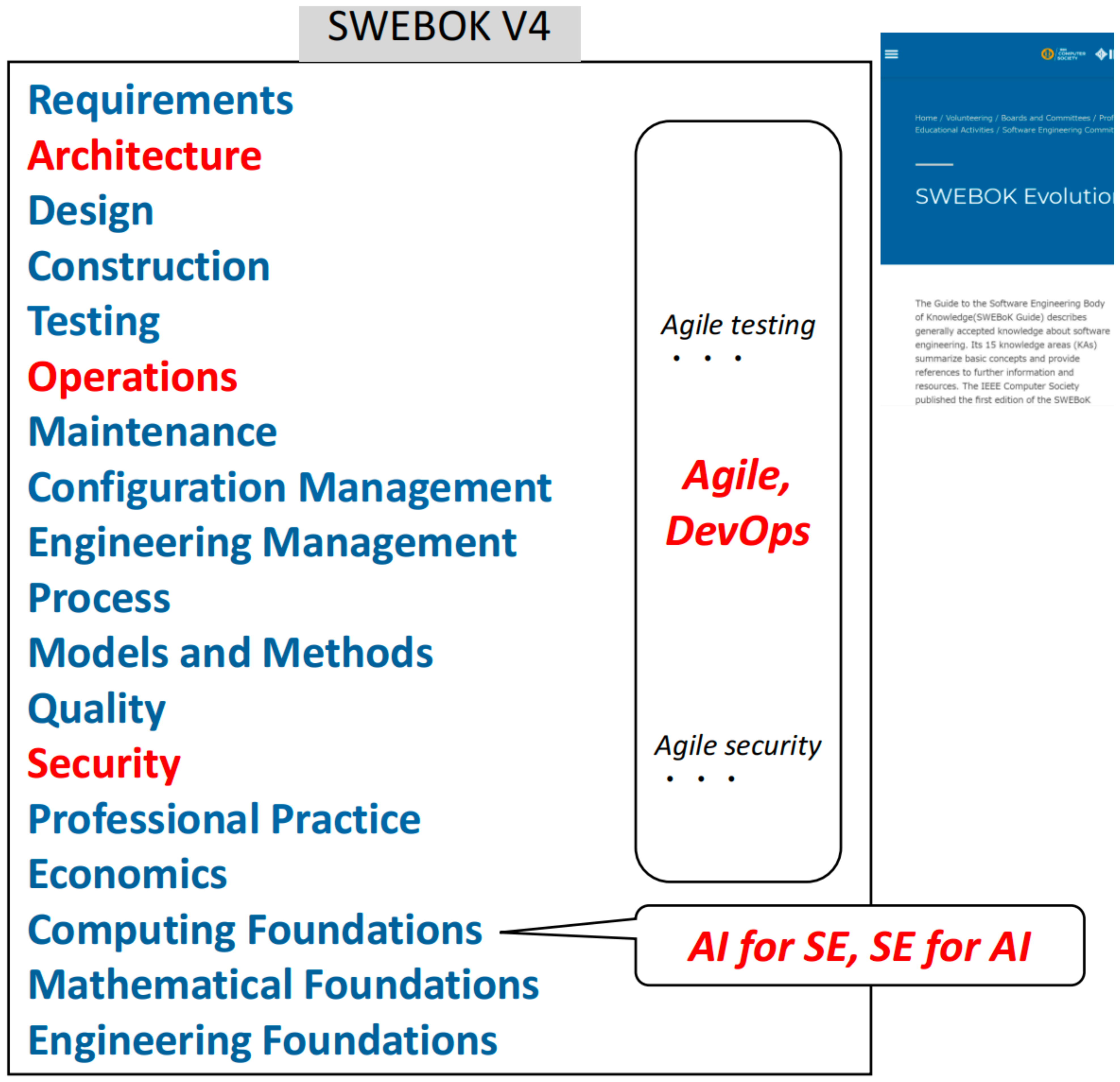
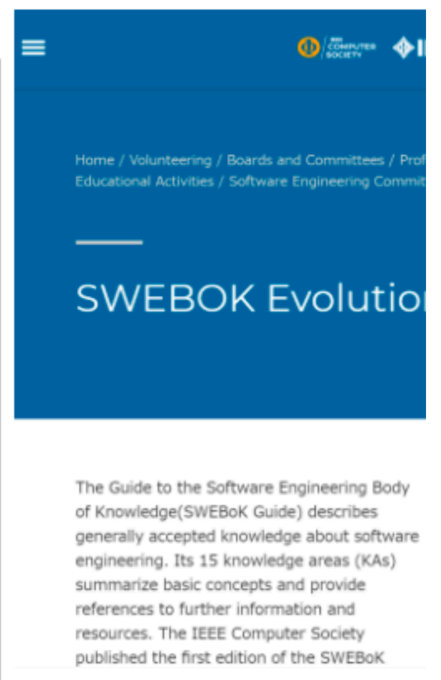
软件工程方法学

Reference

- https://speakerdeck.com/washizaki/swebok-v4-evolution-and-its-impact-on-education

SWEBOK V4




Home / Volunteering / Boards and Committees / Prof
Educational Activities / Software Engineering Commit

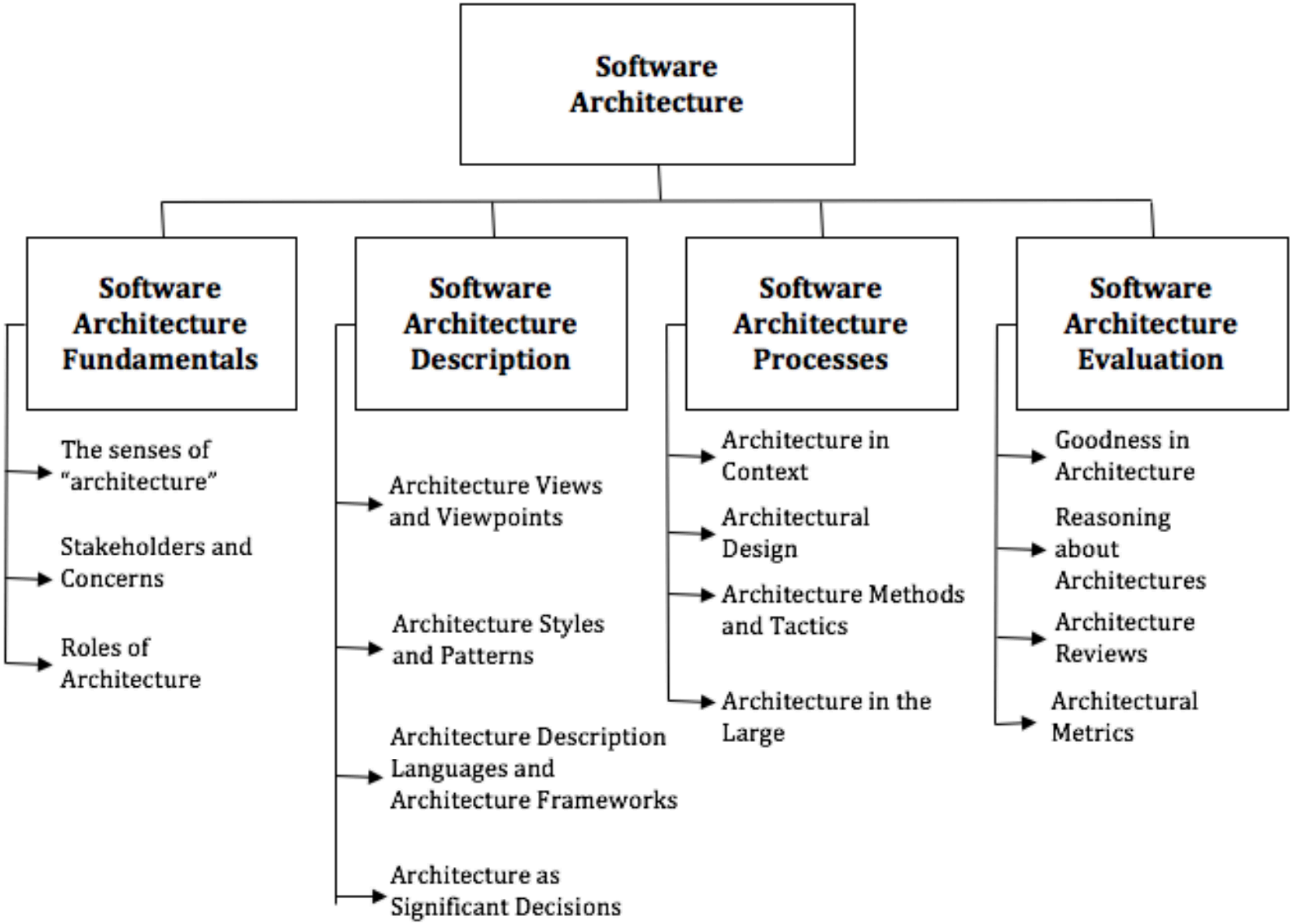
SWEBOK Evolutio

The Guide to the Software Engineering Body of Knowledge(SWEBoK Guide) describes generally accepted knowledge about software engineering. Its 15 knowledge areas (KAs) summarize basic concepts and provide references to further information and resources. The IEEE Computer Society published the first edition of the SWEBoK

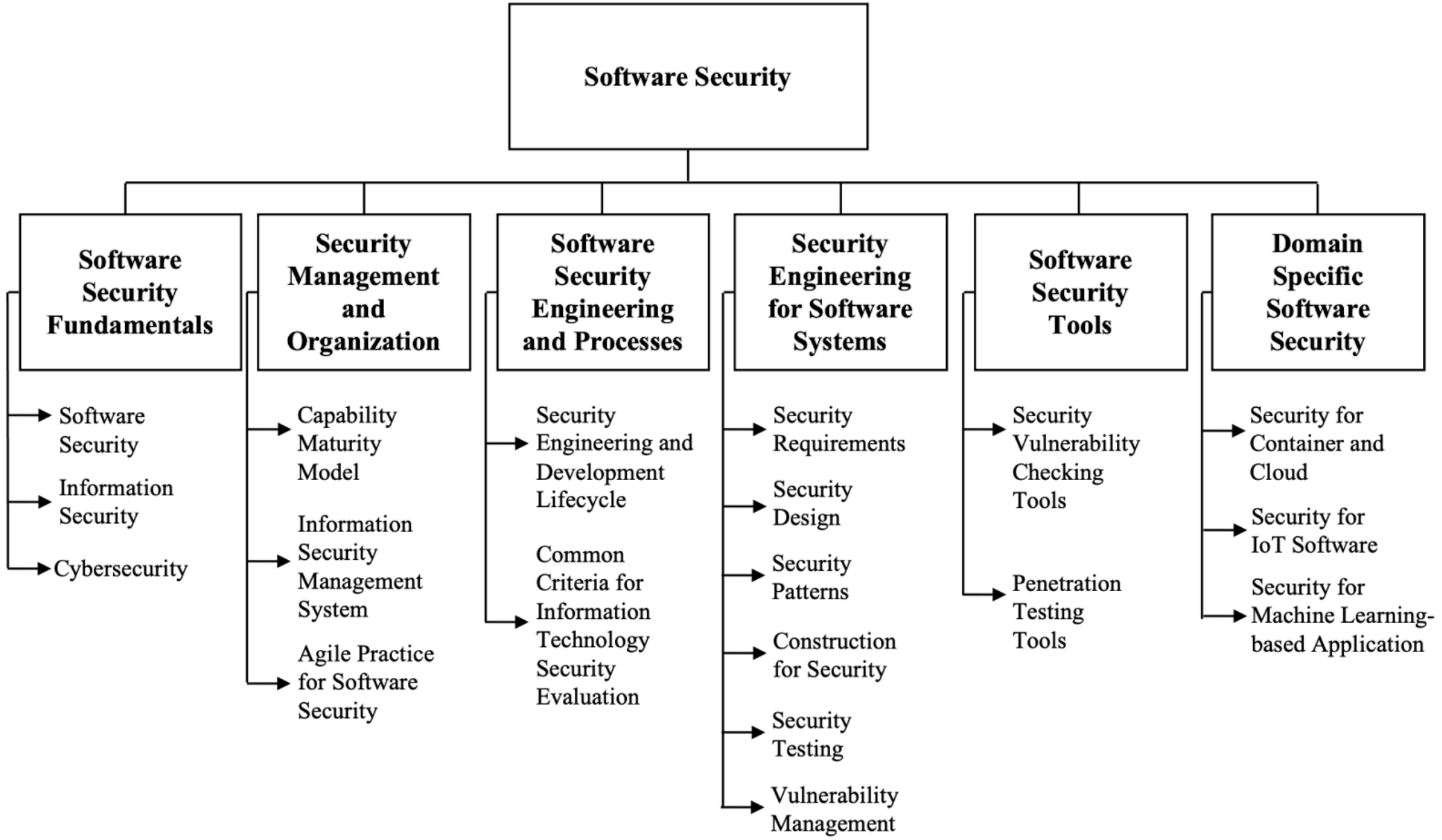
The 17 SWEBOK v4 KAs

- 1. Software Requirements
- 2. Software Architecture
- 3. Software Design
- 4. Software Construction
- 5. Software Testing
- 6. Software Engineering Operations
- 7. Software Maintenance
- 8. Software Configuration Management
- 9. Software Engineering Management
- 10. Software Engineering Models and Methods
- 11. Software Engineering Process
- 12. Software Quality
- 13. Software Security
- 14. Software Engineering Economics
- 15. Software Engineering Professional Practice
- 16. Computing Foundations
- 17. Mathematical Foundations
- 18. Engineering Foundations

