informative model of a fragment of reality, for example, the user can see an airport database as an official online flight timetable of arrivals and departures. The database is not a standalone self-sufficient application, it must be created and administrated (managed) by using the *database management system* (DBMS); alternatively, it can be provided through other software that has the ability to communicate with its data set through specific interfaces. For example, a standard database access method called *Open DataBase Connectivity* (ODBC) makes it possible to access data from any application, regardless of which DBMS it is handling. It is important that databases are networking products; this means that information systems computers and database computers are usually located in different places. User terminals can also connect to the information system from any desired location. Network operation is the basic style of exploitation of the IS.

Before creating a database for the information system, the designer builds a logical (conceptual) model of data, which will be stored in the database. There are many ways to compose such a model; the best recognized is *entity-relationship diagramming*. In such a text and graphic model the entity is something existing in the real world, regarding which data will be stored in the designed database. On the model of an entity, DBMS will generate tables, which will be written copies (records) of data items. An entity may be a physical object, e.g. a particular employee, building, or car and it may be a conceptual object, e.g. a job, a bank, or an automobile manufacturer. Each entity can be described by some properties, which have to be presented in an entity relationship diagram (ER diagram) as attributes of the entity. In the above example, the car is the entity and its model, doors number, color, transmission, number of seats, weight, and release date are the attributes. Each entity must have an attribute, which allows it to be uniquely identified. Such an attribute is called the primary key. In the above example, the social security number may be the primary key of an employee, and the vehicle registration number may by the primary key of a car. A relationship is the thing which allows the description of a substantive connection between different entities. In the above example, there is a relationship between the car and automobile manufacture. There may be a relationship between a bank and a building. The connection of entities is characterized by a multiplicity of relationship, e.g. one-to-one, one-to-many. The relationship may have attributes as well. An example of an ER diagram is shown in Figure 19.

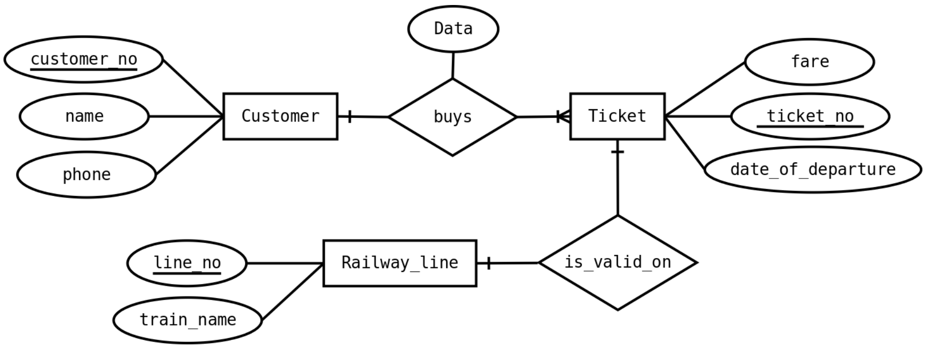

Figure 19: ER diagram illustrating the logical structure of database.

Based on the entity relationship diagram, the database administrator generates a set of related tables – it will be a database skeleton. After this, somebody will have to fill it in with real data. The presented data collection modeling style is closely associated with relational databases, the idea of which was proposed by E. F. Codd in 1970. Nowadays, informaticians have non-relational databases technologies, e.g. based on the concept of a document. It is noteworthy that older, hierarchical databases from mainframe era are still in use. For example, the MS Registry is a hierarchical database that provides data which are critical for Windows OS. Also, IBM Information Management System is the widely used commercial hierarchical database management system.

The database specialist has at his/her disposal many database management systems, which generally do not interact directly with one another, mostly for historical reasons. The *structured query language* (SQL) is a thing that unites all popular relational databases. It is a quaint original data definition and manipulation language, which is widely used for communication between the information system and the data collection. The standard SQL commands such as *Create*, *Select*, *Insert*, *Delete*, and *Update*, can be used by programmers to achieve everything that they needs to do with data. For example, to create a new table describing the car, a programmer can write the following SQL code:

```
CREATE TABLE CAR
(REGISTRATION_NUMBER CHAR(10) PRIMARY KEY,
DOORS_NUMBER SHORT,
SEATS_NUMBER SHORT,
COLOR CHAR(10),
RELEASE_DATE DATA);
```

The next fragment of SQL code may be used to insert some records into the new created table:

```
INSERT INTO CAR VALUES (‘EBX 5525’, 3, 5, ‘BLACK’, 2011);
INSERT INTO CAR VALUES (‘NBH 7789, 5, 7, ‘WHITE’, 2014);
INSERT INTO CAR VALUES (‘NDM 7879’, 4, 5, ‘YELLOW’, 2015);
```

After the table has been filled in, it is possible to select some data from it:

```
SELECT * FROM CAR
WHERE RELEASE_DATE > 2012;
```

In the above example, the database will send in answer on such a SELECT request the list of data concerning cars that were manufactured after 2012. It is worth noting that the SQL code is sufficiently clear and can be read with understanding by a non-computer specialist. This is one of the most important advantages of the SQL language, because the specialists in the subject (non-informaticians) should shape the substantive question about the organization’s data collections.

4.1.2 Discovering IS Functionality

To design an information system, informaticians need to understand the organizational structure of the data and the processes by which these data are generated, transformed and presented. A business or industrial organization is a *complex system*. It is not a simple task to fully comprehend the structure of the data circulating in an organization, which should be represented in the computerized information system. Every specialist working in the organization understands the functioning of the organization from his or her own perspective. An objective, impartial, and multilateral representation of the organization should be developed through targeted analytical activity. One classic way to do this is called *data flow diagramming*.

In information systems theory, the data flow [43] is the term represented by the path taken by data items within an organization moving between data stores, processes (operations), and outside entities, i.e. people, or external systems. Data flows are shown on diagram by a line with an arrowhead to indicate the direction of the flow; each data flow has a unique name indicating what data are being passed. Data flows and processes can be modeled at different levels of detail. The highest level (number 0) is called a context, because it shows the information system as a single process. By this convention, level 1 (as is shown in Figure 20) is designed to review the main functions of an information system, and levels 2 and above - for detailed analysis.

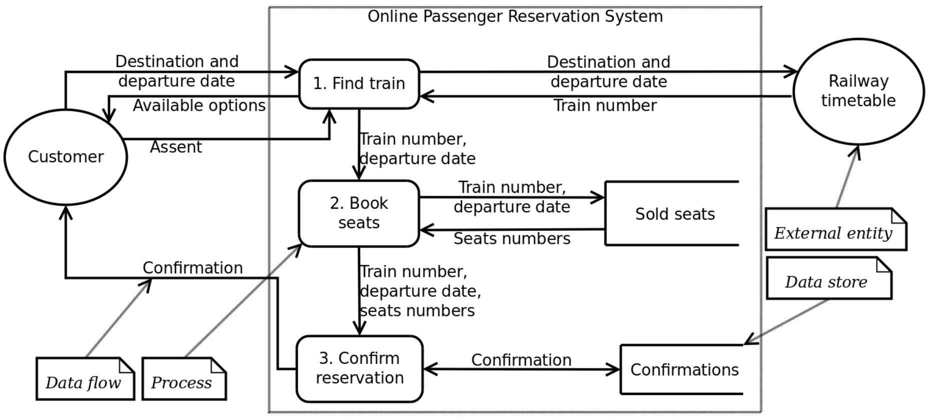

Figure 20: Data flow diagram for a reservation system of a railroad company (level 1).

