Service-Oriented Software Engineering
Mariam Zenaishvili
NOTES

Introduction

Reduced competitiveness of software project:

- Cost overrun
- Time overrun

A typical aspect of SW products

Accidental difficulties

- Attitude
- Maintenance: the main purpose of software maintenance is to modify and update software applications after delivery to correct faults and to improve performance.
- Specification and design
- Teaming

Lifecycle (periods) = life of an SW product

1. Understand what the client wants => so you know what to build
2. build the software

planing (manegarial part)

Two types of testing:
– **Verification** (at the end of each phase)
– **Validation** (at the end of development, typically)

- **Verification** = are we building the product right?
    - "building it right" checks that the specifications are correctly implemented by the system
- **Validation** = are we building the right product?
    - "Building the right thing" refers back to the user's needs

The defect removal efficiency (DRE) gives a measure of the development team ability to remove defects prior to release.

DRE's formula: DRE= Number of defects found internally/ (Number of defects found internally + Number of defects found externally) × 100

For instance:

If the development team finds 900 defects before delivery and the users find 100 defects in a standard period after release (normally 90 days), then the DRE value is 90 percent

900/900+100 = 900/1000;
900/1000 = 9/10;
9/10 * 100 = 900/10;
900/10 = 90%

Typical Aspects of SW products

ESSENTIAL difficulties
– complexity: The complexity of software arises from a large number of unique interacting parts in a software system. The parts are unique because they are encapsulated as functions, subroutines, or objects, and invoked as needed rather than being replicated(გამეორება).

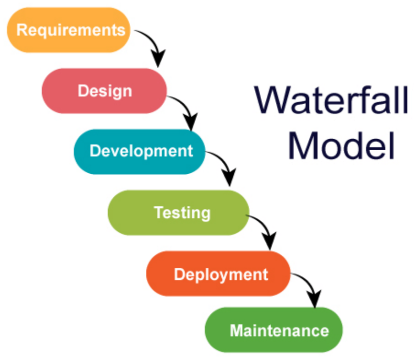
– conformity(შესაბამისობა წესებთან)**:** software must conform(შეესაბამება) to exacting specifications in the representation of each of its parts, in the interfaces to other internal parts, and in the connections to the environment in which it operates. Lack of conformity can cause problems when an existing software component cannot be reused as planned because it does not conform to the needs of the product under development.

– changeability: Complexity and the need for conformity can make changing software an extremely difficult task. Changing one part of a software system often results in undesired side effects in other parts of the system, requiring more changes before the software can operate at maximum efficiency.

– invisibility: Work products such as requirements specifications, design documents, source code, and object code are representations of software, but they are not the software. At the most elemental level, software resides in the magnetization and current flow in an enormous number of electronic elements within a digital device.

Cost proportional to the square of size: C=aS^2

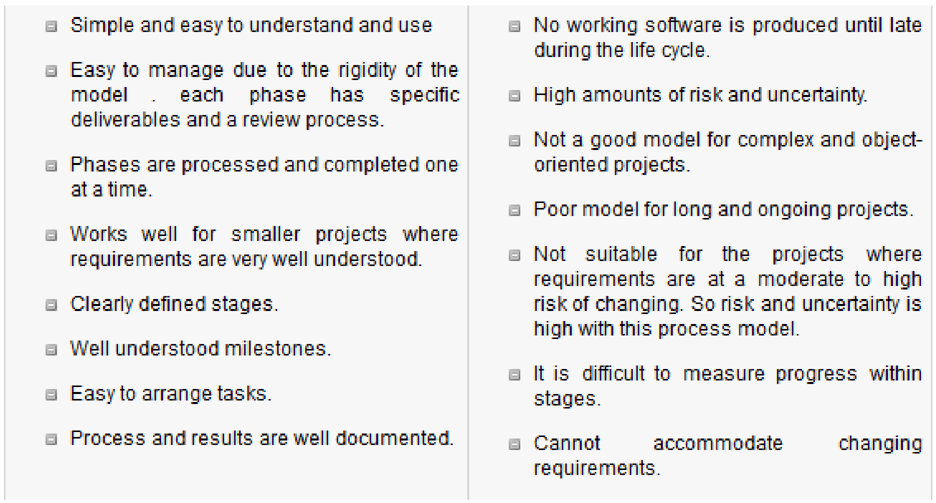
Software Process



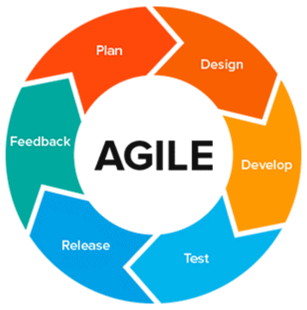
Waterfall Method VS Agile

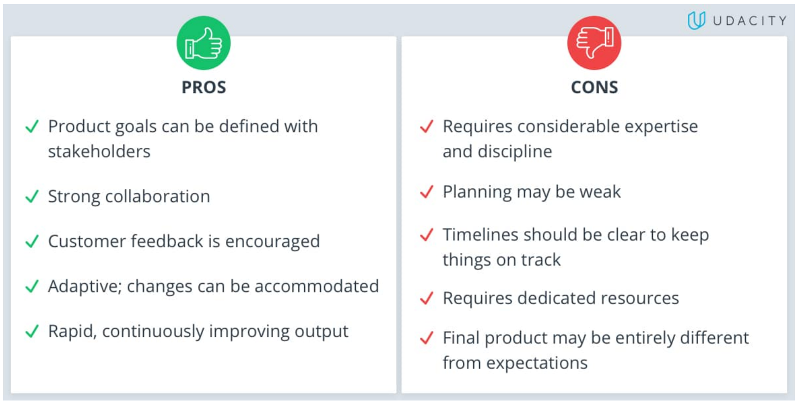
Waterfall:

- linear-sequential life cycle model: development process begins only if the previous phase is complete
- The phases do not overlap



⊟ Simple and easy to understand and use	⊟ No working software is produced until late during the life cycle.
⊟ Easy to manage due to the rigidity of the model . each phase has specific deliverables and a review process.	⊟ High amounts of risk and uncertainty.
⊟ Phases are processed and completed one at a time.	⊟ Not a good model for complex and object-oriented projects.
⊟ Works well for smaller projects where requirements are very well understood.	⊟ Poor model for long and ongoing projects.
⊟ Clearly defined stages.	⊟ Not suitable for the projects where requirements are at a moderate to high risk of changing. So risk and uncertainty is high with this process model.
⊟ Well understood milestones.	⊟ It is difficult to measure progress within stages.
⊟ Easy to arrange tasks.	⊟ Cannot accommodate changing requirements.
⊟ Process and results are well documented.	





UDACITY

PROS	CONS
✓ Product goals can be defined with stakeholders	✓ Requires considerable expertise and discipline
✓ Strong collaboration	✓ Planning may be weak
✓ Customer feedback is encouraged	✓ Timelines should be clear to keep things on track
✓ Adaptive; changes can be accommodated	✓ Requires dedicated resources
✓ Rapid, continuously improving output	✓ Final product may be entirely different from expectations

Verification and Validation Activities (V&V)

- Verification is intended to check that a product, service, or system meets a set of design specifications.
  - verification procedures involve regularly repeating tests devised specifically to ensure that the product, service, or system continues to meet the initial design requirements, specifications, and regulations as time progresses.
  - Verification can be in development, scale-up, or production.
  - Internal process.
- Validation is intended to ensure a product, service, or system results in a product, service, or system that meets the operational needs of the user.

    - For a new development flow or verification flow, validation procedures may involve modeling either flow and using simulations to predict faults or gaps that might lead to invalid or incomplete verification or development of a product.
    - External process.

Rapid Prototyping - a working model of a system developed to demonstrate and validate a customer's expectations of major functions and user interfaces.

- This technique offers a useful method of exploring ideas and getting customer feedback for each of them. In this method, a developed prototype need not necessarily be a part of the ultimately accepted prototype. Customer feedback helps in preventing unnecessary design faults and hence, the final prototype developed is of better quality.
- used when the customers do not know the exact project requirements beforehand. In this model, a prototype of the end product is first developed, tested and refined as per customer feedback repeatedly till a final acceptable prototype is achieved which forms the basis for developing the final product.
- the system is partially implemented before or during the analysis phase thereby allowing the customers to see the product early in the life cycle.
- This technique offers a useful method of exploring ideas and getting customer feedback for each of them. In this method, a developed prototype need not necessarily be a part of the ultimately accepted prototype. Customer feedback helps in preventing unnecessary design faults and hence, the final prototype developed is of better quality.
- Prototyping can be considered as a risk reduction activity that reduces requirements risks

Advantages

    - The customers get to see the partial product early in the life cycle. This ensures a greater level of customer satisfaction and comfort.
    - New requirements can be easily accommodated as there is scope for refinement.
    - Missing functionalities can be easily figured out.
    - Errors can be detected much earlier thereby saving a lot of effort and cost, besides enhancing the quality of the software.
    - The developed prototype can be reused by the developer for more complicated projects in the future.
    - Flexibility in design.

Disadvantages

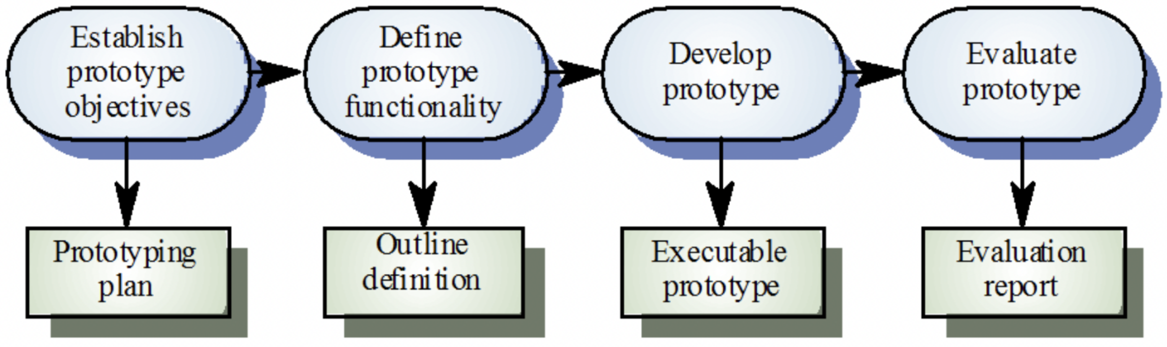
    - There may be too much variation in requirements each time the prototype is evaluated by the customer.

- Poor Documentation due to continuously changing customer requirements.
- It is very difficult for developers to accommodate all the changes demanded by the customer.
- There is uncertainty in determining the number of iterations that would be required before the prototype is finally accepted by the customer.
- After seeing an early prototype, the customers sometimes demand the actual product to be delivered soon.
- Developers in a hurry to build prototypes may end up with sub-optimal solutions.
- The customer might lose interest in the product if he/she is not satisfied with the initial prototype.