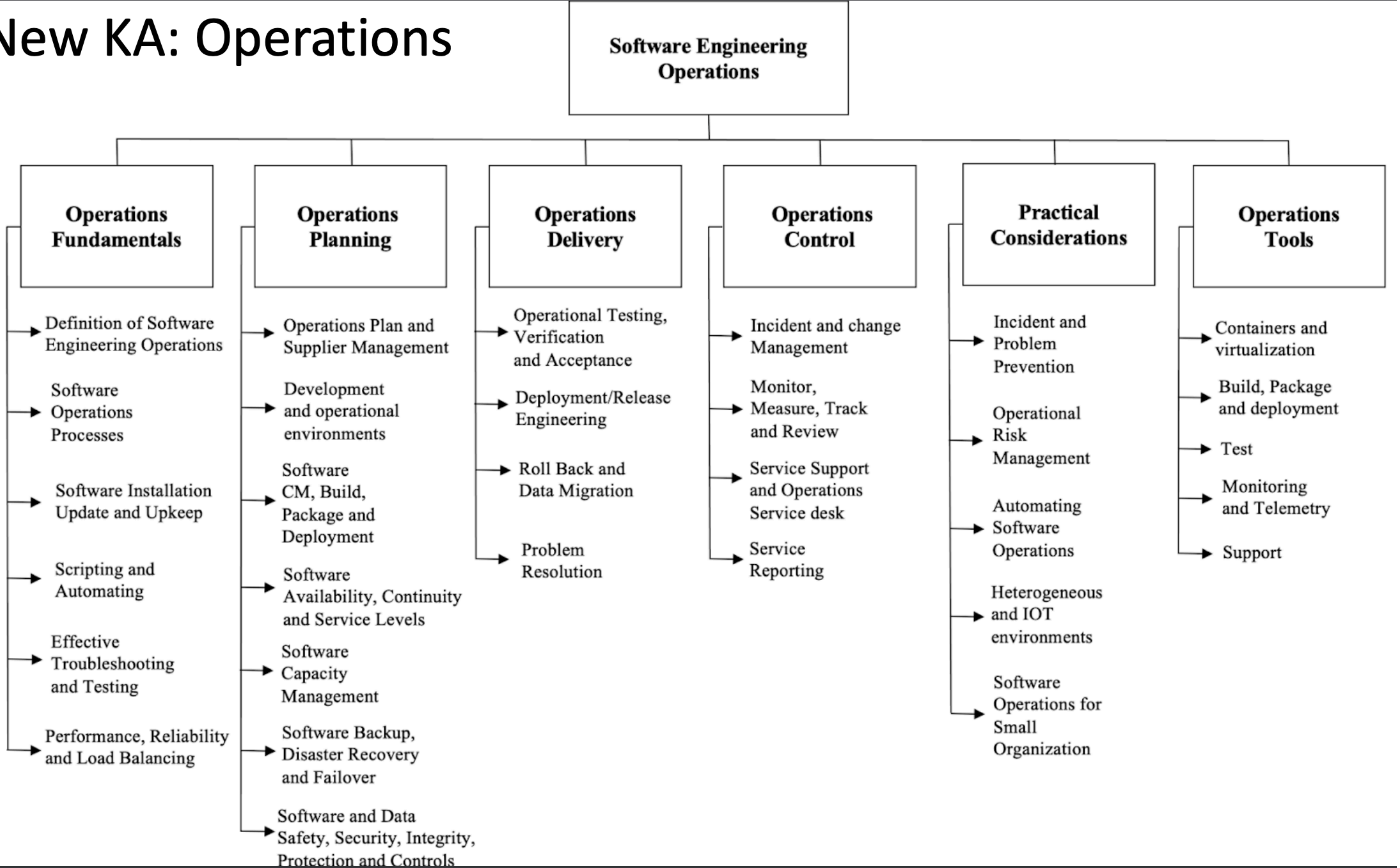
New KA: Software Architecture


Software Architecture
Software Architecture Fundamentals
The senses of "architecture"
Stakeholders and Concerns
Roles of Architecture
Software Architecture Description
Architecture Views and Viewpoints
Architecture Styles and Patterns
Architecture Description Languages and Architecture Frameworks
Architecture as Significant Decisions
Software Architecture Processes
Architecture in Context
Architectural Design
Architecture Methods and Tactics
Architecture in the Large
Software Architecture Evaluation
Goodness in Architecture
Reasoning about Architectures
Architecture Reviews
Architectural Metrics

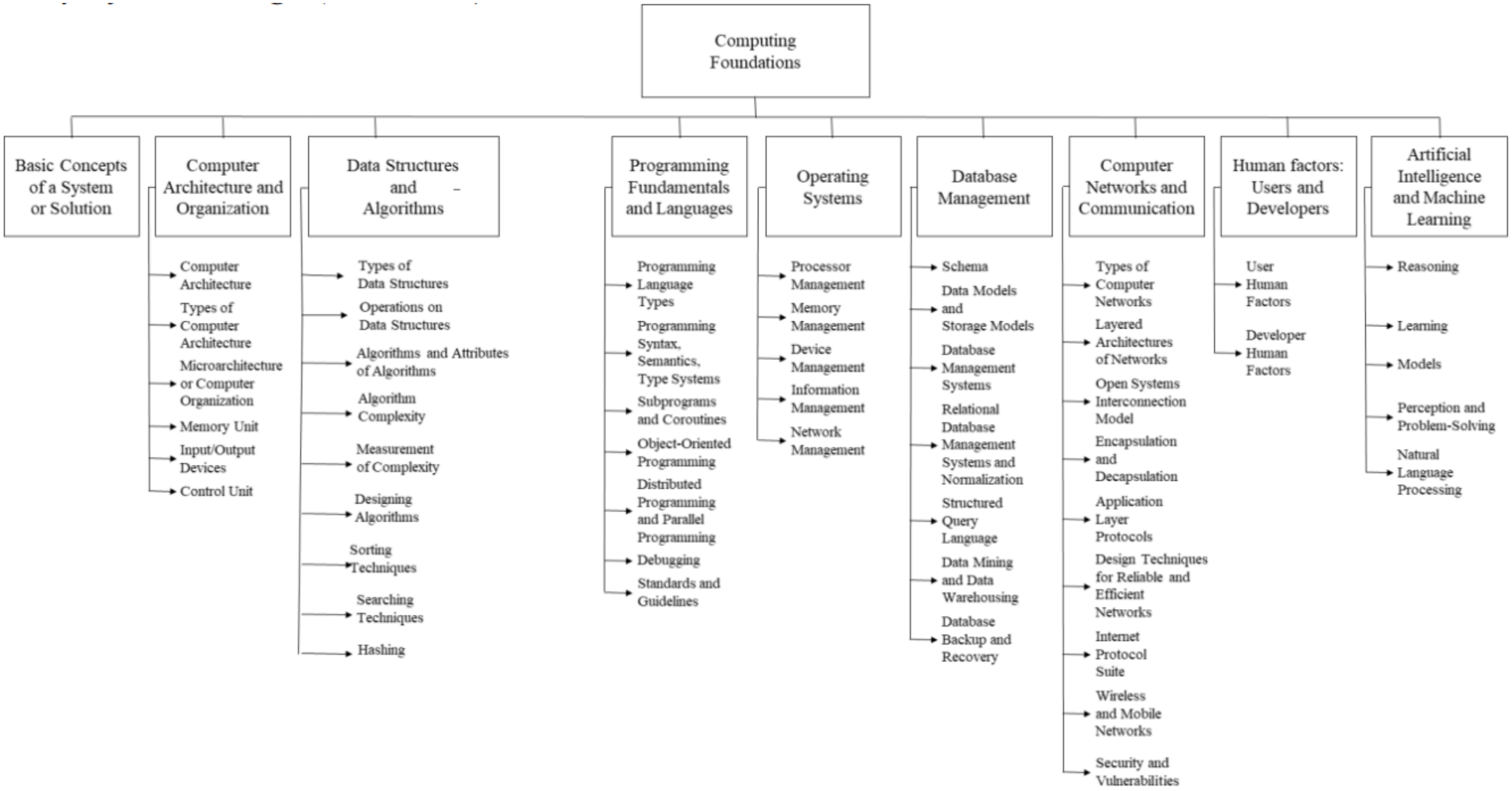
New KA: Software Security


Software Security
Software Security Fundamentals
Software Security
Information Security
Cybersecurity
Security Management and Organization
Capability Maturity Model
Information Security Management System
Agile Practice for Software Security
Software Security Engineering and Processes
Security Engineering and Development Lifecycle
Common Criteria for Information Technology Security Evaluation
Security Engineering for Software Systems
Security Requirements
Security Design
Security Patterns
Construction for Security
Security Testing
Vulnerability Management
Software Security Tools
Security Vulnerability Checking Tools
Penetration Testing Tools
Domain Specific Software Security
Security for Container and Cloud
Security for IoT Software
Security for Machine Learning-based Application

New KA: Operations


Software Engineering Operations
Operations Fundamentals
Definition of Software Engineering Operations
Software Operations Processes
Software Installation Update and Upkeep
Scripting and Automating
Effective Troubleshooting and Testing
Performance, Reliability and Load Balancing
Operations Planning
Operations Plan and Supplier Management
Development and operational environments
Software CM, Build, Package and Deployment
Software Availability, Continuity and Service Levels
Software Capacity Management
Software Backup, Disaster Recovery and Failover
Software and Data Safety, Security, Integrity, Protection and Controls
Operations Delivery
Operational Testing, Verification and Acceptance
Deployment/Release Engineering
Roll Back and Data Migration
Problem Resolution
Operations Control
Incident and change Management
Monitor, Measure, Track and Review
Service Support and Operations Service desk
Service Reporting
Practical Considerations
Incident and Problem Prevention
Operational Risk Management
Automating Software Operations
Heterogeneous and IOT environments
Software Operations for Small Organization
Operations Tools
Containers and virtualization
Build, Package and deployment
Test
Monitoring and Telemetry
Support

Computing Foundation


Computing Foundations
Basic Concepts of a System or Solution
Computer Architecture and Organization
Computer Architecture
Types of Computer Architecture
Microarchitecture or Computer Organization
Memory Unit
Input/Output Devices
Control Unit
Data Structures and Algorithms
Types of Data Structures
Operations on Data Structures
Algorithms and Attributes of Algorithms
Algorithm Complexity
Measurement of Complexity
Designing Algorithms
Sorting Techniques
Searching Techniques
Hashing
Programming Fundamentals and Languages
Programming Language Types
Programming Syntax, Semantics, Type Systems
Subprograms and Coroutines
Object-Oriented Programming
Distributed Programming and Parallel Programming
Debugging
Standards and Guidelines
Operating Systems
Processor Management
Memory Management
Device Management
Information Management
Network Management
Database Management
Schema
Data Models and Storage Models
Database Management Systems
Relational Database Management Systems and Normalization
Structured Query Language
Data Mining and Data Warehousing
Database Backup and Recovery
Computer Networks and Communication
Types of Computer Networks
Layered Architectures of Networks
Open Systems Interconnection Model
Encapsulation and Decapsulation
Application Layer Protocols
Design Techniques for Reliable and Efficient Networks
Internet Protocol Suite
Wireless and Mobile Networks
Security and Vulnerabilities
Human factors: Users and Developers
User Human Factors
Developer Human Factors
Artificial Intelligence and Machine Learning
Reasoning
Learning
Models
Perception and Problem-Solving
Natural Language Processing

1 Basic Concepts of a System or Solution

- The problem to be solved has to be analyzed in greater detail for functional requirements, user interactions, performance requirements, deviceinterfaces, security, vulnerability, durability and upgradability. A system is an integrated set of subsystems, modules and components that perform specific functions independently. Delineating the problem and solution is critical.

- An engineered system ensures the subsystems are designed to be:

  - Modular: Each subsystem (module) is uniform (similarsize).

  - Cohesive: Each subsystem performs one specific task. Ideally, systems should be highly cohesive.

  - Coupled: Each subsystem functions independently, as much as possible. Ideally, systems should be loosely coupled.