With a detailed model of the flows of data, developers can proceed directly to the technical design and coding of system modules. In very simplistic terms, each process must have its own application window and the connected data flows will show which values should be entered in a dialog box, and which results of the calculations are to be expected. Both data flow diagramming and the entity relational model are the main techniques of structured systems analysis and design method [44] (SSADM), which is associated with the waterfall model of the software development process. Initially, SSADM was released inside a *systems approach* [45] to the analysis and design of information systems.

4.2 Analysis and Modeling of Information Systems

The complexity of information system expresses the degree to which its components engage in organized structured interactions. Business systems, such as system of air traffic control at an airport, the banking system, or a production management system achieve the very high level of complexity. The increase in the number of components and in the number of relationships between parts of a system gives an enormous rise in the complexity of the collective behavior of a system. Prior to the design and development of a complex system, it is obligatory to carry out modeling and deep analysis of the future system, together with its possible context (environment). It should be emphasized that even sophisticated modeling does not easily capture the complexity of real business and industrial information systems. A human has a limited capacity to anticipate the behavior of relatively simple systems and cannot predict the behavior of complex systems at all. However, *complexity theory* [46] believes that complex phenomena can allow pattern predictions through modeling.

Models are used in software system engineering because of the impossibility or impracticality of creating experimental conditions in which a developer can directly examine outcomes. The generated model as a *conceptual representation* of a system will typically refer only to some aspects of particular system. This means that the systems analyst should generate several models to embrace the system from different perspectives. Methodical and planned information systems analysis is conducted currently using *information models* [47]. Nowadays, the unified information modeling language (UML [48]) released by the Object Management Group [49] in 1997 is preferred.

4.2.1 Brief Introduction to the Unified Modeling Language

Object-oriented modeling languages began to appear in mid-1970 as various methodologists experimented with different approaches to object-oriented analysis and design. This was quite long and “painful” process until it converges to UML. The main authors of UML were Grady Booch, Ivar Jacobson, and Jim Rumbaugh. They were the first to introduce in 1996 a unified, standard, text-and-graphical modeling notation for IT professionals to interchange blueprints of software system architecture and design plans.

From the outset, UML was independent of programming languages and development processes. This facilitates communication between IT communities that work with essentially different development platforms and hardware and operating systems. UML has integrated the best practices of all object-oriented design of software. It is important that UML supports high-level design concepts such as patterns, frameworks, and collaborations. UML modeling is used for specifying, visualizing, constructing, and documenting software artifacts. UML provides [50] model elements such as concepts and semantics, and the notation, by which a visual rendering of model elements takes place. The major UML diagrams elements (as is shown in figure 21), through which designer expresses the structure and behavior of the information system, are described below:

– Two-dimensional graphical objects (some of them are partitioned) with changeable size, which are containers for text, icons, and other graphical objects;
– Path and area symbols (e.g. time line segments with attached endpoints or rectangular system boundaries); they may also have icons and text qualifiers;
– Text objects, especially names (uniquely identifies some model element), keywords (text enclosed within Latin quotation marks «...» to express some concept), expressions (a linguistic formulas), and labels.

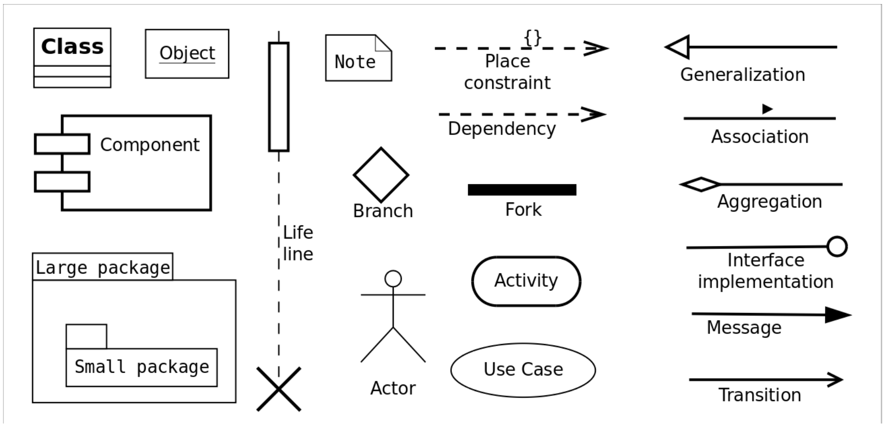

Figure 21: Some UML model elements.

A typical UML model is a set of diagrams; each UML diagram is an insight (view) into the model of a system from the viewpoint of a particular stakeholder. The diagram provides a partial, incomplete representation of the system and has to be consistent semantically with other views. In the UML, there are nine standard diagrams, which can represent static and dynamic views of an information system; use case, class, object, component, and deployment diagrams constitute the static set, and sequence, collaboration, statechart, and activity diagrams – the dynamic set. Below are shown some cumulative information about more popular UML diagrams.

Use Case Diagrams
These diagrams mould an image of interactions of actors with system components. Here, an actor is an external entity that cooperates with a software system to achieve its own goals. Formally, a use case diagram is the graph of actors and use cases enclosed by a system boundary and focused on what a particular actor is doing in a particular action, as it is shown in Figure 22. The main idea of the use-case diagram is to visualize the functional requirements of a system. Typically, only high-level functions of the system are shown and non-functional requirements are not presented in this view. In practice, use case diagrams are used to represent the functionality of system from a top-down perspective. Further particularities are added later to make clear the details of the system's behavior. Use cases are graphically shown as ellipses.

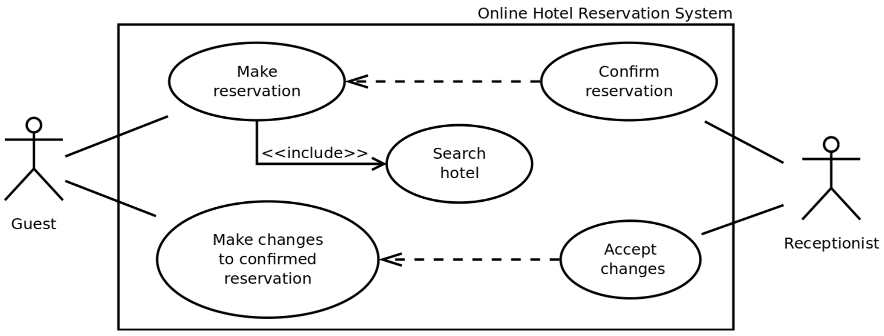

Figure 22: Example of a use case diagram for N system.

Class Diagrams
The class diagram is the graph of entities together with the relationships between them. In fact, the diagram shows the static structures of the information system. Often, class modeling is made at a level of subsystems or system modules, because modeling of a complete system may be impractical due to the huge number of classes. The main concepts of this view are class, association, generalization, dependency, realization, and interface. Here, a class is an internal entity of a software system, which describes the set of objects that share the same attributes, operations, relationships, and semantics. Classes are graphically shown as partitioned boxes. At first, a conceptual class diagram is created to display only the logical structure of system (as is shown in Figure 23). During this first stage, the analyst and designer can choose an optimal set

of domain classes to fulfill the user requirements. Then, the expanded form of class diagram is created to display the physical structure of system. During this second stage, the designer adds to diagram special classes to fulfill the implementation platform requirements. Programmers typically deal with a physical class diagram. A conceptual class diagram has to be approved by a client representative.