Use

The Prototyping Model should be used when the requirements of the product are not clearly understood or are unstable. It can also be used if requirements are changing quickly. This model can be successfully used for developing user interfaces, high technology software-intensive systems, and systems with complex algorithms and interfaces. It is also a very good choice to demonstrate the technical feasibility of the product.

Prototyping process

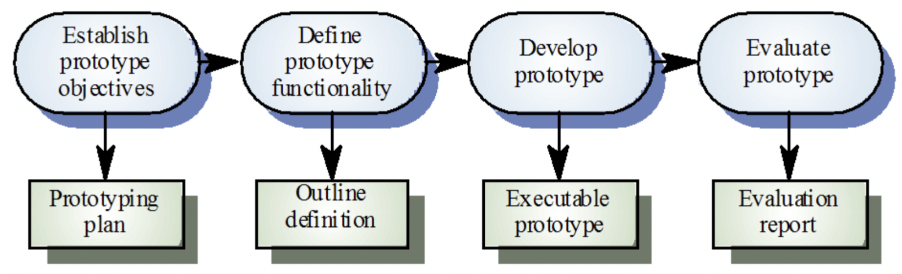


Throwaway prototyping

Throwaway prototyping is a part of the development of a prototype. When the prototype developers are concerned with a respective aspect of the prototype, they make throwaway prototypes to test it and get feedback from it. The end-users take the prototype, evaluate it, and provide feedback for the continuation, and addition within the developmental cycle.

The throwaway prototype model is just there to validate the system's functionalities and requirements. Once they're understood, and worked upon, throwaway prototypes are thrown away because they won't add any more advantage to the prototype.

You can't add them within the product, because they only check one aspect of it. The whole product will have tens or even hundreds of throwaway prototypes constructed alongside the project to check if it doesn't backfire.



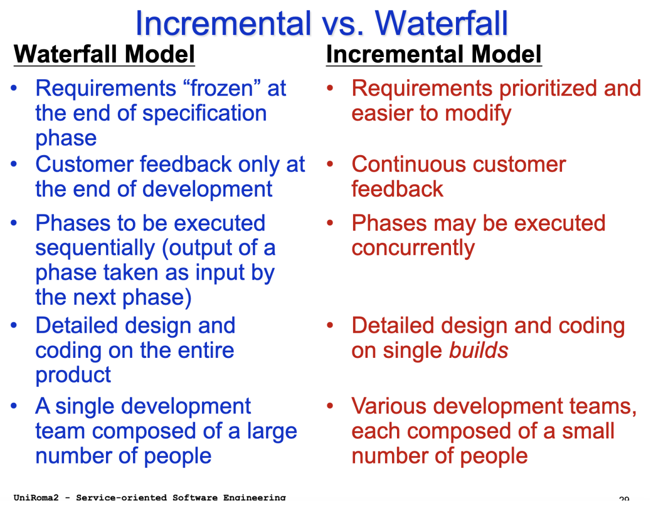
Throwaway delivery:

- Normal organizational quality standards may not have been applied
- Not documented

Process iteration

Incremental development

- Requirements and specifications for each increment may be developed
- effective when the customer wants to be kept up to date about the development advances, as well as when requirements change

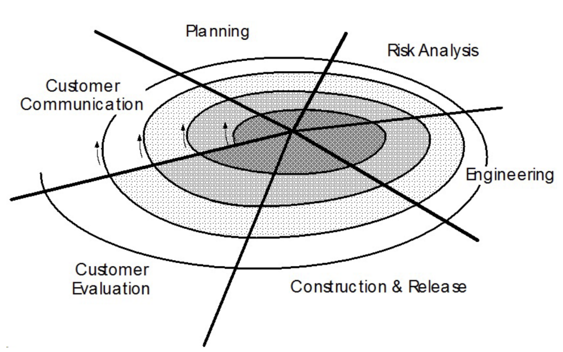


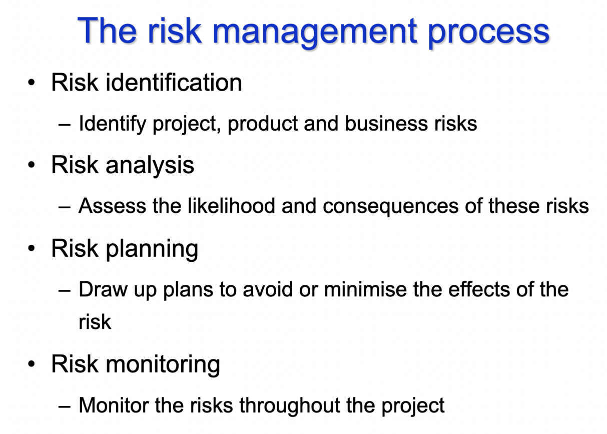
Incremental vs. Waterfall

Waterfall Model	Incremental Model
Requirements “frozen” at the end of specification phase	Requirements prioritized and easier to modify
Customer feedback only at the end of development	Continuous customer feedback
Phases to be executed sequentially (output of a phase taken as input by the next phase)	Phases may be executed concurrently
Detailed design and coding on the entire product	Detailed design and coding on single *builds*
A single development team composed of a large number of people	Various development teams, each composed of a small number of people

UniRoma2 - Service-oriented Software Engineering 20

Spiral Model





The risk management process

- Risk identification
  - Identify project, product and business risks
- Risk analysis
  - Assess the likelihood and consequences of these risks
- Risk planning
  - Draw up plans to avoid or minimise the effects of the risk
- Risk monitoring
  - Monitor the risks throughout the project

Object-oriented modeling (OOM) is an approach to modeling an application that is used at the beginning of the software life cycle when using an object-oriented approach to software development.

- **Communication**. Users typically cannot understand programming language or code. Model diagrams can be more understandable and can allow users to give developers feedback on the appropriate structure of the system. A key goal of the Object-Oriented approach is to decrease the "semantic gap" between the system and the real world by using terminology that is the same as the functions that users perform. Modeling is an essential tool to facilitate achieving this goal.
- **Abstraction**. A goal of most software methodologies is to first address "what" questions and then address "how" questions. I.e., first, determine the functionality the system is to provide without consideration of implementation constraints and then consider how to take this abstract description and refine it into an implementable design and code gave constraints such as technology and

budget. Modeling enables this by allowing abstract descriptions of processes and objects that define their essential structure and behavior.

The concurrent engineering model is intended to reduce development time and cost, by use of a systematic approach to the integrated and concurrent design of both the product and the associated process.

Model-based on formal methods: includes a set of activities that lead to the formal specification of the software product, to avoid ambiguity, incompleteness, and inconsistency and to make easier software verification by use of algebraic techniques and tools

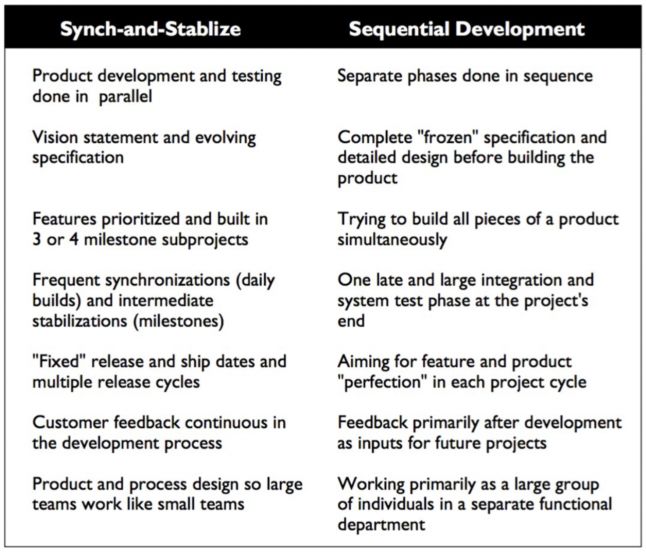
- A significant example is the Cleanroom Software Engineering(1987), which emphasizes the possibility of detecting defects earlier concerning conventional models

Synchronize-and-stabilize:

To start with, a vision statement for the product is prepared which gives the broad goals of the product. Based on this vision statement, and the inputs from market research, a document is prepared that lists out the important features to be incorporated, if necessary with prioritization (a draft specifications document). The development teams work in parallel on various modules and periodically integrate and test the code i.e., the code developed by different teams is 'synchronized'. Initially, the synchronization is done less frequently and at later stages very frequently, sometimes daily. These "periodic system builds" are tested for usability, functionality, and reliability; and feedback is given to the development teams. During the total planned development time, three or four milestones are defined—these milestones are used to 'stabilize' the product. Three typical milestones are alpha release, beta release, and final release as shown in Fig, This flexibility in the development approach coupled with early integration and testing gives a good feel of the product in the initial stages itself. Also, feature prioritization helps in releasing a product with the most important functionality and also helps in meeting the time target.

- continually synchronize what people are doing as individuals and as members of parallel teams (composed of 3 to 8 people) through the production of so-called daily builds, and –periodically
- stabilize the product in increments (or milestones) as a project proceeds, rather than once at the end of a project

synch-and-stabilize vs. waterfall



Synch-and-Stablize	Sequential Development
Product development and testing done in parallel	Separate phases done in sequence
Vision statement and evolving specification	Complete "frozen" specification and detailed design before building the product
Features prioritized and built in 3 or 4 milestone subprojects	Trying to build all pieces of a product simultaneously
Frequent synchronizations (daily builds) and intermediate stabilizations (milestones)	One late and large integration and system test phase at the project's end
"Fixed" release and ship dates and multiple release cycles	Aiming for feature and product "perfection" in each project cycle
Customer feedback continuous in the development process	Feedback primarily after development as inputs for future projects
Product and process design so large teams work like small teams	Working primarily as a large group of individuals in a separate functional department

Planing

- **Vision statement:** Management team <> customer -- get customers needs
- **Specification document:** defining the feature functionality and architecture
- **Schedule and feature team formation:** based on the specification document project management coordinates meeting with a team(3-8 people)

Development:

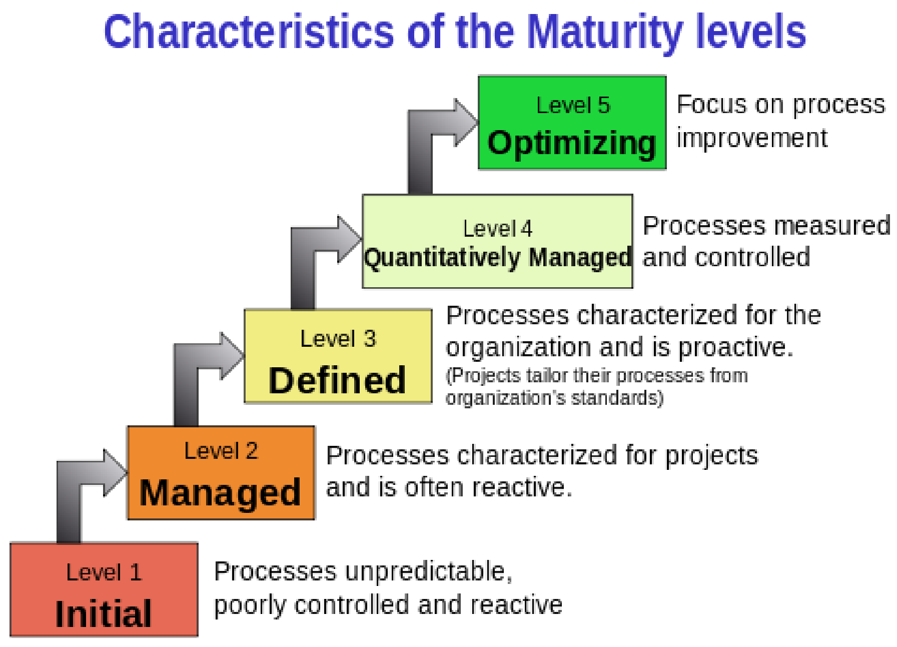
- Divided into 3 or 4 sequential subprojects that each results in a milestone release

Stabilization

- **Internal Testing:** done through testing the completed product within the company
- **External Testing:** done through testing the completed product outside the company by realizing the beta version
- **Release Preparation:** preparing the final version of the product and its documentation

A milestone is a specific point in time within a project lifecycle used to measure the progress of a project toward its ultimate goal.