Data Sturctures and Algorithms

- Types of Data Sturctures
- Operations on Data Structures
- Algorithms and Attributes of Algorithms
- Algorithms Complexity
- Measurement of Complexity
- Designing Algorithms
- Sorting Techniques
- Searching Techniqures
- Hashing

Types of Data Structures

- Data type is an attribute of data. Various data types are identified and defined based on different characteristics of data, the need for grouping data items and various operations performed on data. Data structures are grouped primarily based on the physical and logical ordering of data items. Primarily, data is grouped into three types: basic, composite or compound, and abstract.
    - Basic or primitive data types include character, integer, float or real, Boolean, and pointer data.
    - Compound data types are made of multiple basic or primitive, or even multiple compound data types. Some of the compound data types include sets, graphs, records and partitions.
    - An Abstract Data Type (ADT) is defined by its behavior (semantics) from the user perspective, specifically from the point of possible values and operations.
- Composite or compound data types are further grouped under linear, and hierarchical or nonlinear data types.
    - Linear data types include one dimensional and multidimensional arrays, strings, linked lists (singly linked lists, doubly linked lists, circular lists), stacks, queues, and hash tables.
    - Hierarchical or nonlinear data types include trees, binary trees, n-array trees, B trees, B+ trees, weighted balanced trees, red-black trees, graphs, heaps, binary heaps and graphs.
    - In the current era of free text queries or natural language processing, software engineers may need to understand strings and various operations on strings, and to be able to analyze skip lists.

Designing Algorithms

- A software engineer has to know a few standard algorithms and relevant concepts, including the following:
    - Common types of algorithms: Brute force algorithm, Recursive algorithm, Divide & Conquer algorithm, Dynamic programming algorithms, Greedy algorithm, Backtracking algorithms, Randomized algorithms.
    - Randomized approximation algorithms, randomized rounding, approximation algorithms, P and NP complexity class algorithms, Cook's theorem, reductions and completeness algorithms.
    - Multiple comparison operations performed simultaneously in a network model of computation. Popular sorting network algorithms include comparison networks, zero-one principle, merging network and bitonic sorter.
    - Optimized algorithms for performing several operations on a matrix, such as matrix multiplication, transposition, matrix inversion, median, and finding determinants.
    - Cryptographic complexity and algorithms: secret key (symmetric) encryption algorithms, public key (asymmetric) encryption algorithms and hash functions.
    - One-way functions, class UP, space complexity, deterministic and nondeterministic space complexity classes, the reachability method, and Savitch's theorem.
    - Graph representations, graph algorithms, breadth-first and depth-first search, topological sort, minimum spanning tree,Kruskal and Prim algorithms, and single-source shortest paths (Bellman-Ford and Dijkstra algorithms).
    - Complexity of randomized computation, interactive proofs, complexity of counting, Boolean circuit complexity.

Sorting Techniques

- Popular sorting algorithms include
    - Linear sort, Bubble sort, Quick sort,
    - Merge sort, Radix sort, Heap sort,
    - Bucket sort, Pigeonhole sort, Bitonic sort,
    - Tree sort, Cartesian Tree sort, 3-Way Quick sort,
    - 3-Way Merge sort, and Sorting Singly / Doubly linked lists.

Searching Techniques

- Popular search algorithms include
  - linear, binary,
  - jump, interpolation, exponential,
  - Fibonacci, sub-list (search a linked list in another list),
  - logarithmic, tree and hashing.

Hashing Techniques

- Different properties of hash functions, such as uniformity, efficiency, universality, applicability, deterministic, defined or variable range, data normalization, testing, and measurement, must be understood and considered when designing or choosing a hash function.

- Various types of hash functions are designed for different types of key values, applications, and database sizes. Hash function types include trivial hash function, division method, mid-square method, digit folding method, multiplicative hashing, double hashing, open and closed hashing, rehashing, extendible hashing, and cryptographic and noncryptographic hash functions.

- Software engineers are expected to learn, implement and be able to compare different types of hashing algorithms, various collision resolution techniques, linear probing, quadratic probing, separate chaining, and open addressing.

Programming Fundamentals and Languages

- Programming Language Types
- Programming Syntax, Semantics, Type Systems
- Susbprograms and Coroutines
- Object-Oriented Programming
- Distributed Programming and Parallel Programming
- Debugging
- Standards and Guidelines

Susbprograms and Coroutines

- The subprograms have an entry point and typically have multiple input parameters on which the subprogram acts and produces output. The scope of input parameters is local to the subprogram.

- Subprograms that return value by their name (which can be used as a variable in a statement) are called functions, and subprograms designed not to return any value are called procedures.

Coroutine

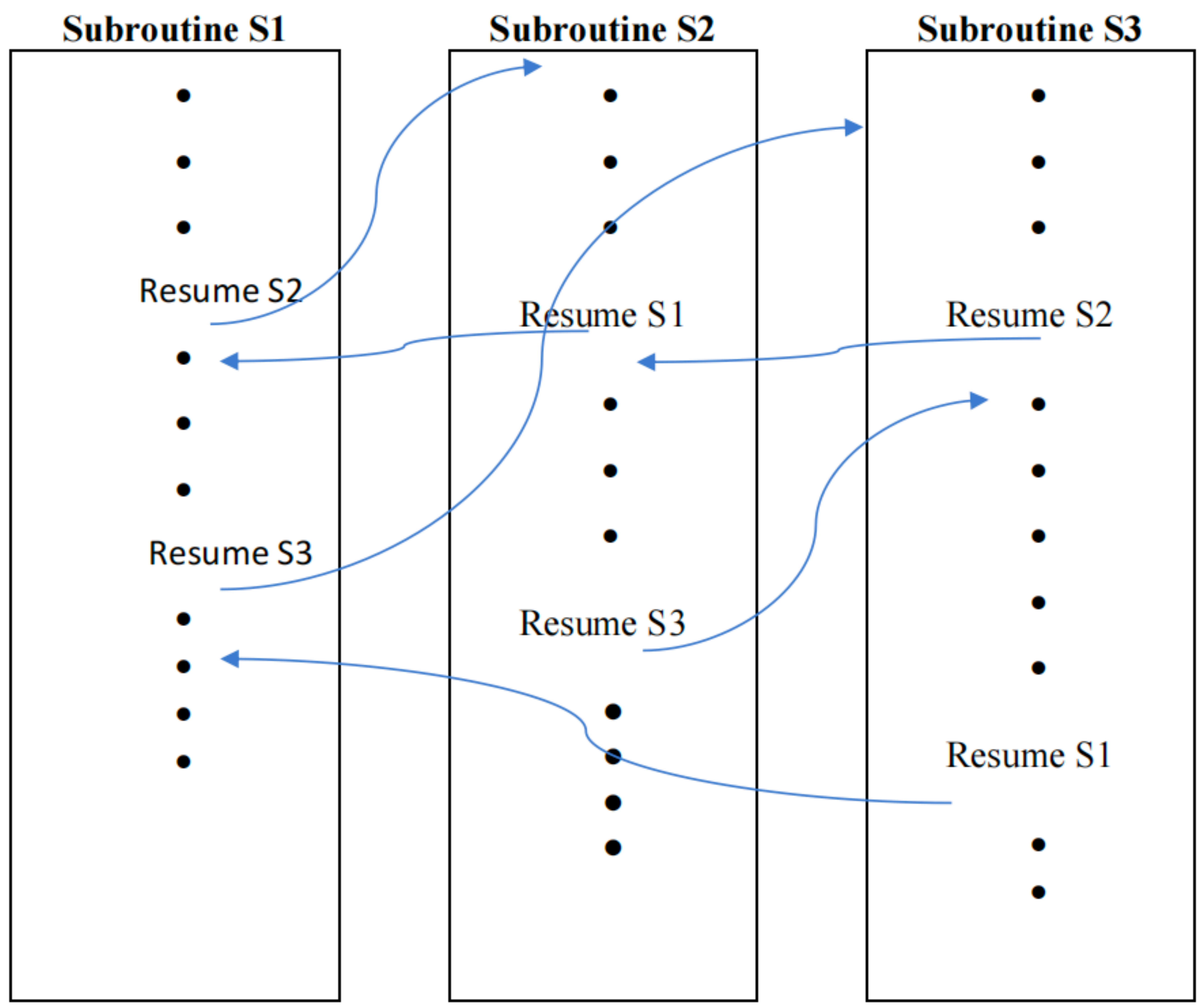
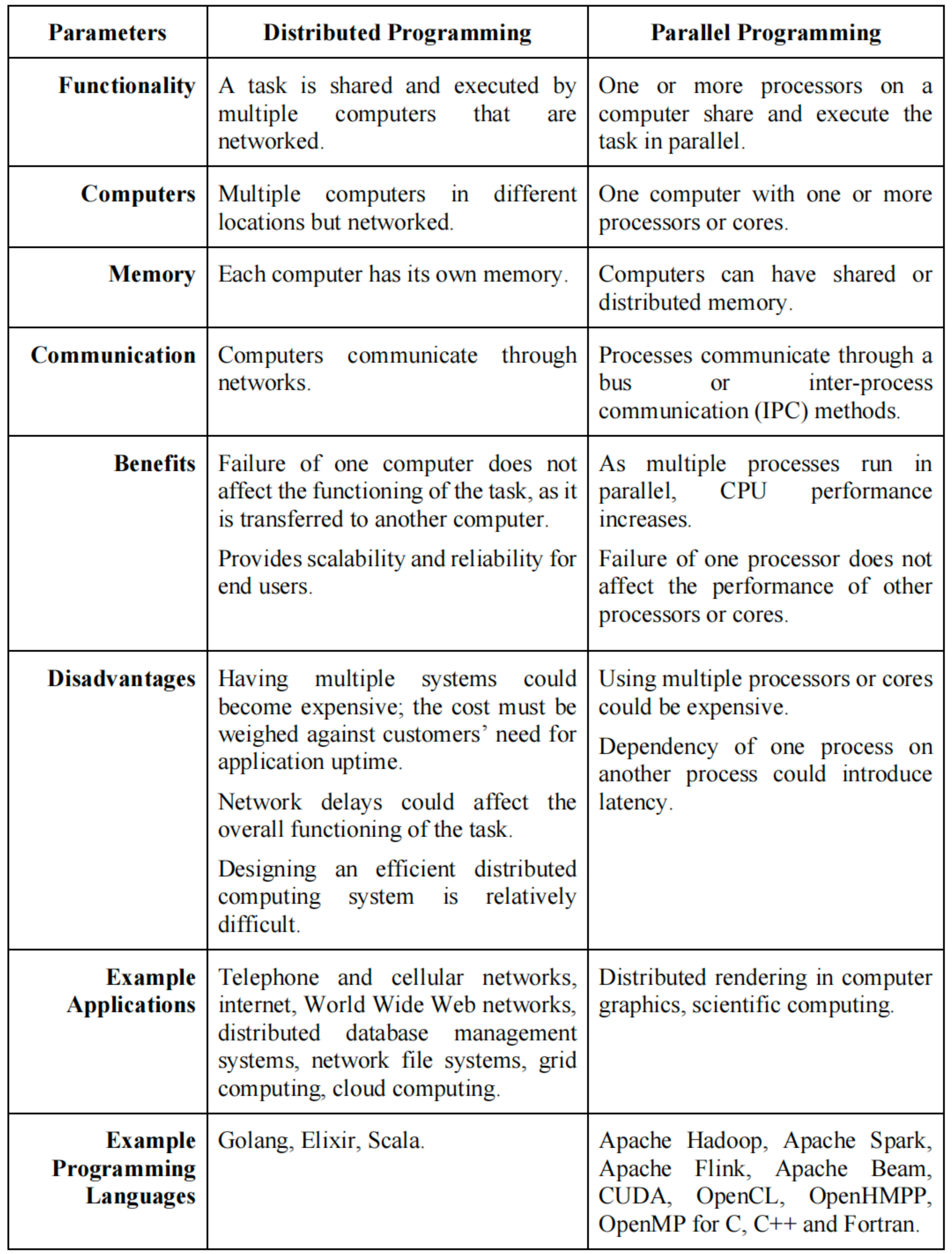


Figure 16.3. Example of Coroutine

Distributed Programming and Parallel Programming



Parameters	Distributed Programming	Parallel Programming
Functionality	A task is shared and executed by multiple computers that are networked.	One or more processors on a computer share and execute the task in parallel.
Computers	Multiple computers in different locations but networked.	One computer with one or more processors or cores.
Memory	Each computer has its own memory.	Computers can have shared or distributed memory.
Communication	Computers communicate through networks.	Processes communicate through a bus or inter-process communication (IPC) methods.
Benefits	Failure of one computer does not affect the functioning of the task, as it is transferred to another computer. Provides scalability and reliability for end users.	As multiple processes run in parallel, CPU performance increases. Failure of one processor does not affect the performance of other processors or cores.
Disadvantages	Having multiple systems could become expensive; the cost must be weighed against customers' need for application uptime. Network delays could affect the overall functioning of the task. Designing an efficient distributed computing system is relatively difficult.	Using multiple processors or cores could be expensive. Dependency of one process on another process could introduce latency.
Example Applications	Telephone and cellular networks, internet, World Wide Web networks, distributed database management systems, network file systems, grid computing, cloud computing.	Distributed rendering in computer graphics, scientific computing.
Example Programming Languages	Golang, Elixir, Scala.	Apache Hadoop, Apache Spark, Apache Flink, Apache Beam, CUDA, OpenCL, OpenHMPP, OpenMP for C, C++ and Fortran.