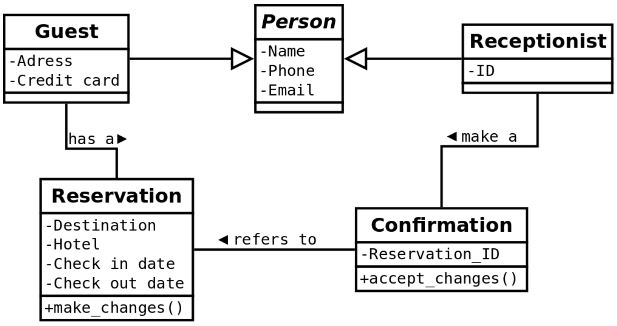

Figure 23: An example of a conceptual class diagram for N system.

Sequence diagram

A sequence diagram links use case and class diagrams. It shows a thorough information flow between objects for a specific use case. In other words, it shows the sequence of calls between the different objects modeling the interactions of these objects during the realization of the use case. The sequence of calls and messages is shown on the sequence diagram in the time order that they occur. The interaction starts always near the top of the sequence diagram and ends at the bottom. Sequence diagrams are usually used to model and examine usage scenarios, the logic of methods, and the logic of services (high-level methods invoked by a wide variety of clients). An example of a sequence diagram is shown in Figure 24.

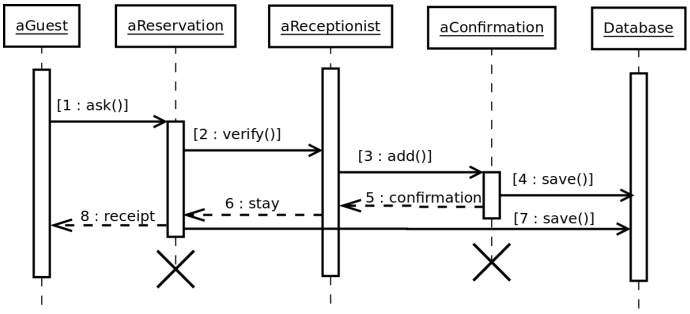

Figure 24: An example of a sequence diagram for N system.

Activity diagram
An activity diagram represents one of most popular UML views. It is a kind of flow chart and shows in a procedural style the flow of control between class objects while processing an activity. The activity diagram can express not only sequential, but also branched or concurrent flows. This diagram can be used at a high level of generalization to model business processes realized by means of a software system, or at a low level to model an internal component's or class's actions. Activity diagrams are useful for examining the logic captured by a single use case or a usage scenario by using both forward and reverse engineering techniques. The relatively simple notation makes the activity diagram a useful tool in customer relations, especially in the context of the negotiation of the business aspects of software systems. An example of an activity diagram is shown in Figure 25.

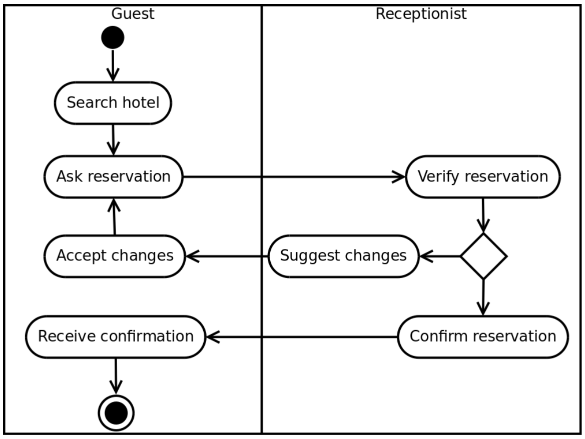

Figure 25: An example of an activity diagram for N system.

Component diagram
A component diagram presents a physical view of the software system. It is a map of software components together with their mutual relations (dependencies) within the system. Here the component is a software element (a structured class) or a module, which may be designed independently (separately of system project) and can be deployed independently. An activity diagram can be used both at a high level of information system modules (logical components) and at a low level of platform dependent component packages (physical components), e.g. containers in .NET or packages in Java technology. Recently, component-based software engineering has gained a great deal of popularity. Components are not only the basis for .NET and Java platforms. For example, Linux specialists intensively use components in KDE and GNOME projects. Nowadays, larger pieces of a system's functionality may be

assembled by reusing (ready-to-use) components. It is a very popular manipulation in the production of computer game software.

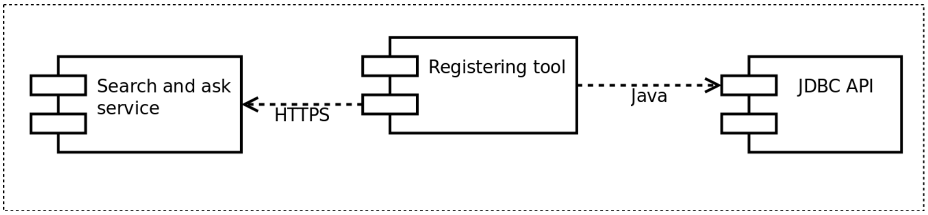

Figure 26: An example of a deployment diagram for N system.

4.2.2 Modeling the Architecture of Information System

The integration of all system components and control of the system interfaces is made by system engineers in accordance with the approved architecture of system. At the analysis and modeling stage this activity should be prepared by the introduction of some high-level logical constructs. These architectural constructs require a lot of mental effort at a high level of abstraction. This is why software architects tend to reuse successful architectural decisions, e.g. via the architectural frameworks [51]. Some examples of such general-purpose architecture frameworks are TOGAF, MODAF, RM-ODP, and Kruchten's 4+1 View Model. The organization domain has its own enterprise architecture frameworks, e.g. government frameworks, consortia-developed frameworks, and defense industry frameworks. The area of software architecture for information system is rich in detailed information. In getting acquainted with this area of expertise, one can familiarize oneself with the idea of architectural views.

It makes sense to classify UML diagrams against the 4 + 1 Architectural View Model, which was introduced by Philippe Kruchten [52] for describing software-intensive systems. This model is commonly used to describe information systems from the viewpoint of different stakeholders, e.g. end-users, development team members, and project managers. The model provides four fundamental views – logical, implementation or development, process and deployment views. Furthermore, it provides usage scenarios or use cases which serve to describe architectural elements (system objects and processes) and to illustrate and validate the architecture design. Logical and process views deal with conceptual architecture models, when implementation and deployment views are known as physical architecture models. Here are some very short, simplified characteristics of these five views with notes about related UML diagrams:

- The first, logical view presents end-user functionality. The logical architecture of information system has to support the functional requirements specified under

the project. The most important UML model for this view is the class diagram, a supporting role relative to its play sequence and communication diagrams.
- The second, implementation view presents the developer's perspective on the software system; this view is intended for software (code) management. In this architecture, system components are described by means of UML component and package diagrams.
- The third, process view is dedicated to system integrators. Such information system parameters as performance, scalability, concurrency, throughput, and others are modeled and analyzed in this view. The UML activity diagram is used to model this perspective, i.e. dynamic aspects of the information system – computational processes and threads complete with communication between them.
- The fourth, deployment view is intended for system engineering. System engineers deal with the topology of software components, with the delivery and installation of modules on the physical layer. They manage physical communication between these components as well. Most often the UML deployment diagram is used to analyze this perspective.
- The separate, fifth view utilizes scenarios or more general use cases view and integrates the four previous views. The usage scenarios have to represent in the architecture the most important requirements, which are presented by means of object scenario diagrams and object interaction diagrams.

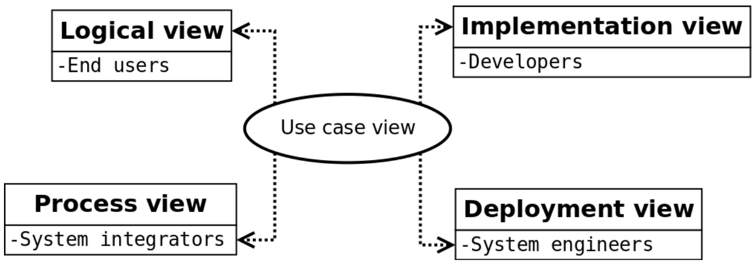

Figure 27: The 4+1 Architectural View Model.