Capability Maturity Model (CMM) - In 1993 the SEI(Software Engineering Institute) has developed a model to determine the maturity level of software development organizations (i.e., a global measure of effectiveness in the application of software engineering best practices).


Characteristics of the Maturity levels
Level 5
Optimizing
Focus on process improvement
Level 4
Quantitatively Managed
Processes measured and controlled
Level 3
Defined
Processes characterized for the organization and is proactive.
(Projects tailor their processes from organization's standards)
Level 2
Managed
Processes characterized for projects and is often reactive.
Level 1
Initial
Processes unpredictable, poorly controlled and reactive

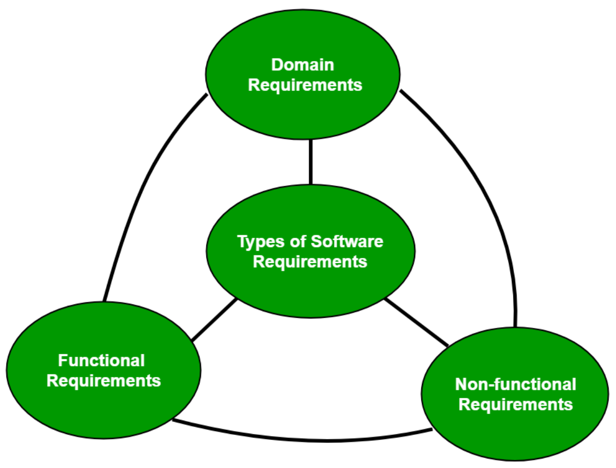
Software Requirements

According to IEEE standard 729, a requirement is defined as follows:

- A condition or capability needed by a user to solve a problem or achieve an objective
- A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents
- A documented representation of a condition or capability as in 1 and 2.

A software requirement can be of 3 types:

- Functional requirements
- Non-functional requirements
- Domain requirements



Functional requirements:

- statements of services the product should provide, how the product should react to particular inputs, and how the

product should behave in particular situations
- For example, in a hospital management system, a doctor should be able to retrieve the information of his patients. Or the user shall be able to search either all of the initial set of databases or select a subset from it

Nonfunctional requirements:
- These are the quality constraints that the system must satisfy according to the project contract. The priority or extent to which these factors are implemented varies from one project to another. They are also called non-behavioral requirements.
- requirements that specify that the delivered product must behave in a particular way, (e.g., execution speed, reliability, etc.)
- requirements which are a consequence of organizational policies and procedures (e.g., process standards used, implementation requirements, etc.)
- requirements that arise from factors that are external to the product and its development process (e.g., interoperability requirements, legislative requirements, etc.)
  - They deal with issues like:
    - Portability
    - Security
    - Maintainability
    - Reliability
    - Scalability
    - Performance
    - Reusability
    - Flexibility

Domain requirements:
- Domain requirements are the requirements that are characteristic of a particular category or domain of projects. The basic functions that a system of a specific domain must necessarily exhibit come under this category.
- For instance, in an academic software that maintains records of a school or college, the functionality of being able to access the list of faculty and list of students of each grade is a domain requirement. These requirements are therefore identified from that domain model and are not user-specific.
- Example There shall be a standard user interface to all databases which shall be based on the Z39.50 standard
- Example: .2Because of copyright restrictions, some documents must be deleted immediately on arrival. Depending on the user’s requirements, these documents will either be printed locally on the system server for manually forwarding to the user or routed to a network printer.

Software requirements problems

- **Ambiguity**(გაურკვევლობა): requirements that may be understood in various ways
    - Example 1: specify a time constraint without providing the time zone (in a product that is used to schedule and manage intercontinental calls)
- **Incompleteness**: the requirements do not include the whole set of important characteristics
- **Clashes or contradictions(შეტაკებები ან წინააღმდეგობები)** among requirements
    - For example each data entry form shall contain no more than 5 editable fields
- Non-functional requirements generically provided by the user (e.g., the product has to be easy-to-use) may turn out to be not quantifiable and thus hard to verify
- It is mandatory to specify non-functional requirements by use of a measure that eventually allows to quantitatively verify if the product meets or not those requirements

Software Requirements are descriptions of features and functionalities of the target system. Requirements convey the expectations of users from the software product. The requirements can be obvious or hidden, known or unknown, expected or unexpected from the client's point of view.

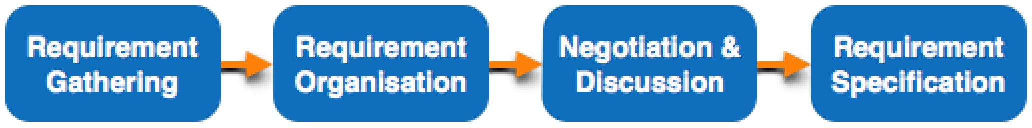
Requirement Engineering

Requirement Engineering is the process to gather the software requirements from clients, analyze and document them is known as requirement engineering.

- **Feasibility Study(მიზანშეწონილობა):** checks If the product contributes to organizational objectives
    - If the product can be engineered using current technology and within budget
    - If the product can be integrated with other products that are used
- **Requirement Gathering:** Analysts and engineers communicate with the client and end-users to know their ideas on what the software should provide and which features they want the software to include.
- **Software Requirement Specification:** SRS is a document created by a system analyst after the requirements are collected from various stakeholders. SRS defines how the intended software will interact with hardware, external interfaces, speed of operation, the response time of the system, portability of software

across various platforms, maintainability, speed of recovery after crashing, Security, Quality, Limitations, etc. The requirements received from the client are written in natural language. It is the responsibility of system analysts to document the requirements in technical language so that they can be comprehended and useful by the software development team.SRS should come up with the following features:

  - User Requirements are expressed in natural language.
  - Technical requirements are expressed in a structured language, which is used inside the organization.
  - Design description should be written in Pseudocode.
  - Format of Forms and GUI screen prints.
  - Conditional and mathematical notations for DFDs etc.

- **Software Requirement Validation:** Concerned with demonstrating that the requirements define the product that the customer wants. Requirements can be checked against the following conditions:
  - If they can be practically implemented
  - If they are valid and as per functionality and domain of software
  - If there are any ambiguities(გაურკვევლობა)
  - If they are complete
  - If they can be demonstrated
- **Requirement Elicitation Process**
- 

- **Requirements management -** managing changing requirements during the requirements engineering process.Classification of requirements in terms of evolution:
  - **enduring** requirements (low probability of change)
  - **volatile** requirements (high probability of change):
    - **mutable** requirements (requirements that change due to the system's environment)
    - **emergent(გადაუდებელი)** requirements (requirements that emerge as an understanding of the product improves)
    - **consequential** requirements (requirements that result from the introduction of the computer system)
    - **compatibility** requirements (requirements that depend on other systems or organizational processes)
- **Requirements management planning:** During the requirements engineering process, you have to plan:
  - **requirements**(unique) identification
  - **change impact analysis** => in terms of **costs** and **feasibility**

- ○ **traceability policies** => relationships between requirements, their sources, and the system design
- ○ **CASE tool support** => the tool support required to help manage requirements change

Object-oriented analysis and design(OOA)

Object-oriented analysis and design(OOA): defines **what** the software product is required to do.

Object-oriented design (OOD): defines **how** the product does what has been specified in the OOA phase

OOA and **OOD** provide a correct, complete, and consistent representation of:

- **Static and structural** aspects of the product (**data model**)
- **Functional** aspects of the product(**behavior model)**
- **The control** aspect of the product(**dynamic model**)

UML(Unified Modeling Language) has been introduced to unify the various notations and provide a standard language for modeling software systems. Its types are:

1. **Use case diagram:** describes how the users interact with the system, to show the relationship between the users and the different use cases in which they are involved
    a. **Relationships**:
        i. Association between actor and use case=>signifies basic communication or relationship
        ii. Extend between two use cases=>when base use is performed extended use case may happen, but not always. For example: log in and verify pass
        iii. Include between two use cases=>when base use case requires included use case to be done to perform. For example: log in and login error

        iv. Generalization of a use case(inheritance, parent)=>The behavior of the ancestor is inherited by the descendant. For example: make a payment and paying options(from checking, from savings)

2. **Class diagram**: describes the static structure of a system by showing the system's classes, their attributes, operations
3. **State diagram**: describes the control aspects of classes
4. **Activity diagram**: a sort of state diagram in which states represent actions, intended to model both computational and organizational processes
5. **Sequence diagram:** shows object interactions arranged in time sequence
6. **Collaboration diagram:** models the interactions between objects in terms of sequenced messages, putting into evidence the relations between objects
7. **Object diagram:** focuses on some particular set of objects and attributes, and the links between these instances at a specific time of execution
8. **Component diagram:** depicts how components are wired together to form larger components and or software systems
9. **Deployment diagram:** models the configuration of the hardware nodes that execute software components, along with the physical deployment of components on nodes

Data Model

- The first step of the data model building activity focuses on the identification of the so-called **entity classes**, i.e., those classes that contribute to defining the application domain and that are relevant to the system
- **Control classes**, which describe the application logic aspects, and **boundary classes** which describe the user interface aspects
- The set of operations of each class are identified once the behavioral model is available, and are thus omitted in the first iteration of the data model

Approaches for discovering classes

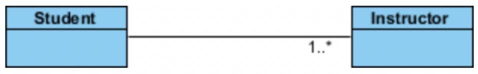
- **Noun phrase:** a phrase in which the nominal part is prevailing(ჭარბობს) concerning the verbal part. The nouns of the phrases used to specify user requirements are considered **candidate classes.**
  - Candidateclasses are then divided into three groups:
    - **Irrelevant**(not part of the application domain, can be skipped)
    - **Relevant**(for the application domain, can be entity classes)
    - **Fuzzy**(undecidable at the current time, to be analyzed later on, when additional information are available to decide their relevance)
- **Common class patterns:** Derives candidate classes from the classification theory of objects. It's just guidance. Only loosely bound to user requirements. possible classification pattern:
  - **Concept**(es. Reservation)

    - **Events**(es. Arrival)
    - **Organization**(es. AirCompany)
    - **People**(es. Passenger)
    - **Places**(es. TravelOffice)
- **Use case is driven:** use case diagrams (and possibly some high-level Sequence Diagrams) have been developed. narrative descriptions for each use case exist. Function-driven(or problem-driven) approach. Relies on the completeness and correctness of use case diagrams.
- **CRC(classes, responsibilities, collaborators):** Animated brainstorming sessions, in which each participant plays the role defined by the set of CRC cards he/she manages.
    - Identifies classes from the analysis of how objects collaborate to perform functions (e.g., use cases)
    - Suitable also for:
        - verification of classes discovered with other methods
        - determination of class properties
- **Mixed:** Based on elements from all four previous approaches. One possible scenario:
    - **Initial classes** - domain knowledge plus common class patterns approach to guide
    - **Noun phrase** approach and/or use case approach to add more classes
    - **CRC** to brainstorm and verify

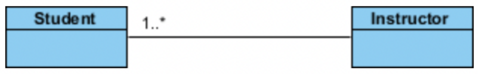
UML - Association

If two classes in a model need to communicate with each other, there must be a link between them, and that can be represented by an association (connector).