Table 16.3. Comparison of Distributed and Parallel Programming

Debugging

- Syntax error
- Runtime error
- Logical error

Standards and Guidelines

- An estimated 82% of vulnerabilities are caused by clashes between programming styles.
    - https://www.ptsecurity.com/wwen/analytics/web-vulnerabilities-2020/
- SC 22 is a subcommittee of the Joint Technical Committee ISO/IEC JTC 1Computing Foundations of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) for defining standards for programming languages, their environments and system software interfaces (ISO/IEC JTC 1/SC 22). Software engineers are recommended to refer these standards as well.
    - https://www.iso.org/committee/45202.html

Artifical Intelligence and Machine Learning

- Reasoning
- Learning
- Models
- Perception and Problem-Solving
- Natural Language Processing
- AI and Software Engineering

Reasoning

- Deductive Reasoning
  - is a standard and strategic approach to mapping available facts, information and knowledge to arrive at a conclusion. In this approach, available facts and information are considered to be authentic.
- Inductive Reasoning
  - is about introducing a hypothesis and creating generalizations from the available facts and premises.
- Abductive Reasoning
  - starts with an incomplete set of data or information and proceeds to derive the most likely conclusion from the latest data.
- Common Sense Reasoning
- Monotonic Reasoning
- Non-Monotonic Reasoning
  - (NMR) occurs when the inference changes values or direction based on new knowledge or information. NMR is based on assumptions and deals with incomplete or not-known facts.

Learning

- Supervised Learning
- Unsupervised Learning
- Semi-supervised Learning
- Reinforcement Learning

Models

- Linear Regression
- Logistic Regression
- Artifical Neural Networks
- Decision Tree
- Naive Bayes
- Support Vector Machine
- Random Forest

Perception and Problem-Solving

- Based on capabilities and functionality, AI systems are categorized into multiple types.
  - Type I AI systems are designed to do specific tasks with intelligence. Examples include Chess games, speech and image recognition, among others.
  - Type II AI systems analyze the current situation or environment and do not normally refer to previous decisions made in a similar situation to arrive at an appropriate action. Reactive systems or reactive machines typically make decisions and execute commands at that instance, referring to the existing knowledge base. A good example is a self-driving cars.
  - Type III, or self-aware, AI systems have consciousness and are mindful. These systems adopt the mind theory and predict the mood of the other person or entity based on the person’s action or type of action. For example, if the driver in the vehicle behind the system honks, then the AI system might conclude that the driver is angry or unhappy. Social and ethical behavior is part of conscious systems.

Natural Language Processing

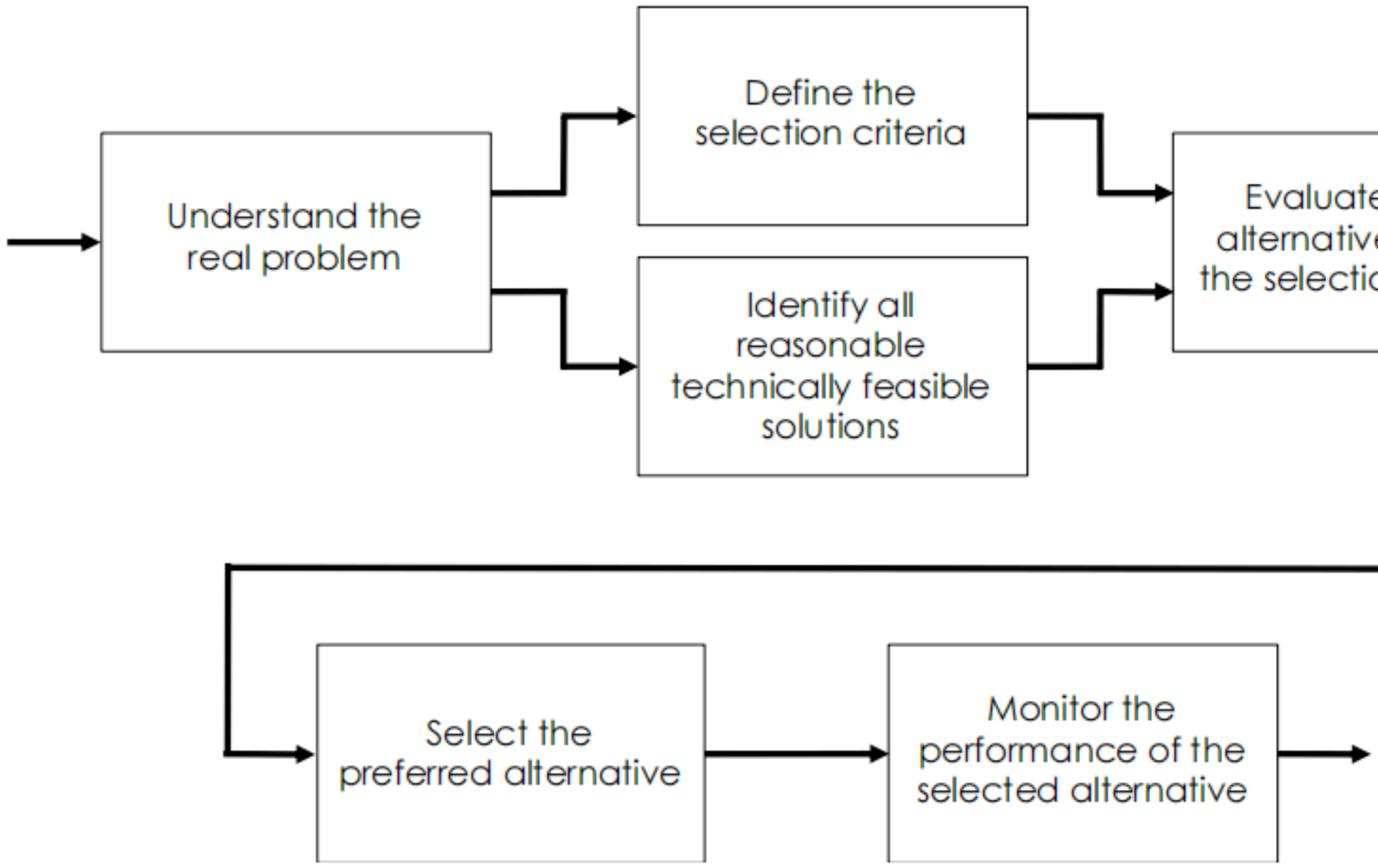
- Natural language processing (NLP) is a crucial part of AI systems, enabling users to interact with the AI systems in a way that is similar to how they interact with other humans.

- AI systems understand human languages and execute commands delivered in those languages. AI systems that work on voice commands need to understand not only the human language, but also the slang or pronunciation of the user.

AI and Software Engineering

- Software engineering and AI are mutually related to each other in basically two ways:
    - AI applications in software engineering (i.e., AI for SE)
    - software engineering for AI systems (i.e., SE for AI).
- AI for SE aims to establish efficient ways of building high-quality software systems by replicating human developers' behavior. It ranges over almost all development stages, from resolving ambiguous requirements to predicting maintainability, particularly well applied in software quality assurance and analytics, such as defect prediction, test case generation, vulnerability analysis, and process assessment [15]. Although humancentric software engineering activities benefit, engineers should be aware of limitations and challenges inherent to the nature of AI and ML, especially the uncertain and stochastic behavior and the necessity of sufficiently labeled and structured datasets [15].
- The development of AI systems is different from traditional software systems since the rules and system behavior of AI systems are inferred from training data rather than written down as program code [16]. Thus, there is a need for particular support of SE for AI, such as interdisciplinary collaborative teams of data scientists and software engineers, software evolution focusing on large and changing datasets, and ethics and equity requirements engineering [16]. Recommended software engineering practices for AI are often formalized as patterns, such as ML software design patterns [17].

Engineering Foundation


Understand the real problem
Define the selection criteria
Identify all reasonable technically feasible solutions
Evaluate each alternative against the selection criteria
Select the preferred alternative
Monitor the performance of the selected alternative

The Engineering Process

Engineering Design

- A product’s life cycle costs are largely influenced by its design. This is true for manufactured products as well as for software. The design of software is guided by the features to be implemented and the quality attributes to be achieved. It is important to note that software engineers use “design” within their own context; while there are some commonalities, there are also many differences between engineering design as discussed in this section and software engineering design as discussed in the Software Architecture KA and Software Design KA. The scope of engineering design is generally viewed as much broader than that of software design.

- Many disciplines engage in problem solving activities where there is a single correct solution. In engineering, most problems have many solutions and the focus is on finding a feasible solution (among many alternatives) that best meets the needs presented. The set of possible solutions is often constrained by explicitly imposed limitations such as cost, available resources, and the state of discipline or domain knowledge. In engineering problems, sometimes there are also implicit constraints (such as the physical properties of materials or laws of physics) that also restrict the set of feasible solutions for a given problem.

Abstraction and Encapsulation

- Abstraction is an indispensible technique associated with problem solving. It refers to both the process and result of generalization by reducing the information of a concept, a problem, or an observable phenomenon so that one can focus on the “big picture.” One of the most important skills in any engineering undertaking is framing the levels of abstraction appropriately.

- “Through abstraction,” according to Voland, “we view the problem and its possible solution paths from a higher level of conceptual understanding. As a result, we may become better prepared to recognize possible relationships between different aspects of the problem and thereby generate more creative design solutions” [2*]. This is particularly true in computer science in general (such as hardware vs. software) and in software engineering in particular (data structure vs. data flow, and so forth).

- According to Dijkstra, “The purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise”

Abstraction and Encapsulation

- Levels of abstraction
  - When abstracting, we concentrate on one of the big picture at a time with confidence that we can then connect effectively with levels above and below.
- Encapsulation
  - Encapsulation is a mechanism used to implement abstraction.
- Hierarchy
  - When we use abstraction in our problem formulation and solution, we may use different abstractions at different times. In other words, we work on different levels of abstraction as the situation calls.
- Alternate Abstracton
  - Sometimes it is useful to have multiple alternate abstractions for the same problem so that one can keep different perspectives in mind.

Empirical Methods and Experimental Techniques

- Designed experiments
    - A designed or controlled experiment is an investigation of a testable hypothesis where one or more independent variables are manipulated to measure their effect on one or more dependent variables.
- Observational studies
    - An observational or case study is an empirical inquiry that makes observations of processes or phenomena within a real-world context.
- Retrospective studies
    - A retrospective study involves the analysis of historical data.

Statistical Analysis

- Unit of Analysis (Sampling Units), Population, and Sample
- Correlation and Regression

Modeling，Simulation， and Prototyping

- Modeling is part of the abstraction process used to represent some aspects of a system.

- Simulation uses a model of the system and provides a means of conducting designed experiments with that model to better understand the system, its behavior, and relationships between subsystems, as well as to analyze aspects of the design.

- Modeling and simulation are techniques that can be used to construct theories or hypotheses about the behavior of the system; engineers then use those theories to make predictions about the system.

- Prototyping is another abstraction process where a partial representation (that captures aspects of interest) of the product or system is built. A prototype may be an initial version of the system but lacks the full functionality of the final version.

Modeling

- Iconic
  - maps
- Analogic
  - A miniature airplane for wind tunnel testing
- symbolic
  - F= Ma

Simulation

- In the context of software engineering, the emphasis will be primarily on discrete simulation.
- Discrete simulations may model event scheduling or process interaction. The main components in such a model include entities, activities and events, resources, the state of the system, a simulation clock, and a random number generator. Output is generated by the simulation and must be analyzed.

Prototyping

- There are many uses for a prototype, including
    - the elicitation of requirements,
    - the design and refinement of a user interface to the system,
    - validation of functional requirements
    - and so on.

Measurement

- Levels(Scales) of Measuremnt

- It is to be noted that measurement may be carried out in four different scales:
    - nominal, 名义上
    - ordinal, 序数上
    - interval, 区间上
    - ratio. 比率上

Measurement

- Levels(Scales) of Measuremnt

- 以下是哪一个度量级别的典型案例
  - Job titles in a company
  - The software development life cycle (SDLC) model (like waterfall, iterative, agile, etc.) followed by different software projects
  - nominal 名义上

Measurement

- Levels(Scales) of Measuremnt

- 以下是哪一个度量级别的典型案例
    - Skill levels (low, medium, high)
    - Capability Maturity Model Integration (CMMI) maturity levels of software development organizations
    - Level of adherence to process as measured in a 5-point scale of excellent, above average, average, below average, and poor, indicating the range from total adherence to no adherence at all
    - ordinal　序数上

Measurement

- Levels(Scales) of Measuremnt

- 以下是哪一个度量级别的典型案例
  - Measurement of temperature in different scales, such as Celsius and Fahrenheit. Suppose T1 and T2 are temperatures measured in some scale. We note that the fact that T1 is twice T2 does not mean that one object is twice as hot as another. We also note that the zero points are arbitrary.
  - Calendar dates. While the difference between dates to measure the time elapsed is a meaningful concept, the ratio does not make sense.
  - Many psychological measurements aspire to create interval scales. Intelligence is often measured in interval scale, as it is not necessary to define what zero intelligence would mean.
  - interval　区间上

Measurement

- Levels(Scales) of Measuremnt

- 以下是哪一个度量级别的典型案例
    - the number of statements in a software program
    - temperature measured in the Kelvin (K) scale or in Fahrenheit (F).
    - ratio　　　比率上

Measurement

- Direct and Derived Measures
  - An example of a direct measure would be a count of how many times an event occurred, such as the number of defects found in a software product.
  - An example of a derived measure would be calculating the productivity of a team as the number of lines of code developed per developer month.