An architecture-centered approach has been in the last several years the standard approach to enterprise system development and integration, especially for complex and critical enterprise systems. Please note that the field of enterprise architecture is still evolving. Nowadays, it seems that enterprise architecture is a path, not a destination.

4.3 Design of Business Information Systems

Business information systems [53] are the combination of people, information technology, and business processes to successfully complete business objectives. Nowadays, most professions make use of business information systems. They provide excellent conditions for information activities not only in manufacturing, marketing, trading, accounting, and finance, but also in science, healthcare, education, and even in entertainment. One of the most important factors of successful information systems is good design. It is necessary that such a system not only has excellent technical design, but also analytical design (i.e. optimal adaptation to the specificity of the profession), graphic design (i.e. visual perfection), and ergonomic design (i.e. high usability of software). Present-day businesses are driven by the information they offer using information technologies and particularly the web. Professional information systems support the processes of enterprise information exchange, so their perfection largely determines the success of a business.

Each information system is planned to improve the business in someway. Before making such an improvement, it is very important to recognize the current business situation, especially from the perspective of business processes. Business analysts, and with them managers, system engineers and developers use business process modeling for this purpose, i.e. some technique to diagram business processes. The visualizing (diagramming) of the current business process as-is helps in understanding of a cause of a failure and possible paths for development by stakeholders. After studying the business process model and before developing the renovated information system, business analysts have to diagram a future process. In the given context, a business process should be understood as a flow of activities. Each of such activities have to represent the work of a person or an internal system, so one can think about the progress of work or about the workflow. In conclusion, the attributes of the business process are:

- Task – the least of process elements, a logically separated portion of the work that has been assigned or done as part of the participants duties;
- Activity – a task or set of tasks carried out in order to produce a deliverable;
- Participants – people or systems that give input to, or perform tasks within a process;
- Roles – participant's positions, an abstraction that links participants and performed tasks or activities;
- Flow control or sequence to cause a process – the logical order in which tasks are performed;
- Events (or other words triggers) – occurrences and appearances that direct a process sequence, inter alia, cause a process to start or to end.

A logically complete hierarchy associated with business process modeling may be presented as follows: cycle, process, subprocess, activity, and task. The introduction of an intermediate level of a subprocess is needed to compartmentalize and focus on specific process segments.

4.3.1 Business Process Modeling Notation

The first standard *business process-modeling notation* (BPMN 1.0) was released to the public in 2004 [54]. From the very beginning, this notation was created for a broad audience, i.e. for business users, business analysts, technical developers, and business people managing real business processes. In fact, BPMN defines a business process diagram, which is based on a well-known flowcharting technique modified for creating better graphical representations of business process operations (activities). In this notation, the business process is a flow of tasks and activities, which are performed by the process participants in roles. These tasks and activities are organized in specified chains directed by events and measured in time. Therefore, a business process model is a representation of a set of tasks and activities where some pairs of them are connected by links. This is consistent with the definition of graph in the mathematical meaning, i.e. tasks and activities may be treated as objects or the graph's vertices.

A basic set of BPMN symbols (as is shown in Figure 28) includes only three flow objects – *event*, *activity*, and *gateway*. An event symbol (circle) indicates an effect on the flow of the process. This effect usually has a cause (trigger) and an impact (result). There are three types of event symbols, based on when they affect the flow: start, intermediate, and end event symbols. An activity symbol (rounded rectangle) is a representation of work that is performed by a business (e.g. by company, a company department, or a particular worker). There are two types of BPMN activity symbols to distinguish the divisible or compound (indivisible) character of real business activities; the types of activities are task and subprocess. A gateway symbol (diamond) marks traditional business decisions, as well as the forking and merging of sequence flows. The type of gateway depends on the diamonds shape's internal marker, which indicates the logical operation on the connected flows.

Besides flow objects, BPMN defines connectors, which create the skeletal structure of the business process. There are three *connectors*: sequence flow, message flow, and association. A sequence flow show the order that activities and tasks will be performed. A message flow shows the messages between participants, and finally an association shows data associated with flow objects. In addition to this, BPMN supports the *swimlane* concept as an organizational entity, which is realized by pool and lane symbols. The pool presents a self-contained sequence of activities, i.e. individual process. It is important that only message flows may cross its boundary. The lane is a part of a pool and, in practice, is used to show who is responsible for specific work or to which group activities belong a specific business function.

BPMN version 2.0 has been available since 2011 [55]. This defines the notation, metamodel, interchange format, and includes some essentially new features. For example, it enables the provision of XML schemes for model transformation and in such a way, that orient BPMN toward business modeling and executive decision support. Some very new concepts were introduced, e.g. a choreography, which represents an expected behavior between two or more participants of a business process.

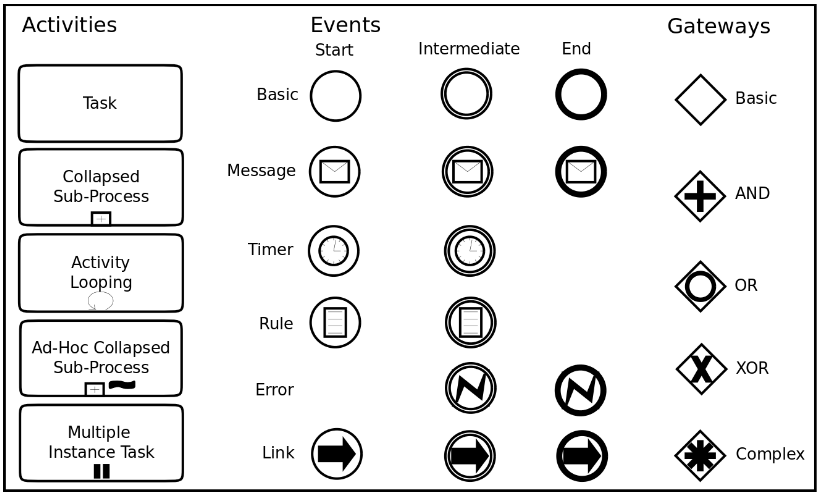

Figure 28: A basic set of BPMN symbols.

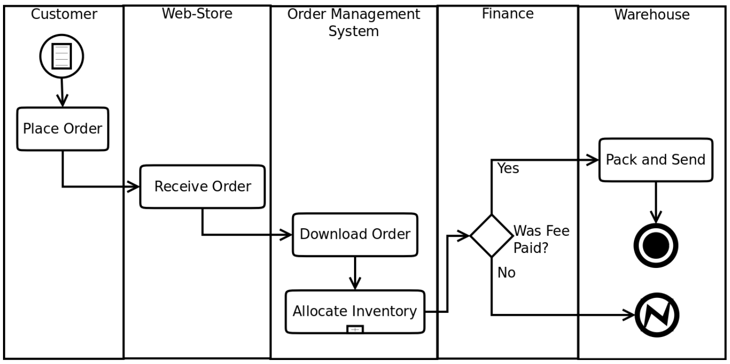

Figure 29: An example of simple business process model.

An example of a business process model is shown in Figure 29. To create such a model one needs a software tool for business process modeling. The above model has been developed in a DIA environment, which is good for educational purposes, but not for real projects of information systems. On the software market, there are many proposals that implement the full range of BPMN 2.0 (Business Process Model and Notation). Bizagi BPM Suite [56] seems to be the most progressive.