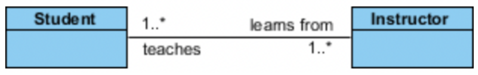
A single student can associate with multiple teachers:



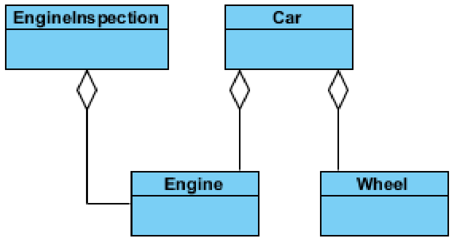
The example indicates that every Instructor has one or more Students:



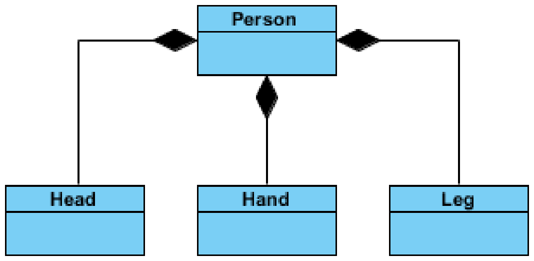
We can also indicate the behavior of an object in an association (i.e., the role of an object) using role names.



Aggregation and **Composition** are subsets of association meaning they are specific cases of association. In both aggregation and composition object of one class "owns" the object of another class. But there is a subtle difference:



- **Aggregation** implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.



- **Composition** implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate from a House.

UML - Generalization
In UML modeling, a generalization relationship is a relationship that implements the concept of object orientation called inheritance. The generalization relationship occurs between two entities or objects, such that one entity is the parent, and the other one is the child. The child inherits the functionality of its parent and can access as well as update it.

NOTE* generalization relation can either be used between actors or between use cases, but not between an actor and a use case

Interaction Diagrams
This interaction is a part of the dynamic behavior of the system. 2 types:

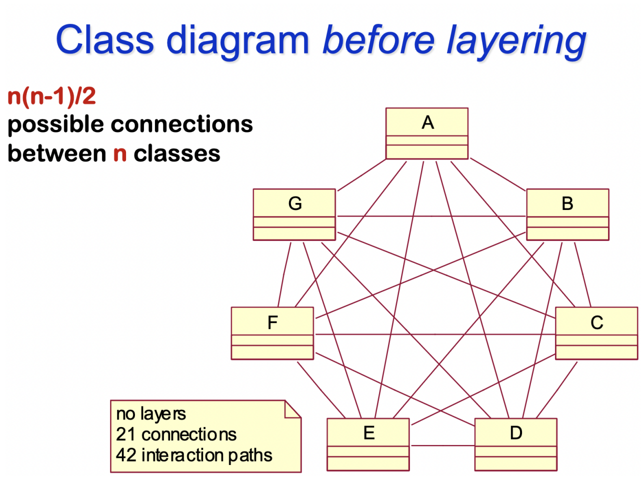
- **The sequence** diagram emphasizes the time sequence of messages
  - It is important to understand the time sequence of message flows. The message flow is nothing but a method call of an object. A message can be:
  - **Signal**=> Denotes asynchronous inter-object communication.
    - The sender continues executing after sending the signal message
  - **Call**=>Denotes synchronous invocation of an operation
    - The sender is blocked after sending the call message until it receives a return message
    - The return message can return some values to the caller or it can just acknowledge that the operation completed
- **The collaboration** diagram emphasizes the structural organization of the objects that send and receive messages.

Sequence diagram and **collaboration** diagram equivalent representations and can be automatically transformed into each other

Dynamic Model
The dynamic model is used to express and model the behavior of the system over time. It includes support for activity diagrams, state diagrams, sequence diagrams, and extensions including business process modeling.

OOA Model Complexity

- The complexity of OOA models has to be handled when products of large size are analyzed.
- The number of communication paths between classes grows exponentially with the addition of new classes
- Hierarchiesreduce the complexity from exponential to polynomial
    - introduce layers of classes and constrain intercommunication between layers
    - only classes in the same layer or adjacent layers are allowed to communicate directly



UML Package

- Represents a group of classes.
- Packages can be nested => An outer package has to access any classes directly contained in its nested packages
- A class can only be owned by one package
    - By declaring a class within a package to be private, protected, or public, we can control the communication and dependencies between classes in different packages

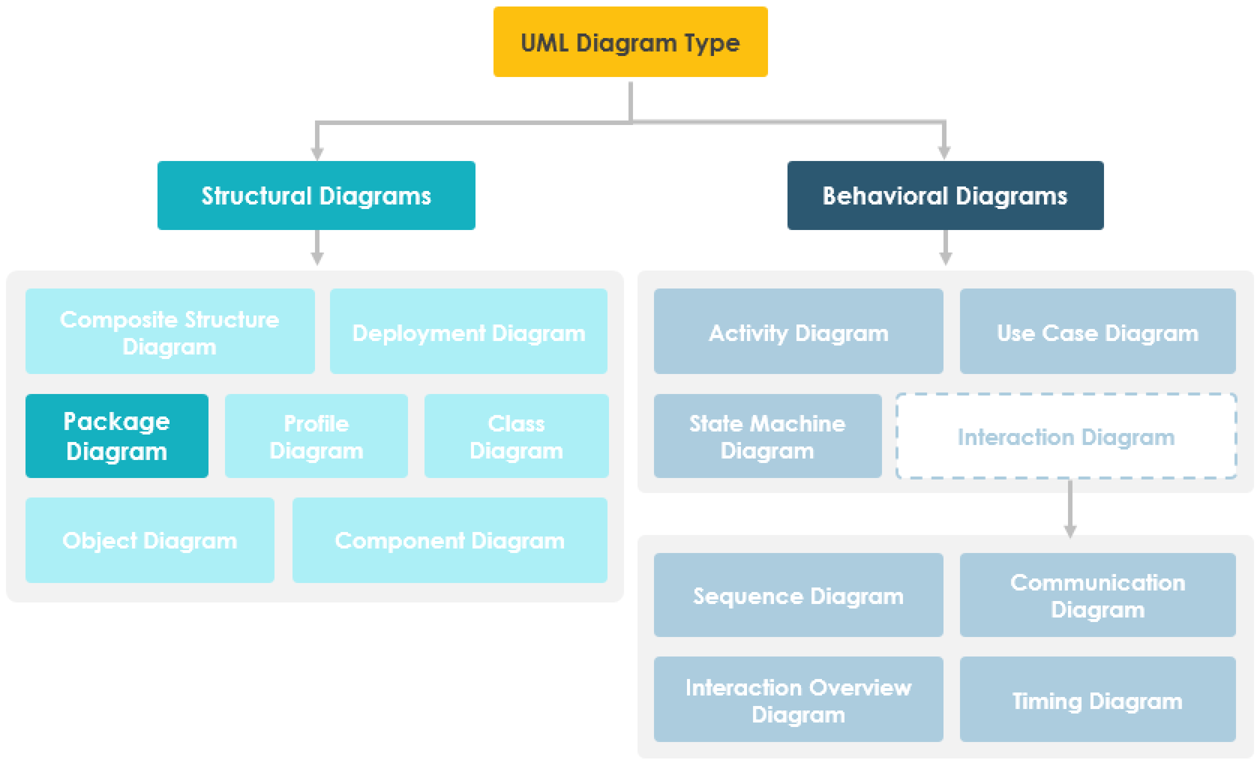
UML Package Diagram

- Package diagrams are used to structure high-level system elements. Packages are used for organizing large system which contains diagrams, documents, and other key deliverables.
- Packages are created in class diagram or use case diagram
- two kinds of relationships:
    - Dependency(usage dependency, access dependency, visibility dependency)
    - Generalization(implies dependency)

- Package Diagram can be used to simplify complex class diagrams, it can group classes into packages.
- A package is a collection of logically related UML elements.

Entity-control-boundary(BCE) approach

- organizes the responsibilities of classes according to their role in the use-case realization:
    - a boundary encapsulates interaction with external actors. Groups boundary classes that:
        - represent the interface between an actor and the product
        - have a visual display or produce sound effects
    - a control ensures the processing required for the execution of a use-case and its business logic, and coordinates, sequences control other objects involved in the use-case. Groups control classes that:
        - intercept user input events and control the execution of a business process
        - represent actions and activities of a use case
    - An entity represents long-lived information relevant to the stakeholders. Groups entity classes that
        - represent the semantics of entities in an application
        - correspond to data structures in the system’s database
        - instantiate objects persisting beyond the program’s execution



Planing

Software project management implies planning, monitoring, and controlling people, processes, and events during the software lifecycle.
The Software Project Management Plan(SPMP) is the document that drives the set of software project management activities

Four "P"

- **People**, the most important element of software success (in this respect SEI has developed the People Management -Capability Maturity Model)
- **Product**, which identifies the characteristics of the software to be built (objectives, data, functions, behavior, alternatives, constraints)
- **Process**, the reference frame within which the overall development and maintenance plan is established
- **The project**, which defines the set of activities to be carried out, along with resources, tasks, durations, and costs

Planning software projects

- Main elements:
    - **Scoping**: understanding the problem and the work to be carried out
    - **Estimation**: predicting time, cost, and effort
        - Has to be accurate to avoid uncertainty=>minimize estimation risks
        - **expert judgment by analogy** (estimation based on similar projects already completed)
            - **partitioning** techniques(bottom-up approach=>begins at the specific and moves to the general.)
                - **"divide et impera"** strategy and are based on:–**size estimation**, e.g., LOC (Lines Of Code) or FP(Function Point)
            - **algorithmic** models (based on resource consumption distributions e.g., Putnam or statistical modeling, historical model)
                - d = f(vi) where **d** is the value to be estimated (e.g., **effort**, **cost**, **duration**) and **vi** the independent variables (e.g., estimated **LOC** or **FP**)
    - **Risks**: defining risk management activities
    - **Schedule**: allocating the resources and identifying milestones throughout the project
    - **Control strategy**: defining a framework for quality assurance and change control