Root Cause Analysis

- Root cause analysis (RCA) is a class of problem-solving methods aimed at identifying underlying causes of undesirable outcomes. RCA methods identify why and how an undesirable outcome happened, allowing organizations to take effective actions to prevent recurrence. Instead of merely addressing immediately obvious symptoms, problems can be solved by eliminating root causes. RCA can play several important roles on software projects, including:
    - identify the real problem to be solved by an engineering effort;
    - expose the underlying drivers of risk when performing project risk assessments;
    - reveal opportunities and actions for software process improvement;
    - discover sources of recurring defects (i.e., defect causal analysis).

RCA techniques

- Change Analysis compares situations where an undesirable outcome happened with similar situations where it did not. The root cause is likely in the area of difference;
- 5-Whys (see, for example, [2*, c4]) starts with an undesirable outcome and uses repeated “Why?” question-answer cycles to isolate root cause;
- Cause-and-Effect diagrams, sometimes called Ishikawa diagrams [15] or Fishbone charts, break down, in successive levels of detail, causes that potentially contribute to an undesirable outcome. Causes are often grouped into major categories such as people, process, tools, materials, measurements, and environment. The diagram takes the form of a tree of potential causes that can all result in that undesirable outcome;
- Fault Tree Analysis (FTA) is a more formal approach to cause-and-effect diagramming which is more specific about and-or relationships between causes and effects. In some cases, any one of multiple causes can drive the effect (an “or” relationship), in other cases a combination of multiple causes are required to drive the effect (an “and” relationship). Cause-and-effect diagrams do not distinguish between and-or relationships, Fault tree analysis does;

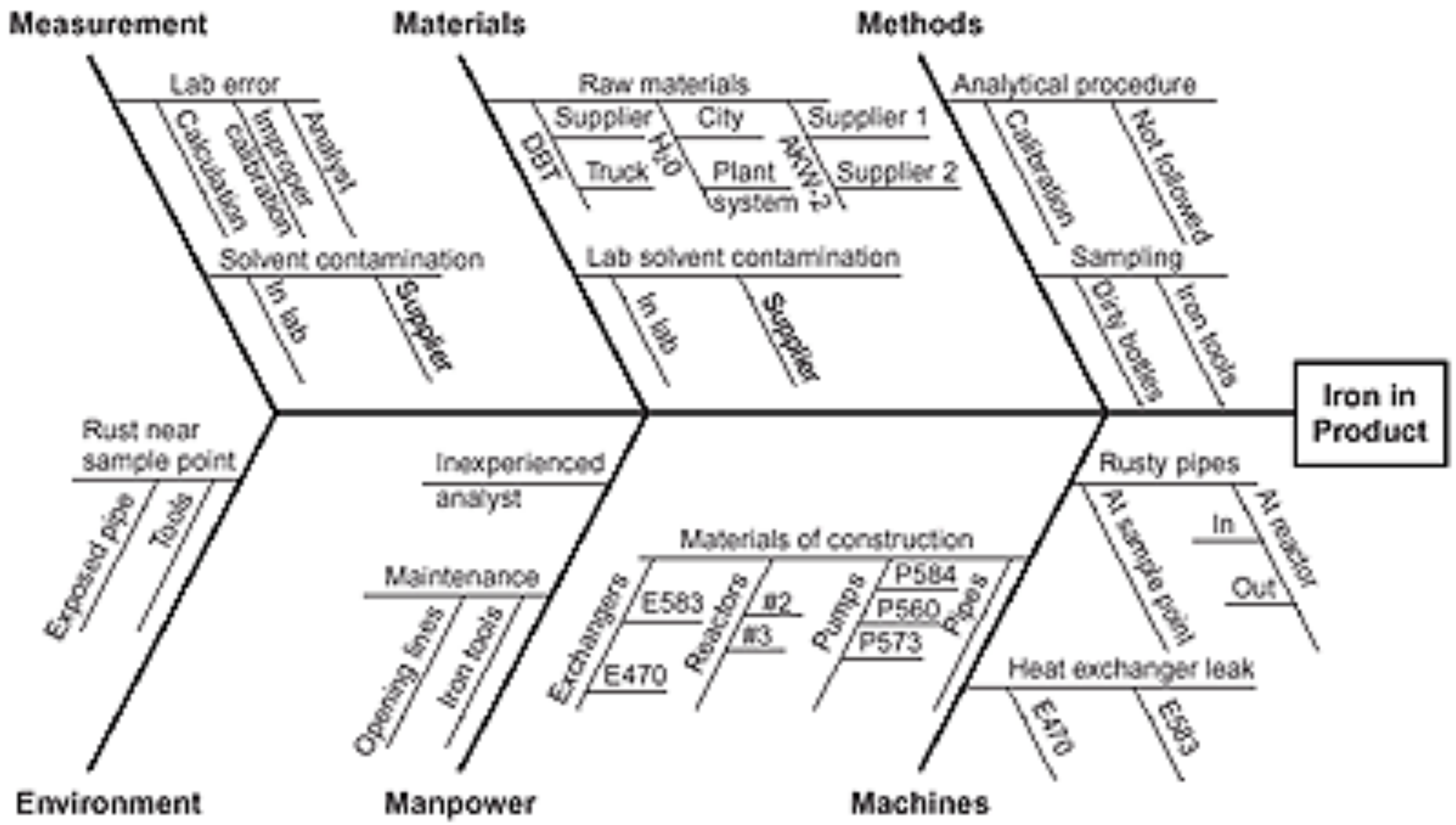
RCA techniques

- Failure Modes and Effects Analysis (FMEA) forward-chains, starting with elements that can fail and cascades into the undesirabl e effects that could result from those failures. This contrasts with the backwards-chaining approaches above that start from an undesirable outcome and work backwards toward causes;

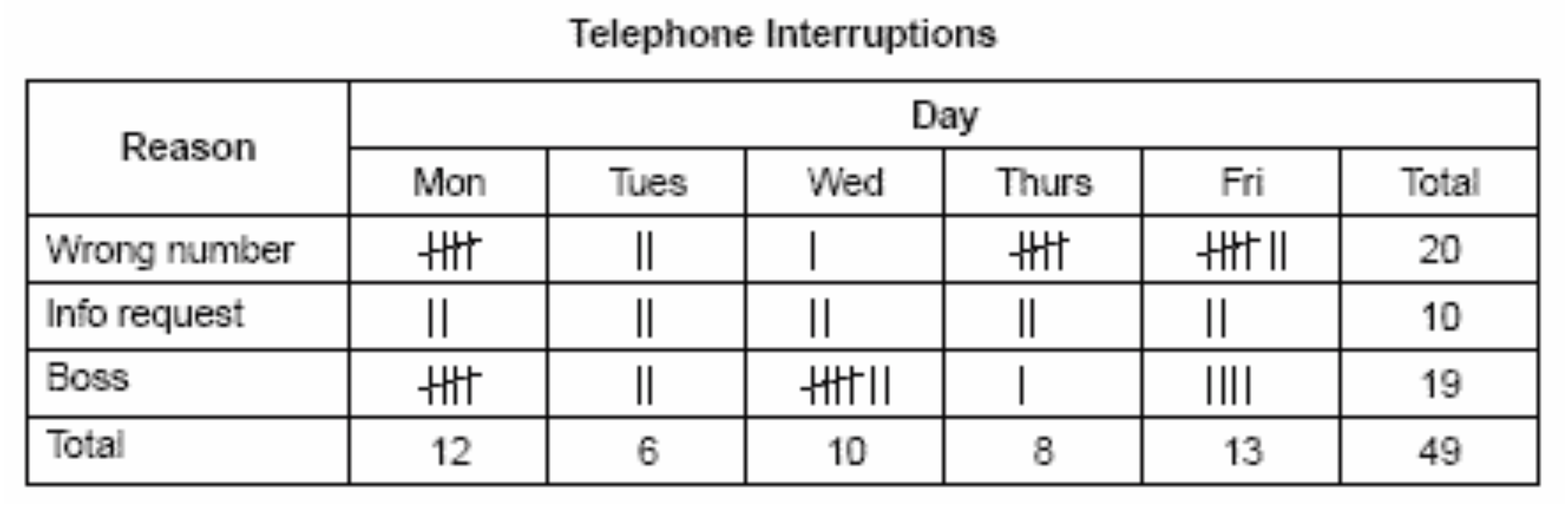
- Cause Map [16] is a structured map of cause-effect relationships that includes an undesirable outcome along with 1) chaining backwards to driving causes and 2) chaining forward into effects on organizational goals. Cause maps demand evidence of the occurrence of causes and the causality of effects and are thus more rigorous than Cause-and-Effect diagrams, FTA, and FMEA;

- Current Reality Tree [17] is a cause-effect tree bound by the rules of logic (i.e., Categories of Legitimate Reservation);

- Human Performance Evaluation posits that human performance depends on 1) input detection, 2) input understanding, 3) action selection, and 4) action execution. An undesirable outcome that results from human performance can be identified from a comprehensive list of potential drivers that includes, for example, cognitive overload, cognitive underload (i.e., boredom), memory lapse, tunnel vision or lack of a bigger picture, complacency, fatigue, etc.


Measurement
Materials
Methods
Lab error
Analyst
Improper calibration
Calculation
Solvent contamination
In lab
Supplier
Raw materials
Supplier
City
Supplier 1
DBT
H_2O
Truck
Plant system
AKW-2
Supplier 2
Lab solvent contamination
In lab
Supplier
Analytical procedure
Calibration
Not followed
Sampling
Dirty bottles
Iron tools
Iron in Product
Rust near sample point
Exposed pipe
Tools
Inexperienced analyst
Maintenance
Opening lines
Iron tools
Materials of construction
Exchangers
E583
E470
Reactors
#2
#3
Pumps
P584
P560
P573
Pipes
Rusty pipes
At sample point
In
Out
At reactor
Heat exchanger leak
E470
E583
Environment
Manpower
Machines