4.3.2 Business Process Reengineering

Business process reengineering [57] (BPR) is the idea of business modernization through redesigning the company processes and the synchronous deployment of information technology. In practice, this means discovering and eliminating wasted or redundant effort and improving processes efficiency [58]. This is done in parallel with the implementation of new software, which will be responsible for maintaining an optimum discipline for the completion of tasks and replace workers in such jobs that a computer can perform better. The business processes which could be redesigned in such a way, cover everything from production to sales, to customer service. It is significant that during the reengineering revolutionary change is expected, not evolutionary improvement. Business process reengineering is particularly important for companies to achieve maximum improvements in their critical areas (e.g. high quality of service or low costs of production) through the use of information technology. This entails the need to rethink not only structure of business processes, but also management style, organizational structure, job definitions, workflow, and so on.

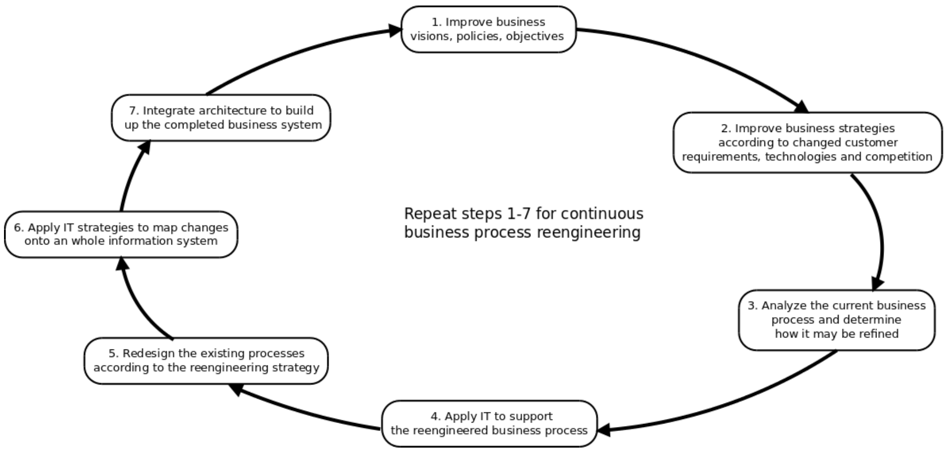
1. Improve business visions, policies, objectives
2. Improve business strategies according to changed customer requirements, technologies and competition
3. Analyze the current business process and determine how it may be refined
4. Apply IT to support the reengineered business process
5. Redesign the existing processes according to the reengineering strategy
6. Apply IT strategies to map changes onto an whole information system
7. Integrate architecture to build up the completed business system
Repeat steps 1-7 for continuous business process reengineering

Figure 30: Business process reengineering cycle.

In accordance with a traditional, structural approach to business engineering, the structure of business and the plan of processes were directly based on business strategy. IT implementation was planned as a last stage of the establishment of the company. Business process reengineering has put forward a process-oriented approach, which has to eliminate some problems typical for business hierarchical structures. Namely, BPR changes the hierarchical relationships between management and employers into an interactive process, which is readily computerized because it follows precise, well-defined procedures. At present, information technology is considered to be the most important factor in the development of new forms of collaboration within an organization and between organizations as well. Among these technologies the following should be mentioned: telecommunication networks, wireless data communication, mobile computing, cloud computing, voice over internet protocol (VoIP), tracking systems, and shared databases.

A big improvement in business processes arose in the 1990s based on the idea of a *workflow management system*, which was positioned as a system that defines, creates and manages the execution of workflows using software [59]. The core of such software is a workflow engine. It is able to understand process definition, act together with workflow participants and invokes the use of necessary external software tools and applications. In figure 31 a workflow reference model is shown, which representents the architecture of a workflow management system identifying system interfaces. It is worth recalling that workflow is defined as a sequence of activities through which a piece of work is completed.

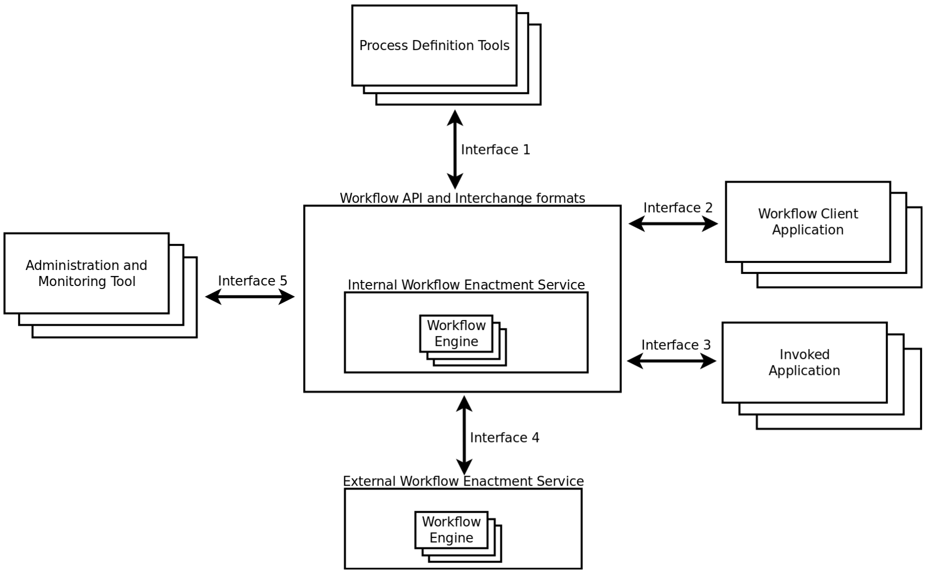
Process Definition Tools
Interface 1
Workflow API and Interchange formats
Interface 2
Workflow Client Application
Administration and Monitoring Tool
Interface 5
Internal Workflow Enactment Service
Workflow Engine
Interface 3
Invoked Application
Interface 4
External Workflow Enactment Service
Workflow Engine

Figure 31: The workflow reference model.

In the same period, another idea for business process design and improvement was popularized – enterprise resource planning (ERP). It is defined as a suite of integrated business applications; here, software integration is the key. Generally, ERP modules carry out common business functions such as information lookups, accounting, banking, inventory control, payroll and taxes, order processing, workflow approvals, or customer care. A common set of data and a synchronization mechanism helps in integrating these applications for decision making and planning. Nowadays, enterprise resource planning is an established category of business management software. Some typical modules of ERP are shown in Figure 32.

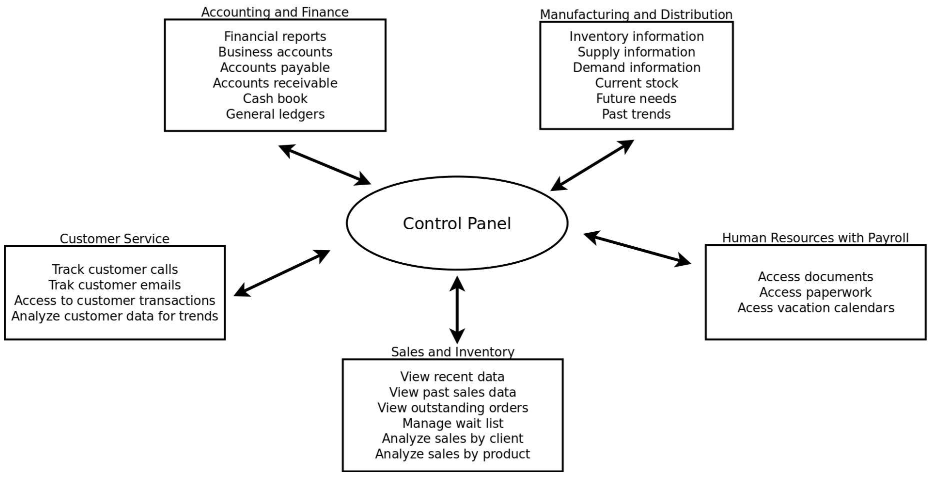

Figure 32: Typical ERP modules.