Function Point (FP) is a weighted measure of software functionality. Measures the amount of functionality in a system based upon the system specification (estimation before implementation)

FP = UFC(**Unadjusted Function point Count**) * TCF(**Technical Complexity Factor**)

FP count - Data categories

- **Several Internal Logical Files(ILF)**: A group of data or control information that is generated, used, or maintained by the software system.
- **Number of External Interface Files(EIF):** A group of data or control information passed or shared between applications

NOTE* file refers to a logically related group of data and not the physical implementation of those groups of data

FP count - Transaction categories

- **Number of External Inputs(EI):** Those items provided by the user that describe distinct application-oriented data, control information that enters an application, and changes the status of its internal logical file

- **Several External Outputs(EO):** All unique data or control information produced by the software systems, e.g., reports and messages.
- **Several External Inquiries(EQ):** All unique input/output combinations, where an input causes and generates an immediate output without changing any status of internal logical files. Example: spell checker

An algorithmic model: COCOMO

- **COCOMO**(**Constructive Cost Model**) is the algorithmic cost estimation model introduced by Boehm
- The model is used to first **estimate** the **development effort** and from that the development time and costs
- The key parameters which define the quality of any software products, which are also an outcome of the Cocomo are primarily **Effort & Schedule:**
  - **Effort**: Amount of labor that will be required to complete a task. It is measured in person-months units.
  - **Schedule**: Simply means the amount of time required for the completion of the job, which is, of course, proportional to the effort put in. It is measured in the units of time such as weeks, months.
- consists of **3 models**:
  - **Basic**(initial estimation)
  - **Intermediate**(used once the system is partitioned into a set of subsystems)
  - **Advanced**(used once each subsystem is partitioned into a set of modules)
- The effort **estimation** is obtained from product size estimation (in KLOC-)
  - estimation of the development mode, which measures the intrinsic level of development complexity, as:
    - organic(for small size products)
    - semidetached(for intermediate size products)
    - embedded(for large size complex products)

Project planning

Project planning deals with the organization of the set of tasks to be carried out to deliver the software product within the estimated time and cost budget.

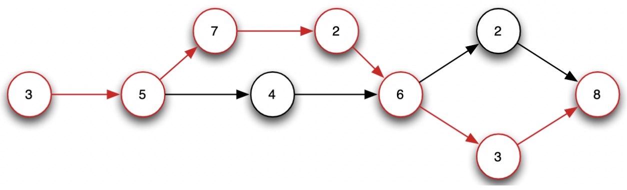
Planning tools

Task network (PERT-like)

– a *graph* in which *nodes* represent ***tasks*** and *arcs* represent ***precedence relationships***

– the task network allows one to obtain:

- the **critical path**, i.e., the sequence of tasks that determine the total calendar time required for the project (any time delays along the critical path will delay the entire project)
- the **expected time to complete a task**

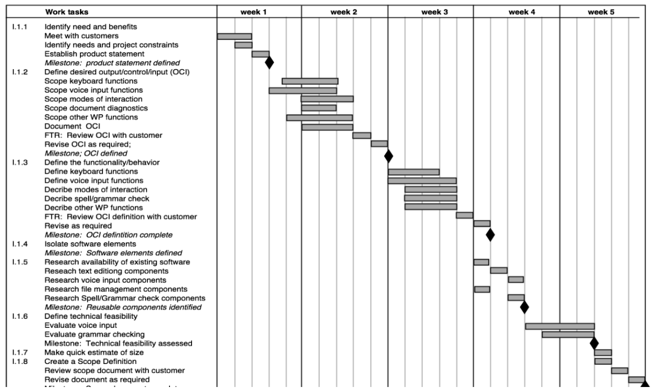


Planning tools (2)

Gantt chart

– *bar diagram* that shows the calendar time allocation of tasks

– does not give info about precedence relationships (thus it is typically supplemented by a *task network* diagram)



The Design Phase

Answers ***how*** that software must be developed through implementing with a code.

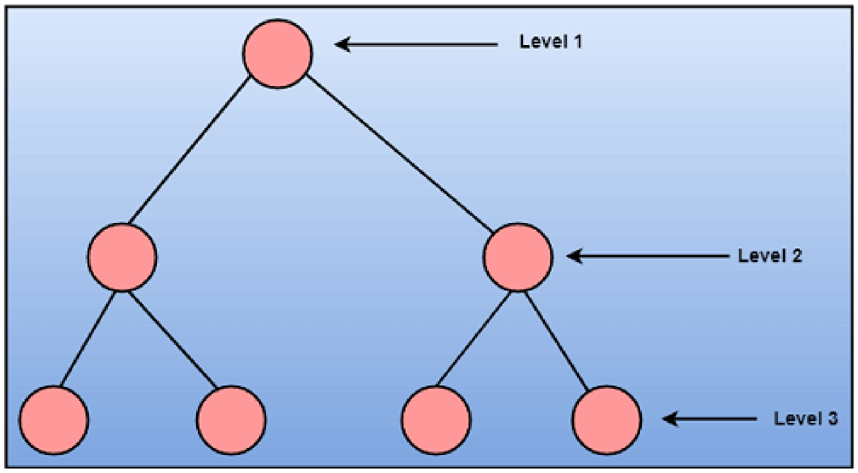
2 categories:

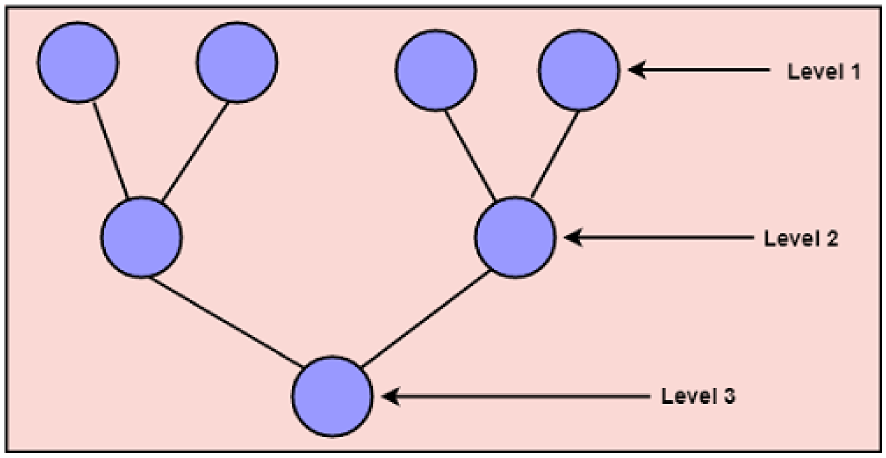
- **architectural** (or preliminary, or high-level)design, which operates the modular decomposition leading to the software architecture
- **detailed** design, which provides the detailed design, in terms of data structures and algorithms, of every single module of the software architecture

Design principles

Software design principles are concerned with providing means to handle the complexity of the design process effectively. Effectively managing the complexity will not only reduce the effort needed for design but can also reduce the scope of introducing errors during design.

- **Problem Partitioning**
- **refinement(დახვეწა) process through**
  - **top-down:** This approach starts with the identification of the main components and then decomposes them into their more detailed sub-components.

- 

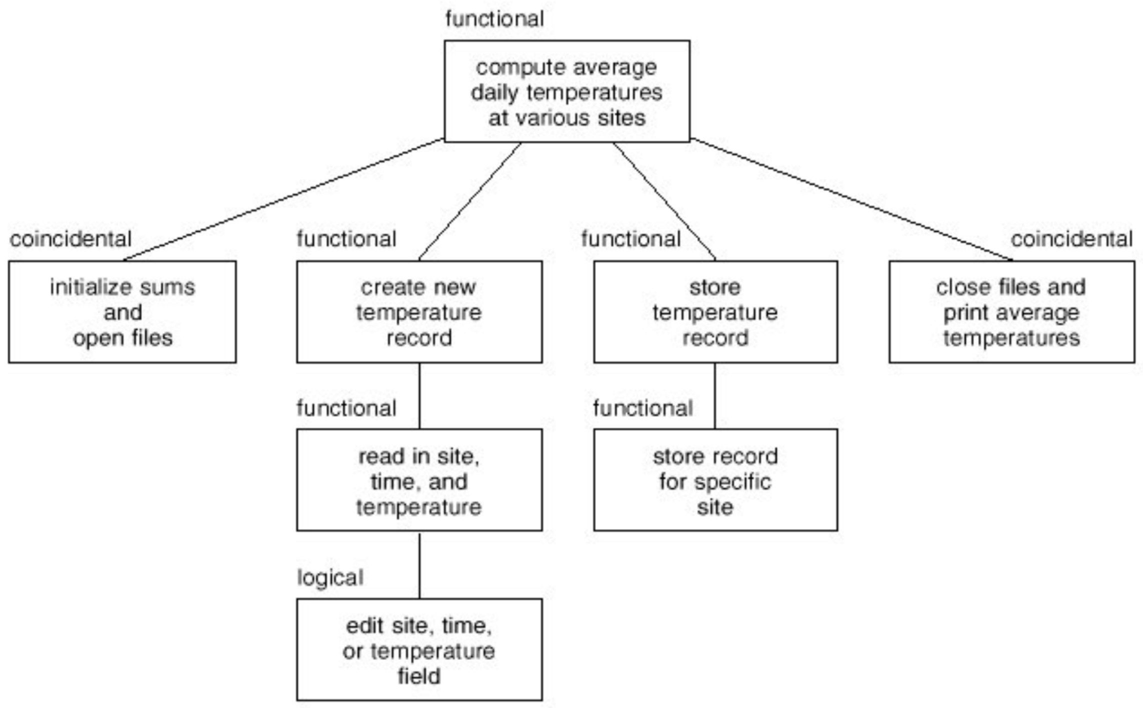
- **Bottom-up Approach:** begins with the lower details and moves towards up the hierarchy, as shown in fig. This approach is suitable in the case of an existing system.
- 

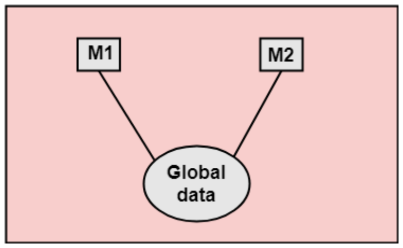
- **Abstraction**
  - Focuses on the essential aspects of a given entity, by suppressing(ჩახშობა) the more complex details
  - common abstraction mechanisms
    - **Functional Abstraction(**functions**)**
    - **Data Abstraction (**data structures**).**
  - A data structure along with the actions to be executed over it is referred to as providing data encapsulation
    - Using data encapsulation design time allows one to obtain significant advantages both during the coding phase and during maintenance
  - Using **Abstract Data Types**(ADTs)dentifies a data type whose instances provide data encapsulation. allows one to improve software quality, in terms of better levels of reusability and maintainability attributes. Example: C++ class