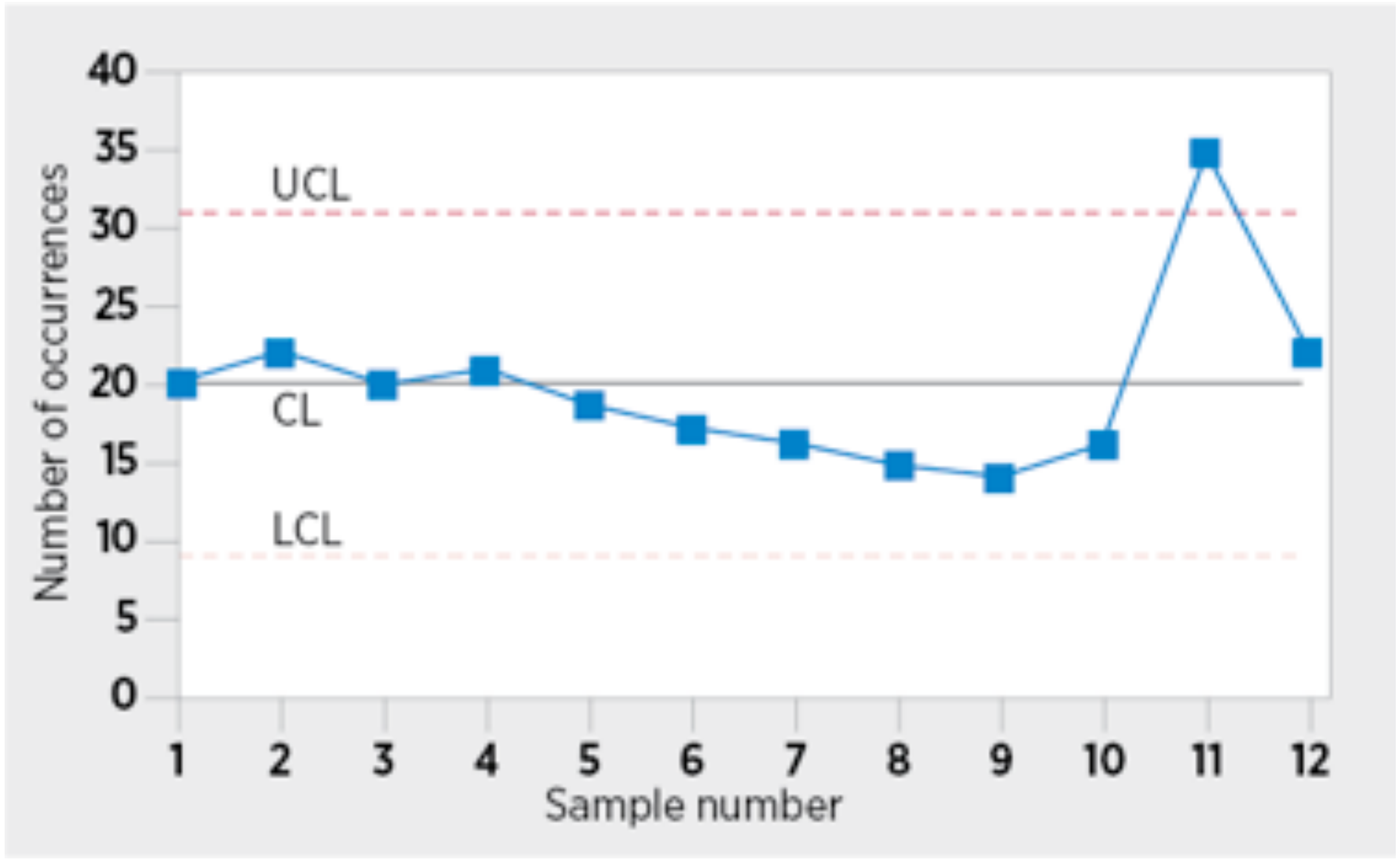
Fishbone



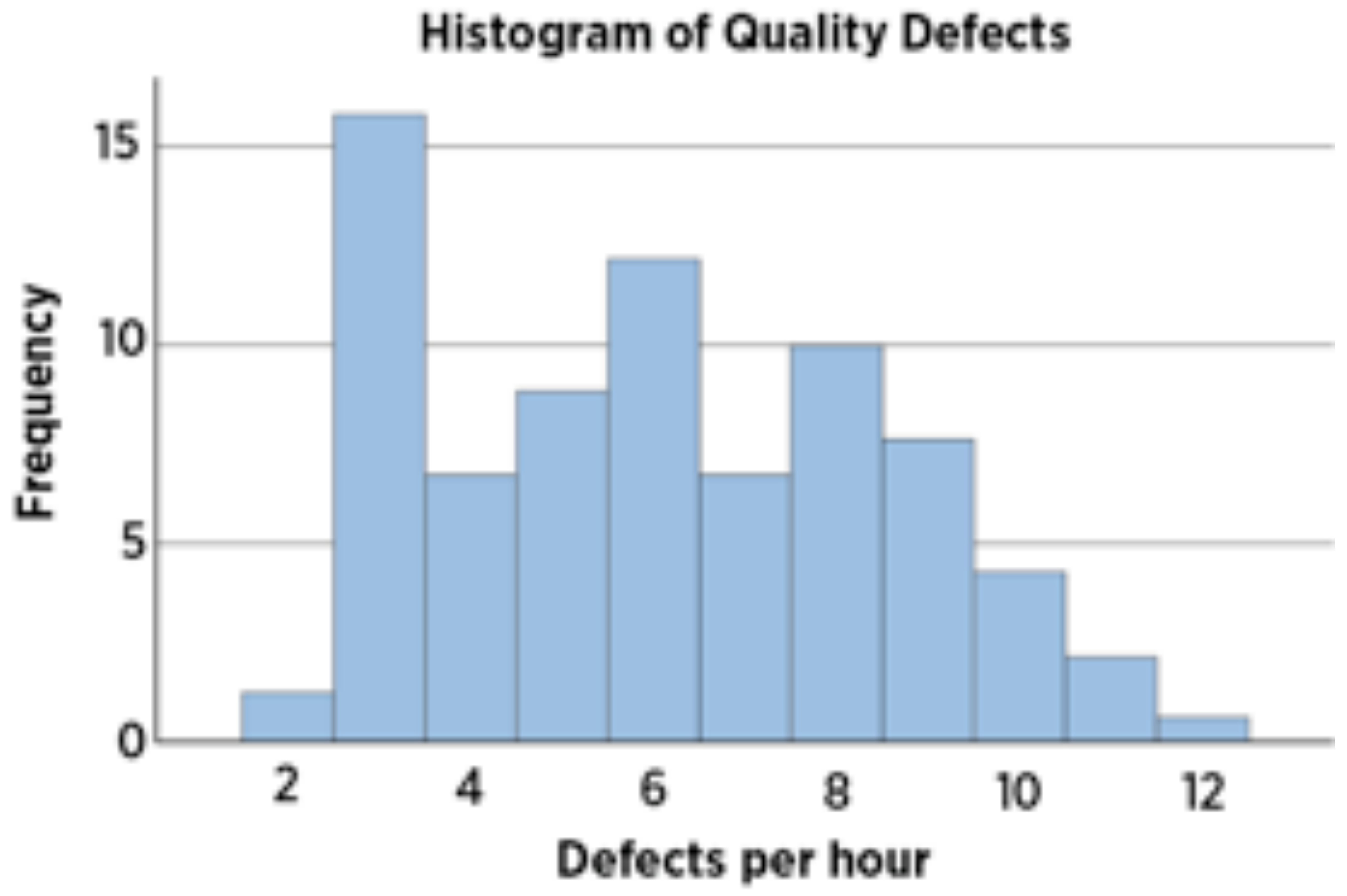
Telephone Interruptions

Reason	Day					
	Mon	Tues	Wed	Thurs	Fri	Total
Wrong number	𝍸	||	|	𝍸	𝍸 ||	20
Info request	||	||	||	||	||	10
Boss	𝍸	||	𝍸 ||	|	||||	19
Total	12	6	10	8	13	49