One of the most important factors in auspicious business process reengineering is considering the right IT infrastructure [60].

The IT infrastructure of organization

The IT infrastructure of organization is planned together with applications portfolios of the organization and may not be changed in a random manner. Here are some IT planning approaches:

– Business-led approach, in which the business strategy defines the IT investment plan;
– Organizational approach, in which the IT investment plan is a compromise between stakeholders;
– Administrative approach, in which a steering committee establishes the IT investment plan;
– Method-driven approach, in which the IS needs are identified using special techniques.

Generally, IT planning consists of four stages: the strategic IT plan, an analysis of information requirements, an allocation of resources, and a project plan. It all is carried out in accordance with the strategy of the company and is intended to ensure the competitive advantage of organization. The main result of the planning is the set up an applications portfolio, through which an organization intends to manage its business. The overall structure of information systems in an organization is called *information technology architecture*. Nowadays, the IT architecture of the organization is designed around business processes whereas departmental hierarchy is not relevant in this context. An example of IT architectural decisions may be the choice between centralized, distributed, or blended computing. Another example would be the choice of the network working group configuration.

4.3.3 Software Measurement

The design of business information systems is not possible without a measure of software, i.e. without expressing software in impartial numbers. The purpose of *software measurement* [61] may be related to the prediction of software complexity or usage, to the control of production hours or costs, and to the assessment of software usability, quality, or security. Software measurement objectives may concern the assessment of the status of IT projects, products, processes, or resources, but also trend identification or self-assurance action. For stakeholders, it is important to know the planned and actual size of a software system, but also the numerical indices of software quality and of the progress of a software project.

Knowing the size of software is important for comparing different systems together, e.g. to evaluate the effort of development teams or to estimate the cost of development work. The oldest approach to measure the size of software is to count the lines of source code [62]. This size metric is called *lines of code* (LOC); in which physical or logical lines of code are distinguished. For larger systems, thousands of lines of code (KLOC) are also used. Modern software systems reach the size of tens of millions of lines of code. As practice shows, the same system developed with different programming languages will have different LOC results, so to compare systems written in different languages, we must take into account the possibility of a language to write concise code. Unfortunately, in order to calculate LOC we have to wait until the system is entirely implemented. This disqualifies the LOC-metric from forecasting IT projects, e.g. prediction of cost and effort of new software development.

A different approach to measure the size of software is based on *function point analysis* [63]. Allan Albrecht at IBM published it in 1979. Function points do not calculate the size of software directly. They measure the functionality offered by a software system. The trick here is that on the one hand, functions represent sets of executable statements (source code logical fragments) that perform certain tasks, but on the other hand, they are visible from the user point of view as services. This allows for a calculation of the function points before a system is developed. It is enough to have conceptual and architectural design. To be language independent, the function points are calculated as a weighted sum of five components that define the application functionality provided to the user. The components are broken into two categories:

– Data functions (reflect data requirements);
  o Internal Logical Files (files maintained by an application);
  o External Interface Files (files accessed by an application but not maintained by it);
– Transactional functions (reflect needed data processing);
  o External Inputs (e.g. forms or questionnaires);
  o External Outputs (e.g. reports);
  o External Inquiries (e.g. sending request to external search engine).

The row (unadjusted) function points are converted according to functional complexity. It is made based on the number of data types or file types referenced. After that, the value adjustment factor is used to take into account the degree of influence of fourteen general system characteristics, such as heavily used configurations, transaction rate, or complex processing (a five level scale from no-influence to strong-influence throughout). The result of the calculations will be established in form of an adjusted function point count, which will represent the size and complexity of a software system.

IT managers treat *software sizing* as a basis for implementing their own software project management activities, i.e. project planning, implementation of project components in a timely manner, and review of project activities. The most important data for managers concerns *software development effort estimation*, which is the process of predicting the most realistic amount of endeavor expressed in terms of person-hours required to develop the software.

Another important aspect of software measurement applies to defects-based metrics; defect density is one of the indicators of the maturity of the software. Software developers and testers use a rate-metric, which describes how many defects occur for each functionality unit of a system. For some critical systems, the *mean time between failure* (MTBF) indicators is applied.

4.4 Human Factors in Information Systems

Complex information systems cause difficulties not only to their developers, but also to end users [64]. From the users' perspective, IS may have overly complex dependencies, a lack of transparency, intricate control logic, confusing terminology, complicated navigation, and so on. Human factor engineers (professional ergonomists) believe that the workplace has to be essentially matched to the worker, because the better the match the higher the level of worker efficiency. In the case of the design of information systems, it is not about physical ergonomics, but most about organizational and cognitive ergonomics, e.g. when designing human-system interfaces developers must account for the cognitive limits of future users. Developers have to optimize the organizational structures, policies, and processes of such socio-technical systems [65]. It is vitally important to create formal methods for vague business processes in complex information systems. The aim is the reducing of the probability of human errors.

The key objectives of applying human factors to the design of information systems are to improve effectiveness, efficiency, safety, and the satisfaction of the end users. This can be achieved by reducing fatigue and of the learning curve and by ensuring operability (in fact, by meeting the user's needs and wants). Software developers and system engineers have to understand that in fact, with information systems they design and create the information environment for the end users. Because of this, they control what information will be accessible and how it will be presented to the end user. Every design decision has implications for users on their knowledge, attention, memory, reactions, and so on. It is clear that in traditional software development processes only after the implementation of the system remains does it remain to be seen whether these requirements are met. In order not to make serious errors in design, designers of information systems need to understand users – who are they, what are their working qualities, what are their professional needs.

It is worth noting that the interaction of users with information technologies is cognitive. This means that IT designers need to take into account the cognitive processes involved in interaction and the cognitive limitations of real users. For IT systems, the core cognitive aspects of users are perception and recognition, attention, memory, and learning. Designers have to focus on these aspects when creating an information system. The discipline known as *Human Computer Interaction* (HCI) embraces the subject of user aspects of software systems most comprehensively, because the goals of HCI is to develop or improve the safety, utility, effectiveness, efficiency, and appeal of systems that include computers [66]. HCI understand human-computer interface to be when people come together with computer-based systems not only as a physical interface (i.e. screens, menus, buttons, sliders, switches), but also as a logical interface. It is a model of the system from the point of view of the user, in other words, the set of available functions and behaviors.

4.4.1 Human Computer Interaction in Software Development

The special interest group on human-computer interaction of the Association for Computing Machinery (ACM SIGCHI) [67] defines HCI as an inter-disciplinary area of knowledge, which is concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them. According to HCI [68] the evolution of human-computer interfaces has passed through some important stages of development, resulting in today's design methodology, which takes into account not only the user's needs, but also the cognitive characteristics of users. In the beginning, in the 1950s interface was realized at the hardware level. First the systems were not interactive and were available for hardware engineers only. In the 1960s-1970s, the first software interfaces which were accessible for a programmer and which were presented for end users at the terminal level with a command language appeared. During this period, the issues concerning the end-users and software quality of use were not at all important. The creation of user interfaces was not clearly separated as a significant aspect of the design of software systems. In the 1980s, the first interfaces at the interaction dialogue level were designed (graphical user interfaces, later with use of multimedia), but still the quality of communication between human and computer restrained the spread of IT. Relatively recently, at the beginning of the 21st century human-computer interfaces have become pervasive. Nowadays, most end-users of software systems are neither computer specialists nor informaticians. The quality of use has become vital to the success of commercial software. This means that software designers cannot ignore the interaction between the user and the product. Interactive dialogues should equally be in the center of their attention as, say, a database or software architecture.