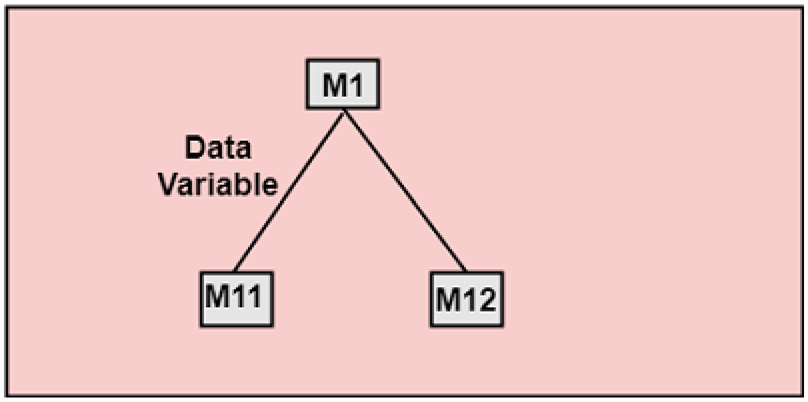
- **Modularity**
    - Modularity specifies the division of software into separate modules which are differently named and addressed and are integrated later on in to obtain the completely functional software.
    - So, a change to one component has minimal impact on the other components
    - Modular decomposition is based on the **“divide et impera”** principle
    - **Information hiding**
        - **Control and data abstraction** are derived from a more general concept, denoted as information hiding
        - **Information hiding** consists in defining and designing a module so that implementation details (both data and functions) that are not required to use that module are hidden, and thus not visible to other modules
    - **Functional independence** is achieved by developing functions that perform only one kind of task and do not excessively interact with other modules. Independence is important because it makes implementation more accessible and faster.
        - measured using two criteria:
            - **Cohesion**: It measures the relative function strength of a module.
                - a measure of the module capability to accomplish internally what is required to execute a function (i.e., without interacting with other modules)
                - **Cohesion levels(1is the worst, 7the best)**
                    - **Coincidental**(no relationship between module elements)
                    - **Logical**(logically related elements, characterized by the fact that only one is executed by the calling module)
                    - **Temporal**(temporally related elements, i.e., the elements are processed at a particular time in program execution)
                    - **Procedural**(elements related by the fact that are executed according to a predefined sequence).
                    - **Communicational**(elements related by the fact that are executed according to a predefined sequence and on a single data structure)

        - **Informational**(each element has a separate portion of code with associated input/output ports; all elements operate on the same data structure)

Example Structure Chart & Modules Cohesion


functional
compute average daily temperatures at various sites
coincidental
initialize sums and open files
functional
create new temperature record
functional
store temperature record
coincidental
close files and print average temperatures
functional
read in site, time, and temperature
functional
store record for specific site
logical
edit site, time, or temperature field

        - **Functional**(all the elements are related by the fact that execute a single function)
- **Coupling**: It measures the relative interdependence among modules. A measure of the degree of interaction between modules
- **Coupling levels** (1 is the worst, 5 the best):
    - **Content:** a module explicitly refers to the content of another module
    - **Common**: two modules that have complete access to the same data structure
    - 

            - **Control**: a module that explicitly(ცალსახად) controls the execution of another module
            - **Stamp**: a module that passes a data structure to another module, which uses some elements of the data structure only
            - **Data**: a module that passes an argument of simple type to another module, or a data structure for which all elements are used
            - 

        - **Factors affecting Coupling**
        - **Coupling** is measured by the number of relations between the modules.
    - **Modular decomposition** is considered good if obtains: Maximum cohesion internal modules and Minimum coupling between modules

Reusability
refers to the ability to use already available components in different products.

- **Advantages**
    - significant reductions in terms of software development cost and time
    - reliability growth, due to the use of already validated components
- Reusability at design time applies to
    - **software modules**
    - **application frameworks**, which incorporate the application logic of a design solution
    - **design patterns**, which identify design solutions to recurrent design problems
    - **software architectures**, which include (a), (b) and (c)

Object-Oriented Design - OOD

consists of the following two sub-phases:

- **preliminary** (or architectural, or system) OOD: defines the overall strategy to build a solution that solves the problem specified at OOA time. Decisions are taken that deal with the overall organization of the software (system architecture)
- **detailed** (or objects) OOD: provides the complete definition of classes and associations to be implemented, as well as the data structures and the algorithm of methods that implement class operations.

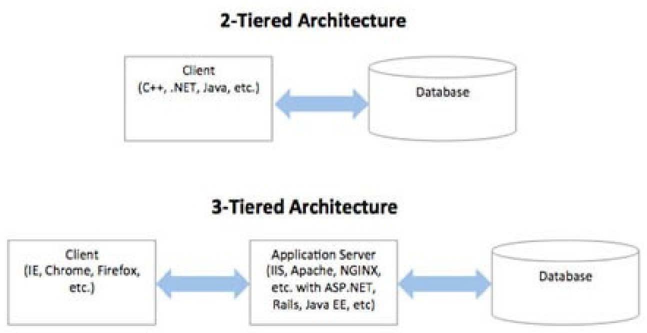
System architecture
defines the structure of the software system components along with the relationships between such components and the principles driving design and system evolution.

The term systems architecture has originally been applied to the architecture of systems that consists of both hardware and software. The main concern addressed by the systems architecture is then the integration of software and hardware in a complete, correctly working device.

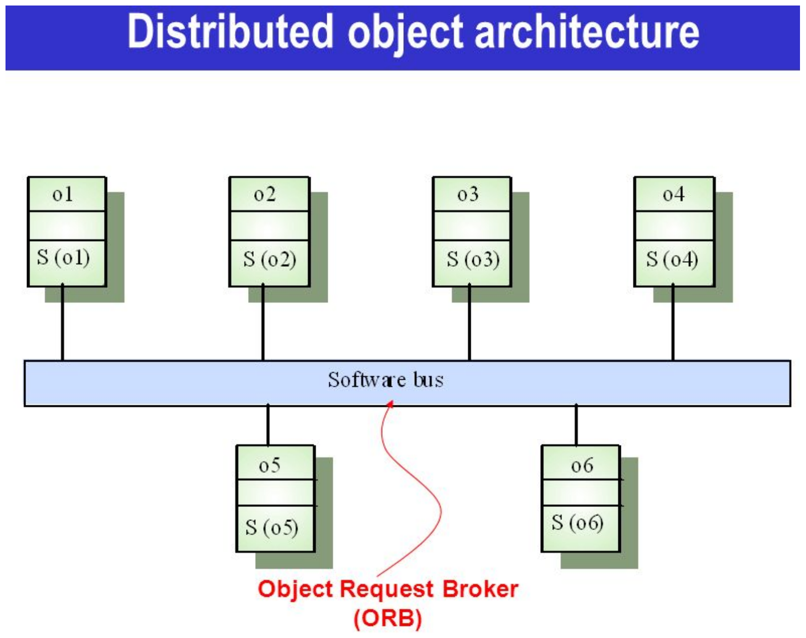
Evolution of system architectures:

- **Mainframe-based architectures**
- **File sharing architectures**
- **Distributed software systems**(The processing of distributed software systems distributed over a set of independent execution hosts)
    - **Client/server (C/S) architectures(**Each process plays the role of client or server)**:**
        - **Application layers:**
            - **The presentation layer** is concerned with collecting user inputs and presenting the results of a computation to system users
            - **Application processing** layer concerned with providing application-specific functionality, e.g., in a banking system, banking functions such as open account, close account, etc.
            - **The data management** layer is concerned with managing access to application data
        - **two-tier(thin client, fat client)**
            - client handles both the Presentation layer (application interface) and Application layer (logical operations), while the server system handles the database layer.
        - **three-tier(upper layer, middle layer, bottom layer)**

        - Each of the application architecture layers executes on a separate processor
        - Allows for better performance than a two-tier thin-client approach and is simpler to manage than a two-tier


2-Tiered Architecture
Client (C++, .NET, Java, etc.)
Database
3-Tiered Architecture
Client (IE, Chrome, Firefox, etc.)
Application Server (IIS, Apache, NGINX, etc. with ASP.NET, Rails, Java EE, etc)
Database

- ○
- ○ **Distributed objects architectures**



- ○
    - ■ No distinction between client and server. Each distributed object acts both as a client and as a server
    - ■ The remote communication between objects is made transparent by the use of middleware based on the software bus concept
        - **abstract bus:** specification of the interface providing communication and data exchange services
        - **bus implementation:** implementation of the abstract bus for a given HW/SW platform
- ○ **Component-based architectures**
    - ■ define software products assembled from a set of software components, which are designed to work together as part of a component framework

        - A component can be used across many different applications and can be reconfigured when application requirements
        - **Important aspects of components:**
            - encapsulation of software structures as abstract components (**variability**)
            - composition of components by binding their parameters to specific values, or other components (**adaptability**)
            - **Component characteristics**
                - A unit of **independent deployment**(never deployed partially)
                - A unit of **third-party composition**(sufficiently documented and self-contained to be “plugged into” other components by a third party)
                - Has no **persistent state**(cannot be distinguished from copies of its own, here will be at most one copy of a particular component)
                - **Replaceable part of a system**(can be replaced by another component that conforms to the same interface)
                - **Fulfills a clear function** and is logically and physically cohesive
                - **May be nested** in other components
    - **Service-oriented architectures**

Distributed software systems
Middleware refers to the software layer that provides connectivity. It’s between the application operating system layers and provides a set of services to establish the required interaction among the various application processes executed by networked hosts

Coupling BCED approach
- **Intra-layer coupling**
    - Desirable
    - localizes software maintenance and evolution to individual layers
- **Inter-layer coupling**
    - to be minimized
    - communication interfaces to be carefully defined

NOTE* The **Law of Demeter** can be used to reduce the inter-layer class coupling

Law of Demeter:

- Also known as the "don't talk to strangers"
- Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
- Each unit should only talk to its friends; don't talk to strangers.
- Only talk to your immediate friends.

SOA(Service-Oriented Architecture)

- a distributed software architecture that consists of multiple autonomous services
- **Component-based:** reuse components, by integrating into a software application
- **Services based:** will be reusing services, without not interacting with them. Making the remote call, when necessary of a function.
    - Each service has a service description => that allows applications to communicate(location, data exchange requirments) with a service.
- Service provider supports services used by multiple clients.
- Service brokers are the missing link between the consumer and the provider. Makes it easier to use service => the role of service brokers becomes critical in situations that need integration among multiple systems as they bridge the interoperability gaps among multiple cloud solutions and thus compensate for the lack of standards.