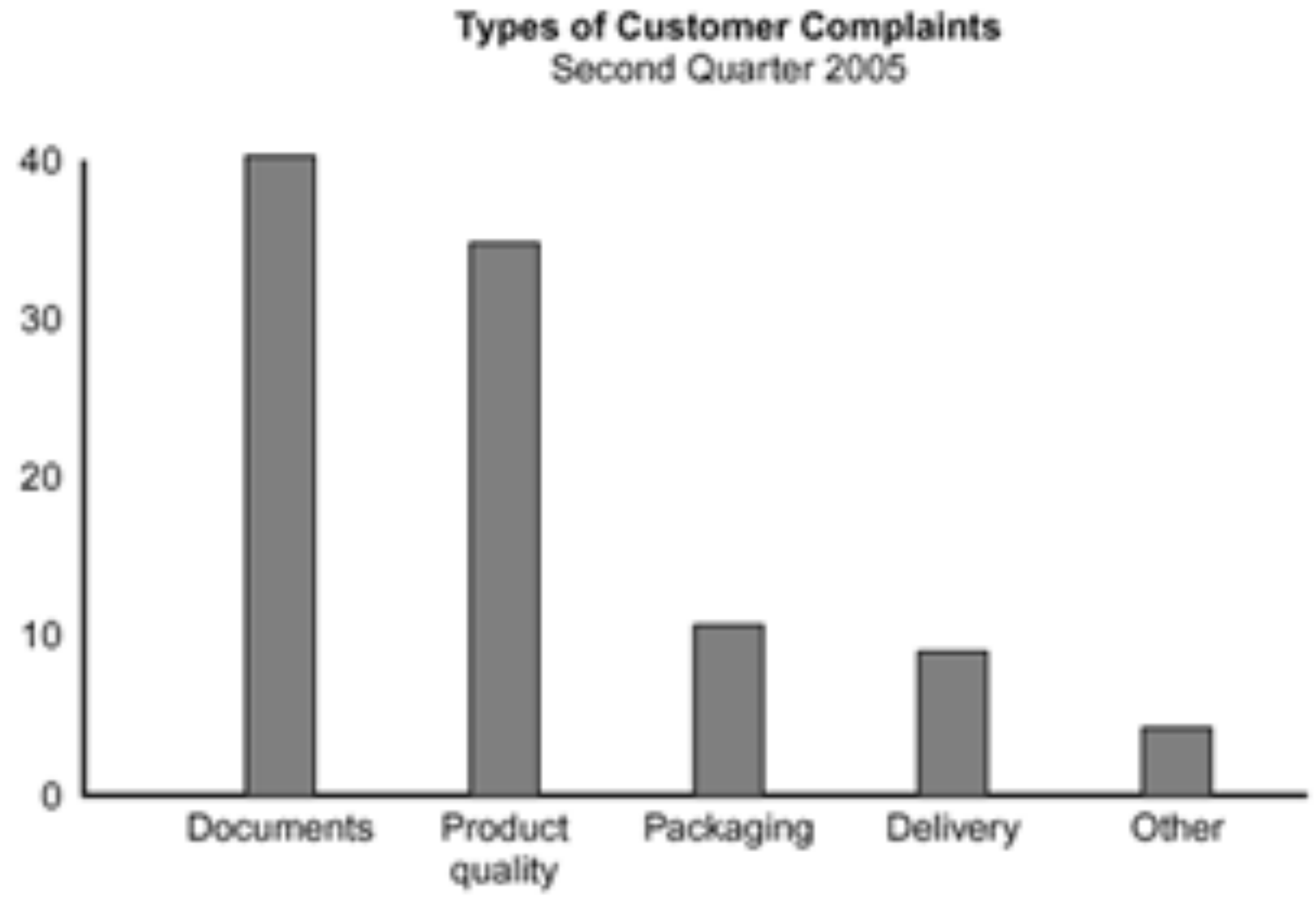
Check sheet



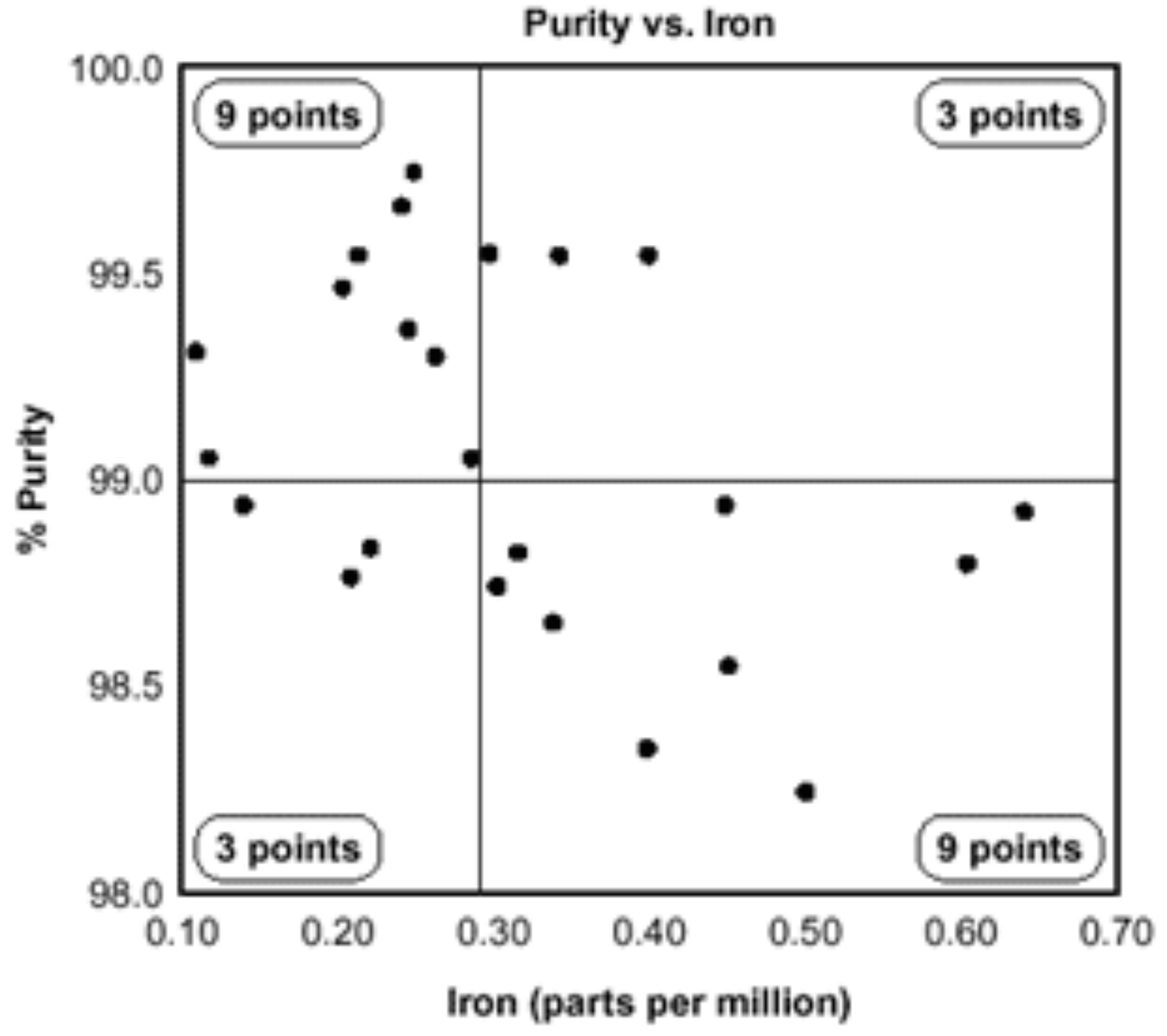
Control Chart


Histogram of Quality Defects
Frequency
15
10
5
0
2
4
6
8
10
12
Defects per hour

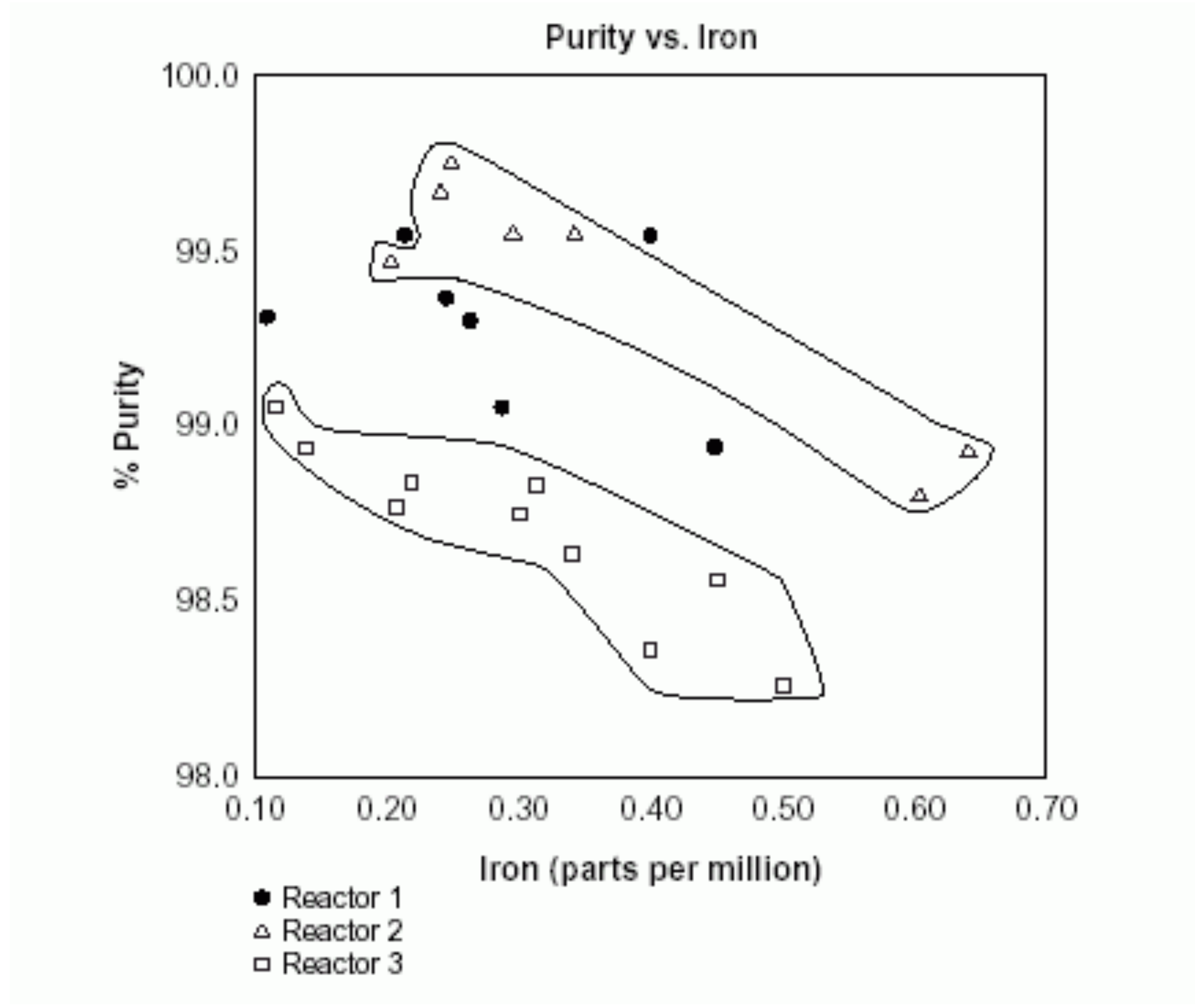
Histogram



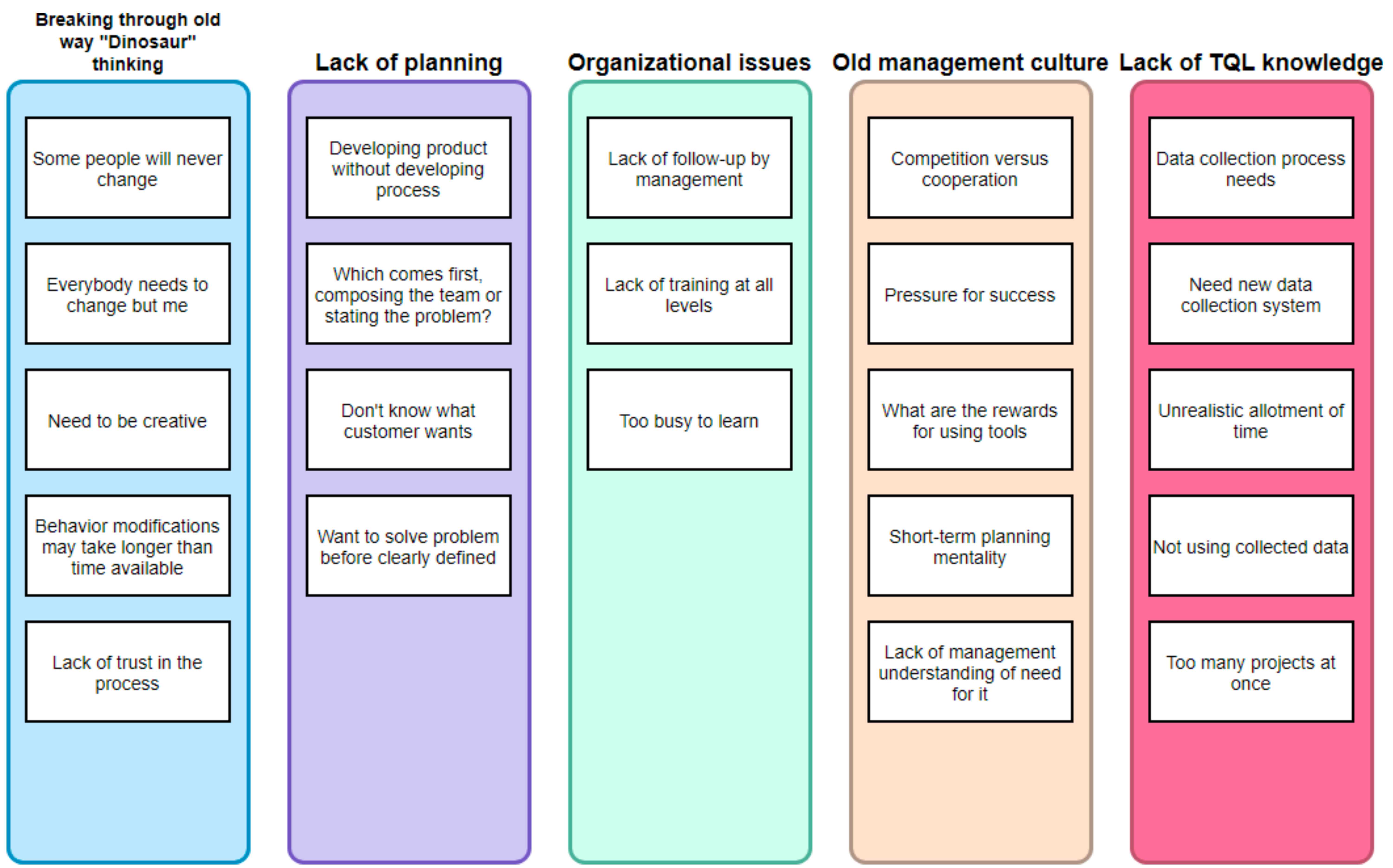
Pareto Chart



Scatter Diagram

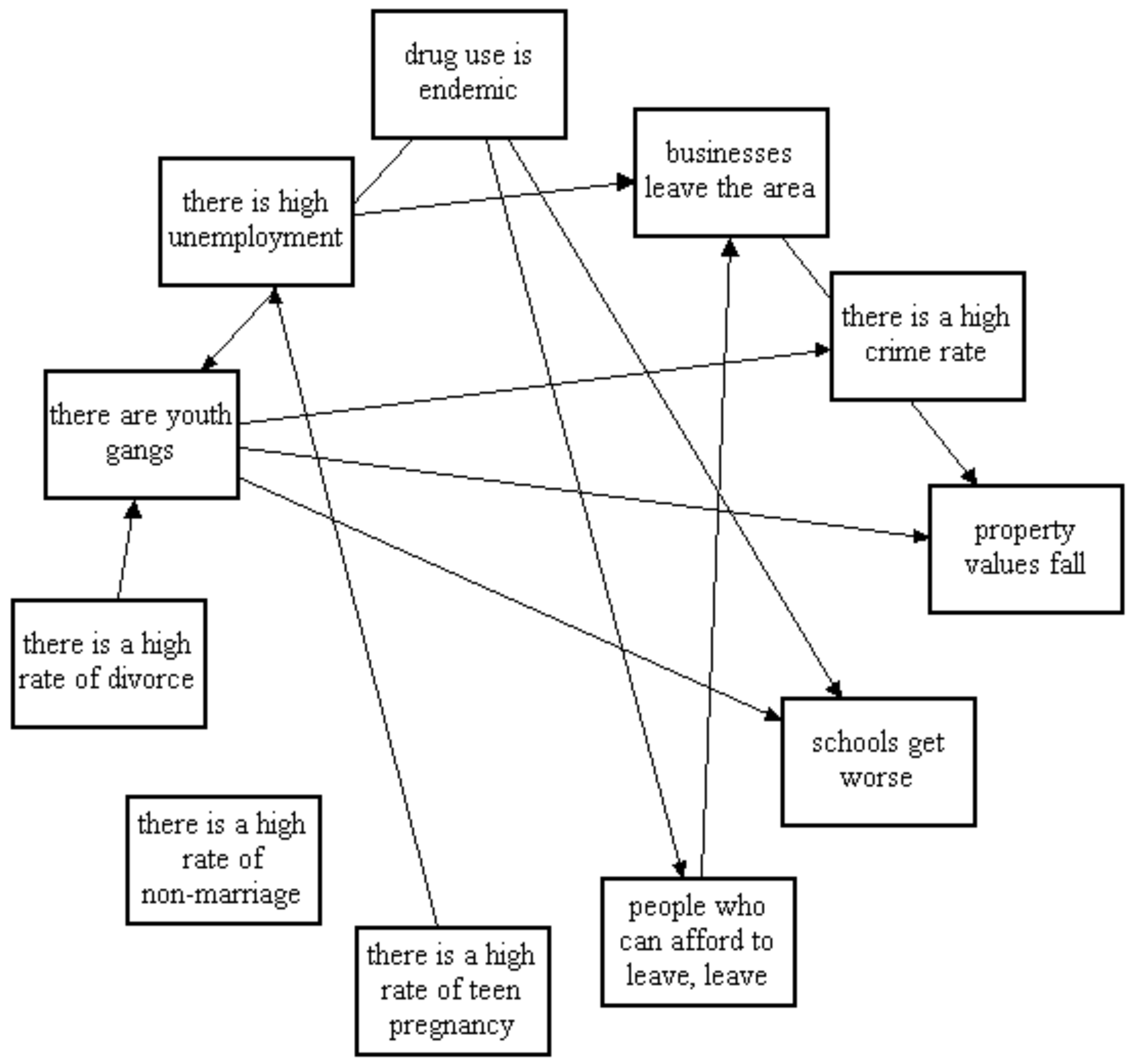


Stratification Diagram

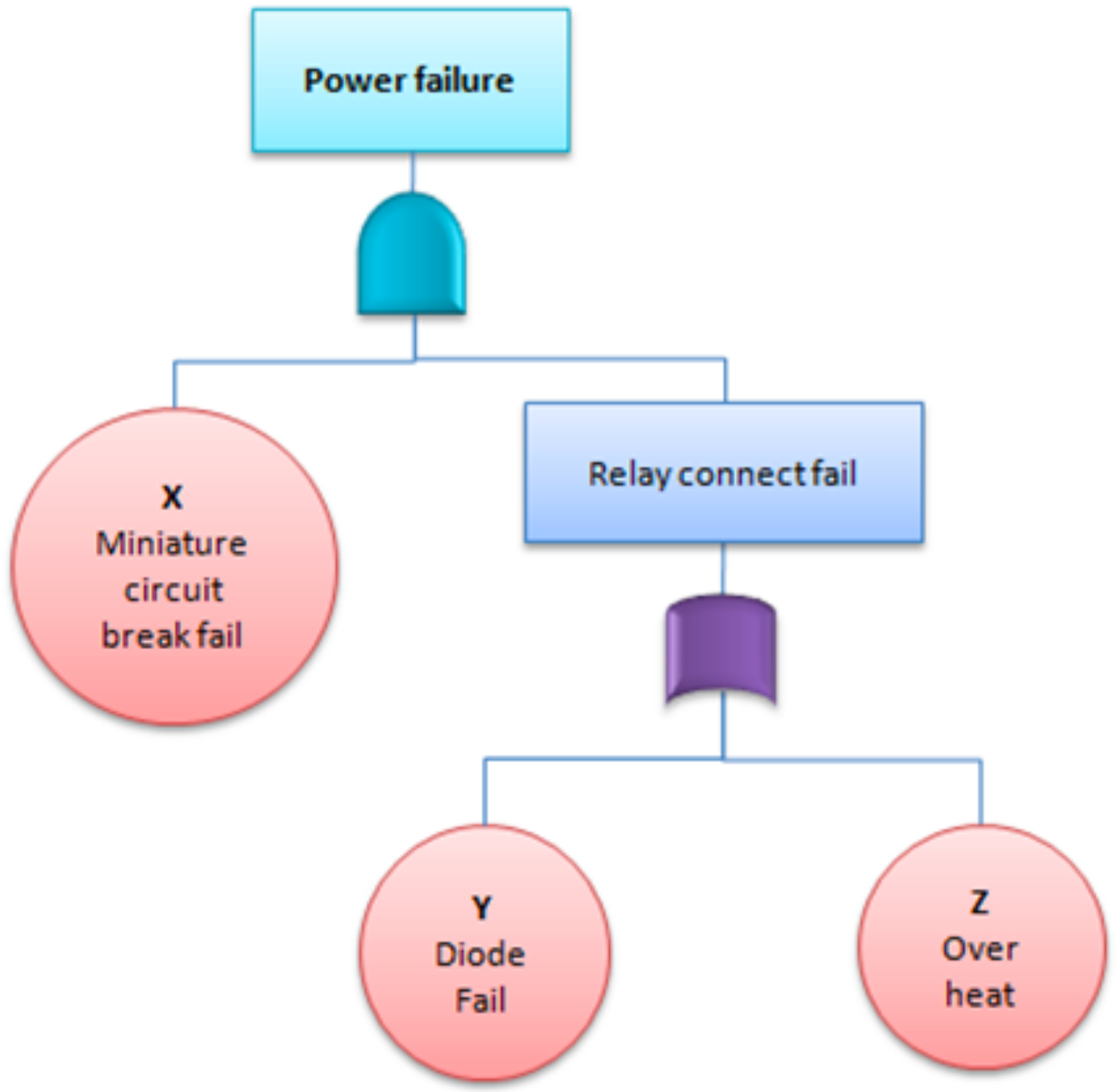


Breaking through old way "Dinosaur" thinking	Lack of planning	Organizational issues	Old management culture	Lack of TQL knowledge
Some people will never change	Developing product without developing process	Lack of follow-up by management	Competition versus cooperation	Data collection process needs
Everybody needs to change but me	Which comes first, composing the team or stating the problem?	Lack of training at all levels	Pressure for success	Need new data collection system
Need to be creative	Don't know what customer wants	Too busy to learn	What are the rewards for using tools	Unrealistic allotment of time
Behavior modifications may take longer than time available	Want to solve problem before clearly defined		Short-term planning mentality	Not using collected data
Lack of trust in the process			Lack of management understanding of need for it	Too many projects at once

Affinity diagram


drug use is endemic
businesses leave the area
there is high unemployment
there is a high crime rate
there are youth gangs
property values fall
there is a high rate of divorce
schools get worse
there is a high rate of non-marriage
people who can afford to leave, leave
there is a high rate of teen pregnancy

Relation Diagram



Fault Tree Analysis

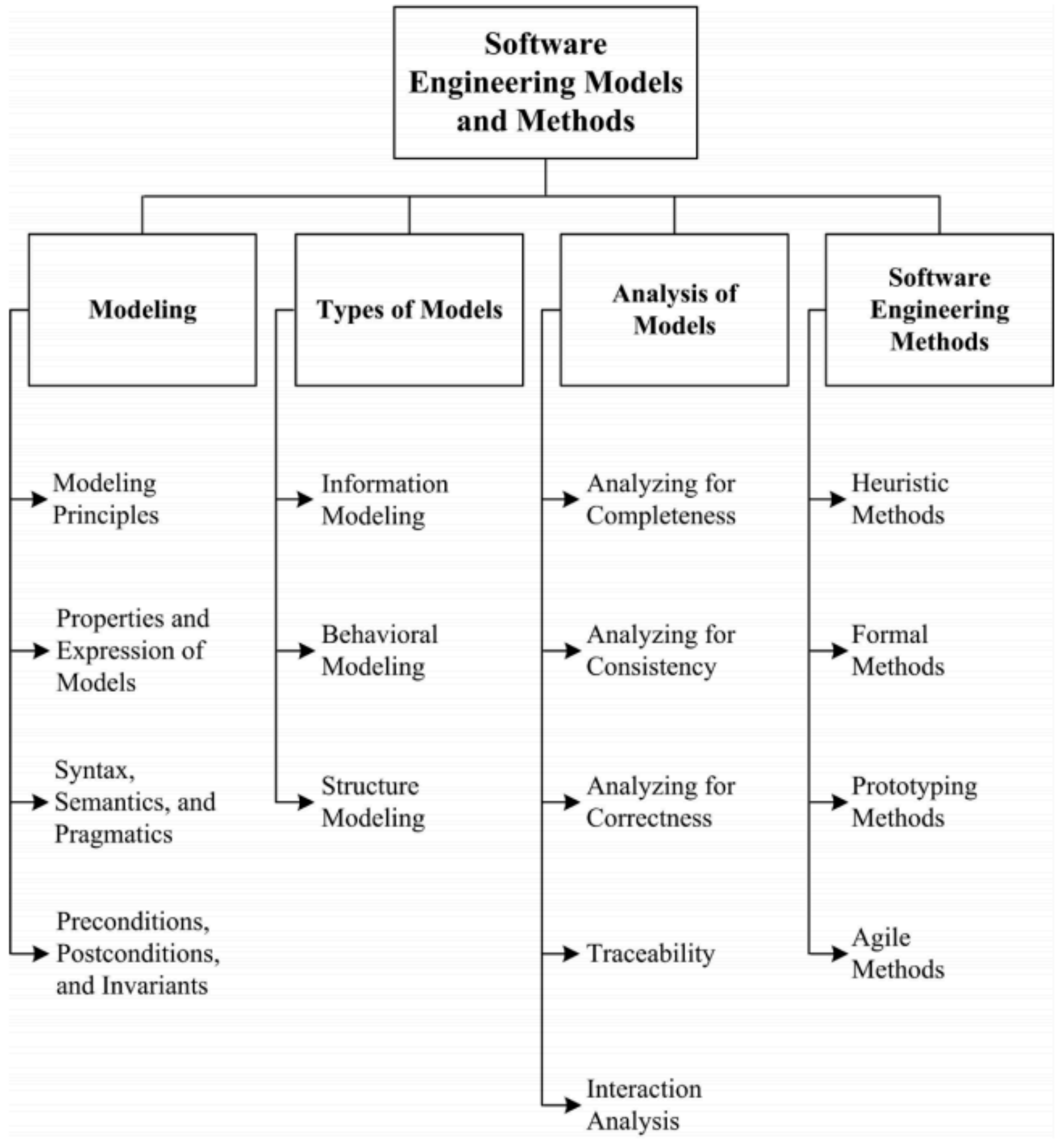

Software Engineering Models and Methods
Modeling
Types of Models
Analysis of Models
Software Engineering Methods
Modeling Principles
Properties and Expression of Models
Syntax, Semantics, and Pragmatics
Preconditions, Postconditions, and Invariants
Information Modeling
Behavioral Modeling
Structure Modeling
Analyzing for Completeness
Analyzing for Consistency
Analyzing for Correctness
Traceability
Interaction Analysis
Heuristic Methods
Formal Methods
Prototyping Methods
Agile Methods

Figure 9.1. Breakdown of Topics for the Software Engineering Models and Methods KA

Software Engineering Models and Methods

Modeling

- Modeling Principles
  - Model the Essentials
  - Provide Perspective
  - Enable Effective Communication

Modeling

- Properties and Expression of Models
  - Completeness:
    - the degree to which all requirements have been implemented and verified within the model.
  - Consistency:
    - the degree to which the model contains no conflicting requirements, assertions, constraints, functions, or component descriptions.
  - Correctness:
    - the degree to which the model satisfies its requirements and design specifications and is free of defects.

Modeling

- Syntax,Semantics, and Pragmatics
  - Modeling languages are defined by syntactic and semantic rules.
    - For textual languages, syntax is defined using a notation grammar that defines valid language constructs (for example, Backus-Naur Form (BNF)).
    - For graphical languages, syntax is defined using graphical models called metamodels.
  - Semantics for modeling languages specify the meaning attached to the entities and relations captured within the model. For example, a simple diagram of two boxes connected by a line is open to a variety of interpretations. Knowing that the diagram on which the boxes are placed and connected is an object diagram or an activity diagram can assist in the interpretation of this model.
  - Meaning is communicated through the model even in the presence of incomplete information through abstraction; pragmatics explains how meaning is embodied in the model and its context and communicated effectively to other software engineers.

Modeling

- Preconditions, Postconditions, and Invariants
- When modeling functions or methods, the soft- ware engineer typically starts with a set of assumptions about the state of the software prior to, during, and after the function or method exe- cutes.