New specializations among designers became a kind of response to the needs of the IT industry in the context of the achievements of HCI. The first group, *interaction designers* are professionals who are involved in the design of all the interactive aspects of a software product. Interaction designers have to convert business needs and software requirements into user-centered software system experiences. Together with other stakeholders, they have to create user interface mockups or wireframes (prototypes), user flows, and functional specifications; as a result, their work maximizes the quality of use of the software. *Usability engineers* (the second new specialization) perform the measurement of the quality of use (usability or human factors). These people are experts in the relationship between people and software systems, and concentrate on product evaluation. Usability engineers may also be called *user experience specialists*. They ensure that software is user-friendly and the average user can understand and use the software system.

It is worth noting that the terminology concerning human factors and HCI in information systems is fuzzy. The terms are mirrored or rather loosely defined. A full taxonomy was never published. For example, the very popular concept *software usability* is merging semantically in itself the quality of use of a software system. According to the definition from the ISO standard 9241 [69], usability is the "effectiveness, efficiency and satisfaction with which specified users achieve specified goals in particular environments". Effectiveness means the "accuracy and completeness with which specified users can achieve specified goals in particular environments", the efficiency means the "resources expended in relation to the accuracy and completeness of goals achieved", and the satisfaction means the "comfort and acceptability of the work system to its users and other people affected by its use". Such a non-precise and non-technical definition allows for different interpretations of this concept and the results of measurement of usability according to different methodologies are often not comparable. Nevertheless, there are some successful techniques for designing human-computer interaction and ergonomic user interfaces. Achievements in this field rely more on heuristic techniques, rather than mathematical models.

4.4.2 User Centered Development Methodology

Traditional system-centered design places emphasis on the functionality of software systems [70]; with such an approach, developers add user interfaces to data structures and to business logic at the end of the design process. One can say that system-centered design accents the correct software rather than usable software. This means that users have to familiarize and adapt themselves to the software product. At the turn of the 20th and 21st centuries, the situation has changed fundamentally – user-centered design is needed and user interfaces (UI) have became more important. A new product design philosophy was introduced. This is known as "user-centered". User centered approach highlights the users' tasks. It is possible only if the user participates in the early stages of software development. Under this approach, the early prototyping of user interfaces is vitally necessary. This forces the iterative development of user interfaces by a UI-designer and a UI-programmer. Such a user centered development methodology involves users in the design, testing and revision processes, in which the monitoring of usability of software becomes possible.

ISO 9241-210:2010 describes the widely understood human-centered design process for interactive systems. Essential activities in *user-centered design* [71-72] of information systems are:

- Understanding and specifying the context of use of information systems, especially understanding the parent organization and its business;
- Specifying the users and organizational requirements;
- Producing design solutions, preparation of prototypes (mockups or wireframes);
- Evaluating design with users against specified requirements (correction of requirements is permitted).

The basis of user-centered design is a multidisciplinary design team, which provides among other things the participation of non-technical professionals. Together with the active participation of users in all stages of the life cycle of a system, such a team guarantees a non-technocratic approach to information systems. User-centered design is widely practiced as an approach to design of the business information systems, especially to the design of user interfaces and dialogues for such systems. The designer focuses on the user tasks and goals, and on understanding the user environment – not only physical, but also organizational and social.

Bibliography

[1] Paul E. Ceruzzi, *A History of Modern Computing (History of Computing)*, MIT Press, 2003.

[2] Charles Chen, *Overview of Computer Science*, charliedigital.com, http://charliedigital.com/?s=overview+of+computer+science (Accessed May 8, 2015).

[3] Jan L.A. van de Snepscheut. *What Computing is all about.* Text and Monographs in Computer Science. Springer-Verlag, 1993.

[4] Friedrich L. Bauer, *Origins and Foundations of Computing*, Springer-Verlag, Berlin 2010.

[5] Computational Science Education Project, *CSEP e-book*, http://www.phy.ornl.gov/csep/CSEP/TOC.html, (Accessed May 8, 2015).

[6] Rául Rojas (ed), *Encyclopedia of Computers and Computer History*. Chicago: Fitzroy Dearborn, 2001.

[7] Francis Heylighen, Cliff Joslyn, *Cybernetics and Second-Order Cybernetics*, in: R.A. Meyers (ed.), Encyclopedia of Physical Science & Technology, Academic Press, New York, 2001, http://pespmc1.vub.ac.be/papers/cybernetics-epst.pdf

[8] Lenhart Schubert, *Computational Linguistics*, in: Stanford Encyclopedia of Philosophy, Feb 6, 2014, http://plato.stanford.edu/entries/computational-linguistics/ (Accessed May 8, 2015).

[9] David Lazer, Alex Pentland, Lada Adamic, and others, *Computational Social Science*, Science 6 February 2009: 323 (5915), 721-723.

[10] Tal Yarkoni, *Psychoinformatics: New horizons at the interface of the psychological and computing sciences*, Current Directions in Psychological Science, Vol 21(6), Dec 2012, 391-397.

[11] Robert L. Solso, Otto H. MacLin, M. Kimberly MacLin, *Cognitive psychology*. Boston: Allyn & Bacon, 2005.

[12] Rajiv Khosla, Ernesto Damiani, William Grosky, *Human-Centered e-Business*, Kluwer Academic Publishers, 2003.

[13] Ricardo R. Gudwin, *Computational Semiotics*, http://www.dca.fee.unicamp.br/~gudwin/compsemio/ (Accessed May 8, 2015).

[14] Jon Kleinberg, Éva Tardos, *Algorithm Design*, Tsinghua University Press, 2005.

[15] Dexter C. Kozen, *The Design and Analysis of Algorithms*, Springer, 1991.

[16] Massachusetts Institute of Technology, *Big O notation*, mar 16 2003, http://web.mit.edu/16.070/www/lecture/big_o.pdf (Accessed May 8, 2015).

[17] NIST/SEMATECH, *e-Handbook of Statistical Methods*, April, 2012, http://www.itl.nist.gov/div898/handbook/, (Accessed May 8, 2015).

[18] Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, Ramasamy Uthurusamy, *Advances in Knowledge Discovery and Data Mining*, MIT Press, 1996.

[19] David Hemmendinger, computer programming language, Encyclopædia Britannica, Inc., 2015, http://www.britannica.com/EBchecked/topic/130670/computer-programming-language, (Accessed May 8, 2015).

[20] *TIOBE Programming Community Index*, TIOBE Software, 2015, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, (Accessed May 8, 2015).

[21] Stephen O'Grady, *The RedMonk Programming Language Rankings*, RedMonk, 2015, http://redmonk.com/sogrady/2013/02/28/language-rankings-1-13/, (Accessed May 8, 2015).

[22] Pierre Carbonnelle, *The PYPL PopularitY of Programming Language Index*, 2015, http://pypl.github.io/PYPL.html, (Accessed May 8, 2015).

[23] Barzokas Vassilios, *Trends for Languages*, 2012, http://trendyskills.com/, (Accessed May 8, 2015).

[24] Allen B. Tucker, Robert E. Noonan, *Programming Languages: Principles and Paradigms*, McGraw-Hill, 2007.