<u>Services Design Principles</u>

- **Loose coupling**
    - a service should hold a minimum amount of information about other services and ideally should not depend on other services
- **Service contract**
    - the contract is typically defined in the service interface in the form of a set of operations
    - between service providers and user
- **Autonomy**
    - Each service is self-contained, such that it can operate independently without the need for other services.
    - services do not directly communicate with each other
- **Abstraction**
    - Details of service are hidden. Reveals its interface when it provides operations

- **Reusability**
    - A key goal of SOA is to design services that are reusable
- **Composability**
    - Services are designed to composite with larger services.
- **Statelessness**
    - Where possible, services maintain little or no information about specific client activities
- **Discoverability**
    - Service provides an external description to help allow it to be discovered by a discovery mechanism

Software Architectural Broker Patterns

- Broker registers the service.
- Clients locate services with a help of a broker.
- After the broker has brokered the connection between client and service, communication between client and service can be direct or via the broker

Transparency

- The broker provides both location and platform transparency.
- Location transparency: if services changed location, clients won't be notified about it, only the broker will be notified.
- Platform transparency: each service can execute on a different hardware/software platform and does not need to maintain information about the platforms that other services execute on

Technology Support for SOA

- SOAs are successful on web services technology platforms
    - Web services: use www for application-to-application communication
        - Web services are APIs, that provide communication among different software applications on www
        - Web service is business functionality provided by the company for other companies or programs to use

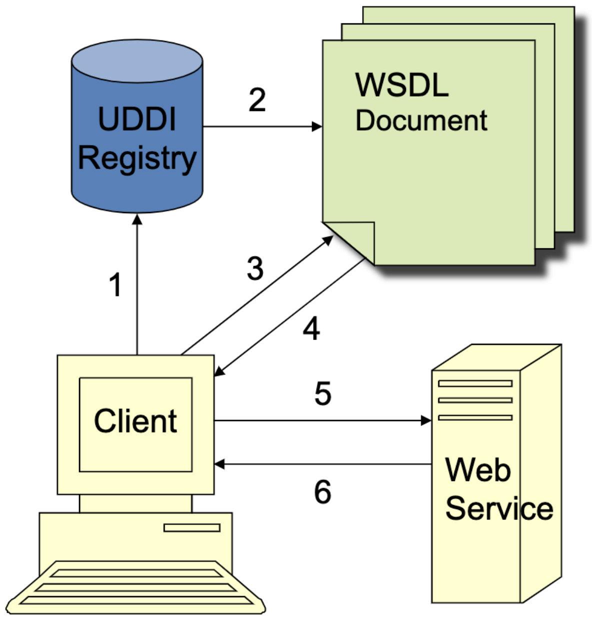
Web Service Protocols

- XML is used to exchange data.
- Simple Object Access Protocol(SOAP) is developed by www consortium, using XML for serializing and HTPP to permit the exchange of information.

Registration Services

- Registration service is provided for services to make their service available to clients. Enables to be published and located via www.
- WSDL is an XML-based language used to describe what a service does, where it resides, and how to invoke it.

Web Service Protocols and Standards



1. Client queries UUDI registry to locate service
2. Registry refers client to WSDL document
3. Client accesses WSDL document
4. WSDL provides data to interact with web service
5. Client sends SOAP-message request
6. Web service returns SOAP-message response

WSDL(Web Services Description Language)

- WSDL is an XML notation for describing a web service. A WSDL definition tells a client how to compose a web service request and describes the interface that is provided by the web service provider.

REST(Representational State Transfer)

- Roy T.Fielding used to describe an architecture style of network systems
- RESTful API: a resource-based API that uses the HTTP protocol

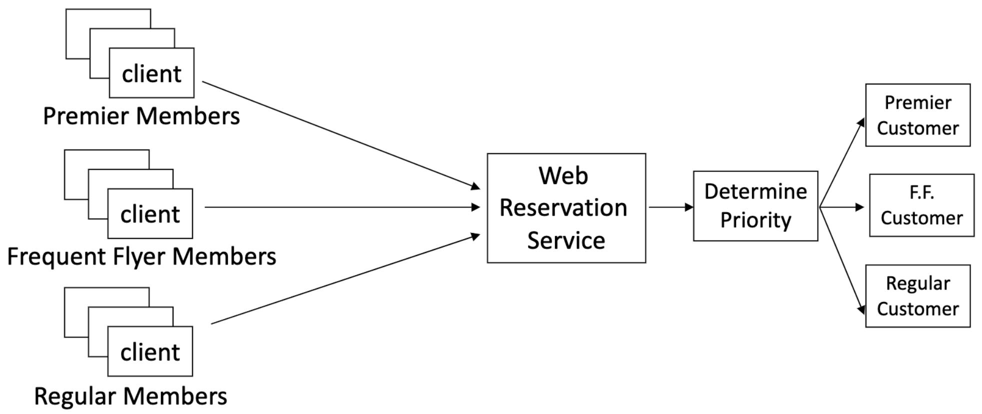
Conventional vs. REST-based design

- **Conventional software** is typically a software application that can accomplish some specific tasks. For instance, a web browser is a conventional software.

Two main approaches to design and implement
the Web reservation service

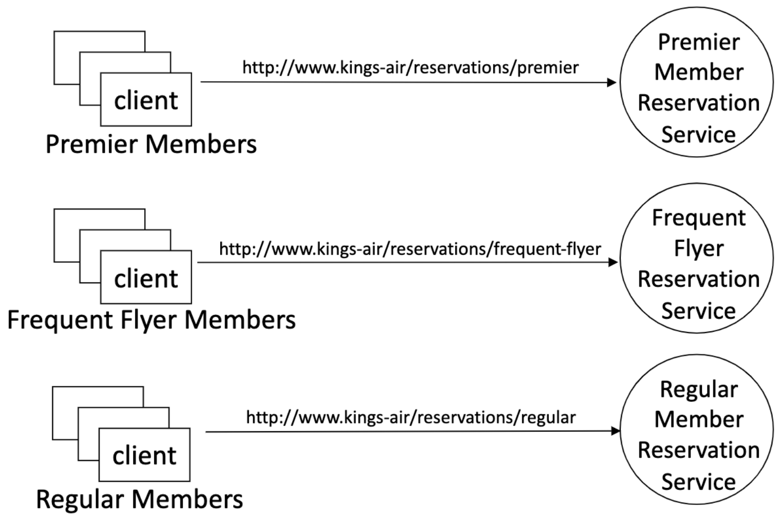
Single URL approach

- The Web service is responsible for examining incoming client requests to determine their priority and process them accordingly



Multiple URLs approach

One URL for premier members, a different URL for frequent flyers, and still another for regular customers


client
Premier Members
http://www.kings-air/reservations/premier
Premier Member Reservation Service
client
Frequent Flyer Members
http://www.kings-air/reservations/frequent-flyer
Frequent Flyer Reservation Service
client
Regular Members
http://www.kings-air/reservations/regular
Regular Member Reservation Service

<u>Software Architectural Transaction Patterns</u>

- Transaction from a client
- A transaction is a request from a client to a service that consists of two or more operations that perform a single logical function

Transaction Property (ACID)

- **Atomicity**(A): indivisible unit of work. Either entirely completed or aborted
- **Consistency**(C): After the transaction executes, the system must be in a constant state
- **Isolation**(I): A transaction's behavior must not be affected by other transactions
- **Durability** (D): Changes are permanent after a transaction completes. These changes must survive system failures. This is also referred to as **persistence**

Two-Phase Commit Protocol

- Address managing atomic transactions in distributed systems

Compound Transaction Pattern

- can be used when the client's transaction requirement can be broken down into smaller flat atomic transactions, in which each atomic transaction can be performed separately and rolled back separately

Long-Living Transaction Pattern

- Has a human in the loop and that could take a long and possibly indefinite time to execute, because individual human behavior is unpredictable
- splits a long-living transaction into two or more separate transactions

Negotiation Pattern

- A client agent acts on behalf of the user and makes a proposal to a service agent
- The service agent attempts to satisfy the client's proposal, which might involve communication with other services
- The service agent then offers the client agent one or more options that come closest to matching the original client agent proposal
- The client agent may then request one of the options, propose further options, or reject the offer

Negotiation Services

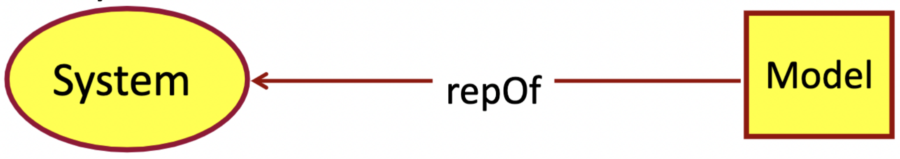
- The client agent, who acts on behalf of the client, may do any of the following:
    - **Propose a service**. The client agent proposes a service to the service agent. This proposed service is negotiable
    - **Request a service.** The client agent requests a service from the service agent. This requested service is nonnegotiable
    - **Reject a service offer.** The client agent rejects an offer made by the service agent.
- The service agent, who acts on behalf of the service, may do any of the following:
    - **Offer a service.** In response to a client proposal, a service agent offers a counter-proposal.

        - **Reject a client request/proposal.** The service agent rejects the client agent's proposed or requested service.
        - **Accept a client request/proposal.** The service agent accepts the client agent's proposed or requested service.

<u>Service Coordination in SOA</u>

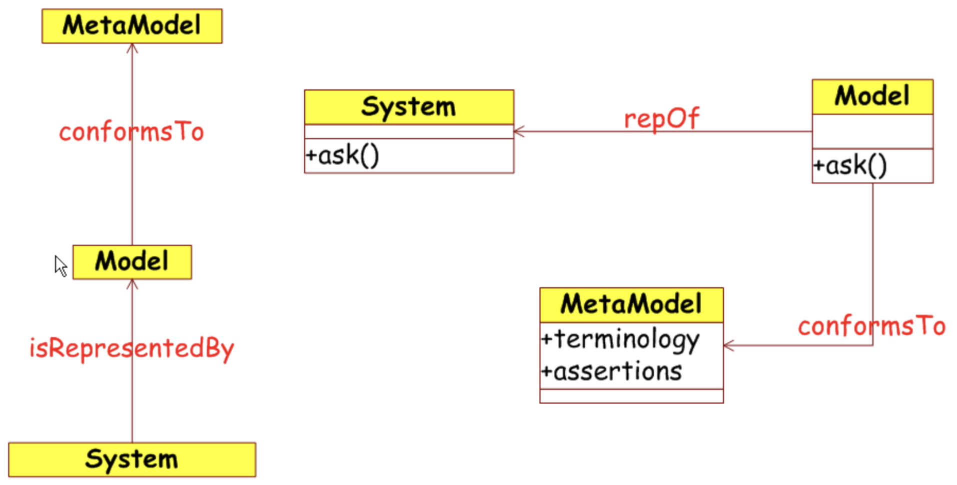
- In SOA applications that involve multiple services, coordination of these services is usually required
- Types:
    - **Orchestration** consists of centrally controlled workflow coordination logic for coordinating multiple participant services
        - This allows the reuse of existing services by incorpo-rating them into new service applications
    - **Choreography** provides distributed coordination among services, and it can be used when coordination is needed between different business organizations
        - can be used for collaboration between services from different service providers provided by different business organizations
        - Whereas orchestration is centrally controlled, choreography involves distributed control