  - Preconditions: a set of conditions that must be satisfied prior to execution of the function or method. If these preconditions do not hold prior to execution of the function or method, the function or method may produce erroneous results.
  - Postconditions: a set of conditions that is guaranteed to be true after the function or method has executed successfully. Typically, the postconditions represent how the state of the software has changed, how parameters passed to the function or method have changed, how data values have changed, or how the return value has been affected.
  - Invariants: a set of conditions within the operational environment that persist (in other words, do not change) before and after execution of the function or method. These invariants are relevant and necessary to the software and the correct operation of the function or method.

Types of Models

- Information Modeling
  - An information model is an abstract representation that identifies and defines a set of concepts, properties, relations, and constraints on data entities.
- Behavioral Modeling
  - Behavioral models identify and define the func- tions of the software being modeled.
  - Behavioral models generally take three basic forms: state machines, control-flow models, and data- flow models.
- Structure Modeling
  - Structure models illustrate the physical or logical composition of software from its various component parts.

Analysis of Models

- Analyzing for Completeness
    - Models may be checked for completeness by a modeling tool that uses techniques such as structural analysis and state-space reachability analysis (which ensure that all paths in the state models are reached by some set of correct inputs); models may also be checked for complete- ness manually by using inspections or other review techniques
- Analyzing for Consistency
    - Typically, consistency checking is accomplished with the modeling tool using an automated analysis function; models may also be checked for consistency manually using inspections or other review techniques
- Analyzing for Correctness
    - Analyzing for correctness includes verifying syntactic correctness of the model (that is, correct use of the modeling language grammar and constructs) and verifying semantic correctness of the model (that is, use of the modeling language constructs to correctly represent the meaning of that which is being modeled). To analyze a model for syntactic and semantic correctness, one analyzes it—either automatically (for example, using the modeling tool to check for model syntactic correctness) or manually (using inspections or other review techniques)—searching for possible defects and then removing or repairing the confirmed defects before the software is released for use.

Analysis of Models

- Traceability
  - Modeling tools typically provide some automated or manual means to specify and manage traceability links between requirements, design, code, and/or test entities as may be represented in the models and other software work products.
- Interaction Analysis
  - Interaction analysis focuses on the communications or control flow relations between entities used to accomplish a specific task or function within the software model. This analysis examines the dynamic behavior of the interactions between different portions of the software model, including other software layers (such as the oper- ating system, middleware, and applications).

Software Engineering Methods

- Heuristic Methods
  - Structured Analysis and Design Methods:
  - Data Modeling Methods:
  - Object-Oriented Analysis and Design Meth- ods
- Formal Methods
  - Specification Languages:
  - Program Refinement and Derivation:
  - Formal Verification:
  - Logical Inference:

Software Engineering Methods

- Prototyping Methods
    - Prototyping Style
    - Prototyping Target
    - Prototyping Evaluation Techniques
- Agile Methods
    - Rapid Application Development (RAD)
    - eXtreme Pro- gramming (XP)
    - Scrum
    - Feature-Driven Development (FDD)