[25] Дудаков С.М. *Математическое введение в информатику*: Учебное пособие. – Тверь: Твер. гос. ун-т, 2003.[26]

[27] Terrence W. Pratt, Marvin V. Zelkowitz, *Programming Language Design and Implementation*, Prentice Hall, 2001.

[28] Kenneth C. Louden, *Programming Languages: Principles and Practice*, Cengage Learning, 2012.

[29] Dewayne E. Perry, Gail E. Kaiser, *Models of Software Development Environments, Software Engineering,* IEEE Transactions on Software Engineering archive, vol. 17, issue 3, pp. 283 - 295.

[30] Capers Jones, *The Technical and Social History of Software Engineering*, Addison-Wesley Professional, 2013.

[31] Grady Booch, *Object Solutions*, Addison-Wesley, 1995.

[32] Len Bass, Paul Clements, Rick Kazman, *Software Architecture in Practice*, SEI Series in Software Engineering. Addison-Wesley, 2003.

[33] Kenneth E. Kendall, Julie E. Kendall, *Systems Analysis and Design*, New Jersey: Prentice Hall, Eighth Edition, 2011.

[34] Jim Johnson, *CHAOS: The Dollar Drain of IT Project Failures*, Application Development Trends, January 1995.

[35] Deborah Hartmann Preuss, *Interview: Jim Johnson of the Standish Group*, InfoQueue, Aug 25, 2006, http://www.infoq.com/articles/Interview-Johnson-Standish-CHAOS, (Accessed May 8, 2015).

[36] Kathy Schwalbe, *Information Technology Project Management*, Cengage Learning, 2013.

[37] *PMI Education & Resources*, Project Management Institute, 2015, http://www.pmi.org/, (Accessed May 8, 2015).

[38] *A Guide to the Project Management Body of Knowledge* (PMBOK Guide), Project Management Institute, 2013, http://www.pmi.org/pmbok-guide-and-standards/pmbok-guide.aspx/, (Accessed May 8, 2015).

[39] *PRINCE2 Foundation Online*, ILX Group 2015, https://www.prince2.com/, (Accessed May 8, 2015).

[40] *What is Scrum? Why Scrum? Who is Scrum for?*, Scrum Alliance, Inc., 2015, https://www.scrumalliance.org/, (Accessed May 8, 2015).

[41] *What is Scrum?*, SCRUM.ORG, 2015, https://www.scrum.org/resources/what-is-scrum, (Accessed May 8, 2015).

[42] *What is ITIL?*, ACELOS, 2015, https://www.axelos.com/best-practice-solutions/itil, (Accessed May 8, 2015).

[43] Andrew Stellman, Jennifer Greene, *Beautiful Teams: Inspiring and Cautionary Tales from Veteran Team Leaders*, 2009.

[44] Mark Lejk, David Deeks, *An Introduction to Systems Analysis Techniques*, Pearson Education, 2002.

[45] Geoff Cutts, *Structured Systems Analysis and Design Methodology*, Van Nostrand Reinhold Co., New York, NY, 1988.

[46] Charles West Churchman, *The Systems Approach*, Delacorte Press, 1968.

[47] Per Bak, *How Nature Works: The Science of Self-Organized Criticality*. New York: Springer-Verlag, 1996.

[48] Y. Tina Lee, *Information modeling: from design to implementation*, National Institute of Standards and Technology, 1999.

[49] *Unified Modeling Language (UML) Resource Page*, Object Management Group, Inc., 2015, http://www.uml.org/, (Accessed May 8, 2015).

[50] *About OMG*, Object Management Group, Inc., 2015, http://www.omg.org/gettingstarted/gettingstartedindex.htm, (Accessed May 8, 2015).

[51] James Rumbaugh, Ivar Jacobson, Grady Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998.

[52] ISO/IEC/IEEE 42010:2011, *Systems and software engineering — Architecture description*, http://www.iso-architecture.org/ieee-1471/, (Accessed May 8, 2015).
[53] Philippe Kruchten, *Architectural Blueprints — The "4+1" View Model of Software Architecture*. IEEE Software 12 (6), November 1995, pp. 42-50.
[54] Raymond Frost, Jacqueline Pike, Lauren Kenyo, Sarah Pels, *Business Information Systems: Design an App for That*, Flat World Knowledge, Inc., 2011.
[55] Stephen A. White, *Introduction to BPMN*, http://www.omg.org/bpmn/Documents/
[56] Robert Shapiro, Stephen A. White, Conrad Bock and 4 more, *BPMN 2.0 Handbook Second Edition: Methods, Concepts, Case Studies and Standards in Business Process Management Notation (BPMN)*, 2011.
[57] *Bizagi BPM Suite*, Bizagi, 2015, http://www.bizagi.com/en/bpm-suite, (Accessed May 8, 2015).
[58] Yih-Chang Chen, *Empirical Modelling for Participative Business Process Reengineering*, University of Warwick, 2001.
[59] Michael Hammer, James Champy, *Reengineering the Corporation: A Manifesto for Business Revolution*, Collins Business Essentials, 2006.
[60] *Passports to Success in BPM: Real World, Theory and Applications*, WorkFlow Management Coalition, 2015, http://www.wfmc.org/, (Accessed May 8, 2015).
[61] Efraim Turban, Ephraim McLean, James Wetherbe, *Information Technology for Management: Making Connections for Strategic Advantage*, Wiley, 2000.
[62] Norman E. Fenton, Shari Lawrence Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, PWS Publishing Company, Boston, 1997.
[63] Linda M. Laird, M. Carol Brennan, *Software Measurement and Estimation: A Practical Approach*, Wiley-IEEE Computer Society Press, 2006.
[64] *The IFPUG Guide to IT and Software Measurement*, Auerbach Publications, 2012.
[65] Ben Shneiderman, *Software psychology: Human factors in computer and information systems*, Winthrop Publishers, 1980.
[66] *Definition and Domains of ergonomics*, International Ergonomics Association, 2015, http://www.iea.cc/whats/, (Accessed May 8, 2015).
[67] Daniel D. McCracken, Rosalee J. Wolfe, Jared M. Spool, *User-Centered Website Development. A Human-Computer Interaction Approach*, Prentice Hall, 2003.
[68] Human-Computer Interaction Resources, SIGCHI, 2015, http://www.sigchi.org/resources/hcibib, (Accessed May 8, 2015).
[69] David Redmond-Pyle, Alan Moore, *Graphical User Interface Design and Evaluation*, Prentice-Hall, 1995.
[70] ISO 9241-210:2010, *Ergonomics of human-system interaction -- Part 210: Human-centred design for interactive systems, http://www.iso.org/iso/catalogue_detail.htm?csnumber=52075*.
[71] Donald A. Norman, Stephen W. Drape, *User Centered System Design: New Perspectives on Human-computer Interaction*, CRC Press, 1986.
[72] Ben Shneiderman, C. Plaisant, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Pearson Education, Inc., 2005.
[73] Debbie Stone, Caroline Jarrett, Mark Woodroffe, Shailey Minocha, *User Interface Design and Evaluation*. San Francisco, California, USA: Elsevier, 2005.
[74] Sudakov R., Yatsko A., E*lements of applied theory of geometric programming*, Znanie, Moscow, 2004, 220 p. ISBN 507-0029851 (in Russian).
[75] Susłow W., *Analysis and conceptual modelling in software system engineering: humanistic perspective*, University Publishing House of the Koszalin University of Technology, Koszalin 2013, 215 p. ISSN 0239-7129 (in Polish).

Index