Model-Driven Engineering(MDE)

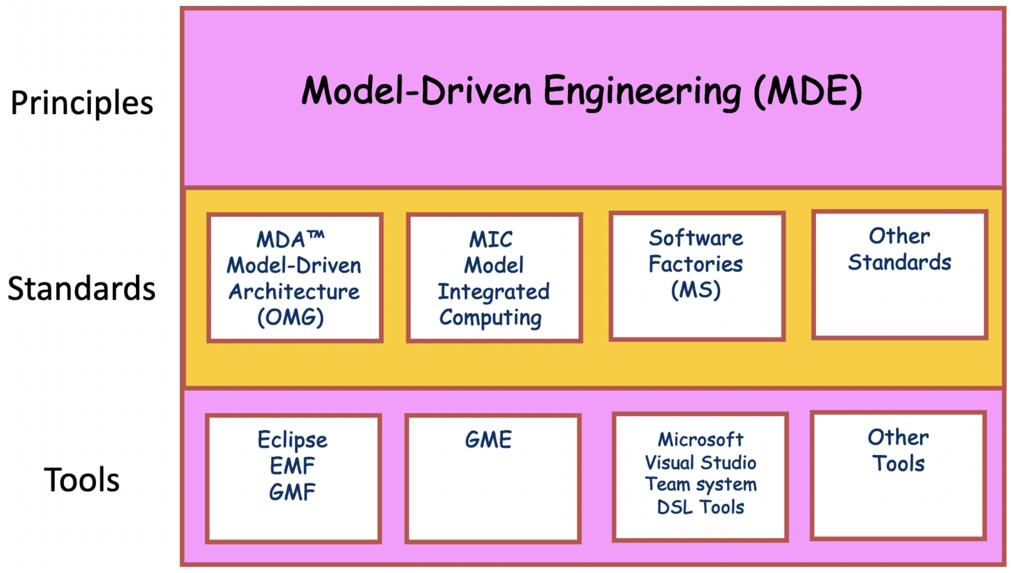
- **Modeling**: form of description. It's an abstract of a system
    - Represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality.
    - 

    - **System** is a set of elements in interaction
    - **Abstraction** is the process of removing physical/temproral details of the system to focus on details of greater importance.
- **Meta-Model** provides the primitives used to build and interpret the model.

- The map legend is the **meta-model**



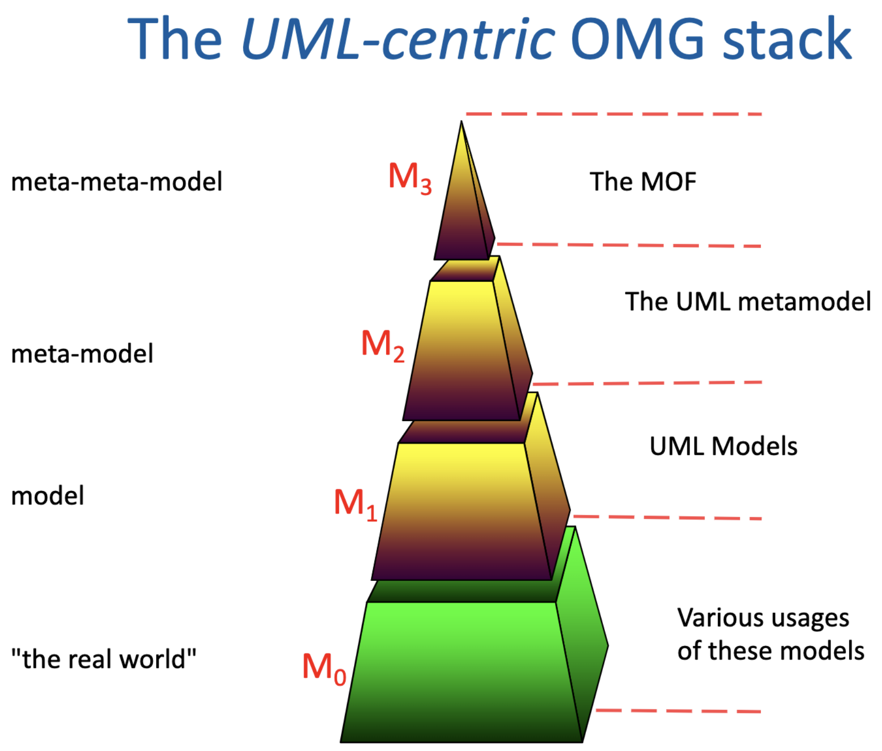
<u>Model-Driven Engineering Principles</u>


Principles
Model-Driven Engineering (MDE)
Standards
MDA™ Model-Driven Architecture (OMG)
MIC Model Integrated Computing
Software Factories (MS)
Other Standards
Tools
Eclipse EMF GMF
GME
Microsoft Visual Studio Team system DSL Tools
Other Tools

Model-Driven Architecture(MDA)

Model-driven architecture(MDA) provides guidelines for structuring software specifications that are expressed as models.

- ○ MDA separates business and application logic from underlying platform technology.



-
    - MDA standards
        - **UML MM** : to define models of software systems
        - **SPEM MM:** to define models of software processes
        - **QVT MM**: to define model transformations
- **The Model-Driven Architecture** starts with the well-known and long-lasting idea of separating the specification of the operation of a system from the details of the way that system uses the capabilities of its platform
- **MDA** provides an approach for, and enables tools to be provided for:
    - specifying a system independently of the platform that supports it
    - specifying platforms
    - choosing a particular platform for the system and
    - transforming the system specification into one for a particular platform

- Basic **MDA** concepts:
    - **Abstraction**
    - **Refinement**
    - **Model-driven:** MDA increases the power of models in that work
    - **Platform** is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns.
- **MDA viewpoints**
    - **Computation Independent Viewpoint** focuses on the environment of the system, and the requirements for the system
        - It is a view of a system from the computation independent viewpoint. It show details of the structure of systems.
        - sometimes called a domain model
    - **Platform Independent Viewpoint(PIM)** focuses on the operation of a system while hiding the details necessary for a particular platform
        - It's a view of a system from the platform independent viewpoint
    - **Platform Specific Viewpoint (PSM)**combines the platform independent viewpoint with an additional focus on the detail of the use of a specific platform by a system
        - Combines the specifications in the PIM with the details that specify how that system uses a particular type of platform
        - A platform model provides a set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform
- **Mappings**
    - provides specifications for transformationof a PIM into a PSM for a particular platform
    - The platform model will determine the nature of the mapping

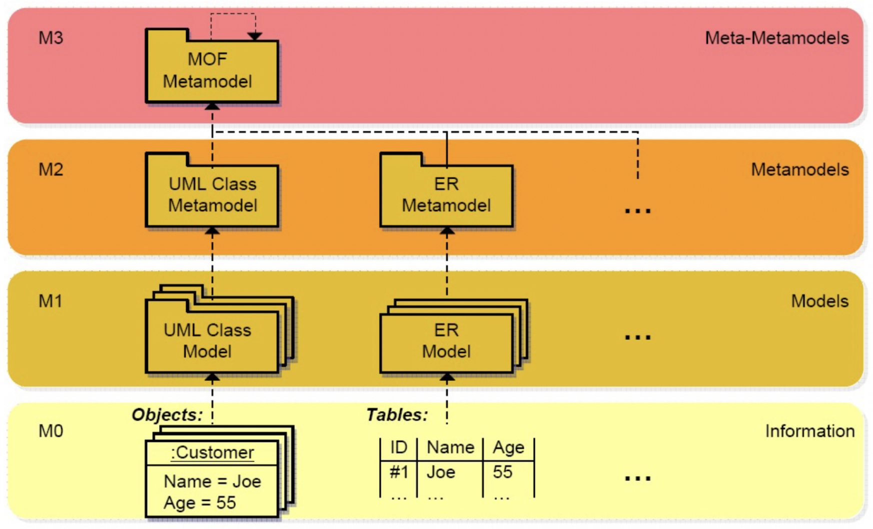
Meta Object Facility(MOF)

- **MOF** is a standard metamodeling framework for model and metadata driven systems.
- **MOF** is the MDA's basic mechanism for defining modeling languages.
- **MOF** looks remarkably like a small subset of **UML.**

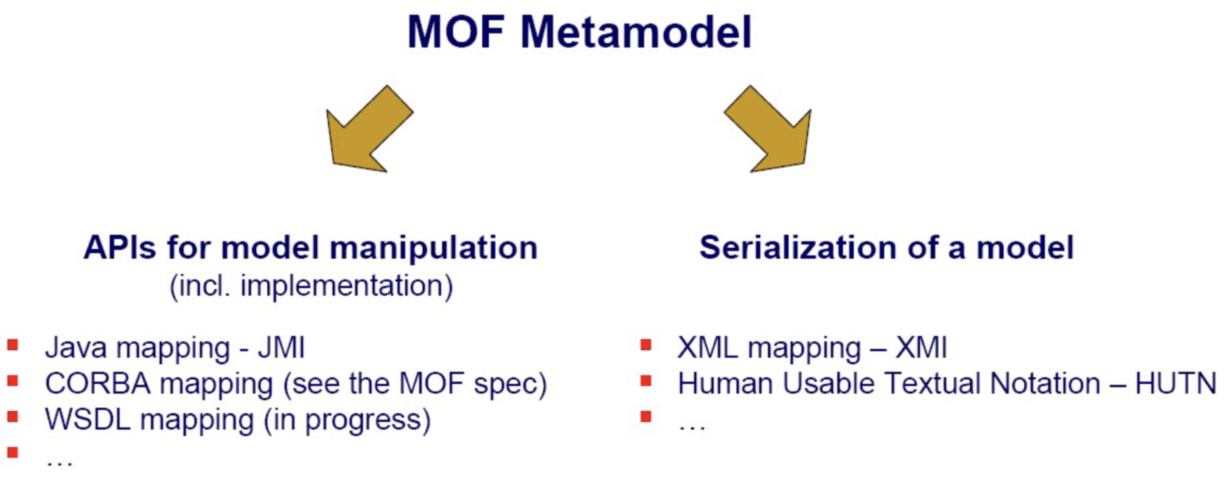
<u>Metamodelingin MOF</u>

- Meta model is a model of a modeling language.
- MOF provides a set of concepts to define metamodels, in particular
  - Class diagrams to define abstract syntax and
  - OCL to define semantics of a modeling language

Metamodeling Framework



-
- **Serialization** is a process of translating a data structure or object state into a format that can be stored or transmitted and reconstructed later.



MOF Metamodel

APIs for model manipulation
(incl. implementation)

- Java mapping - JMI
- CORBA mapping (see the MOF spec)
- WSDL mapping (in progress)
- …

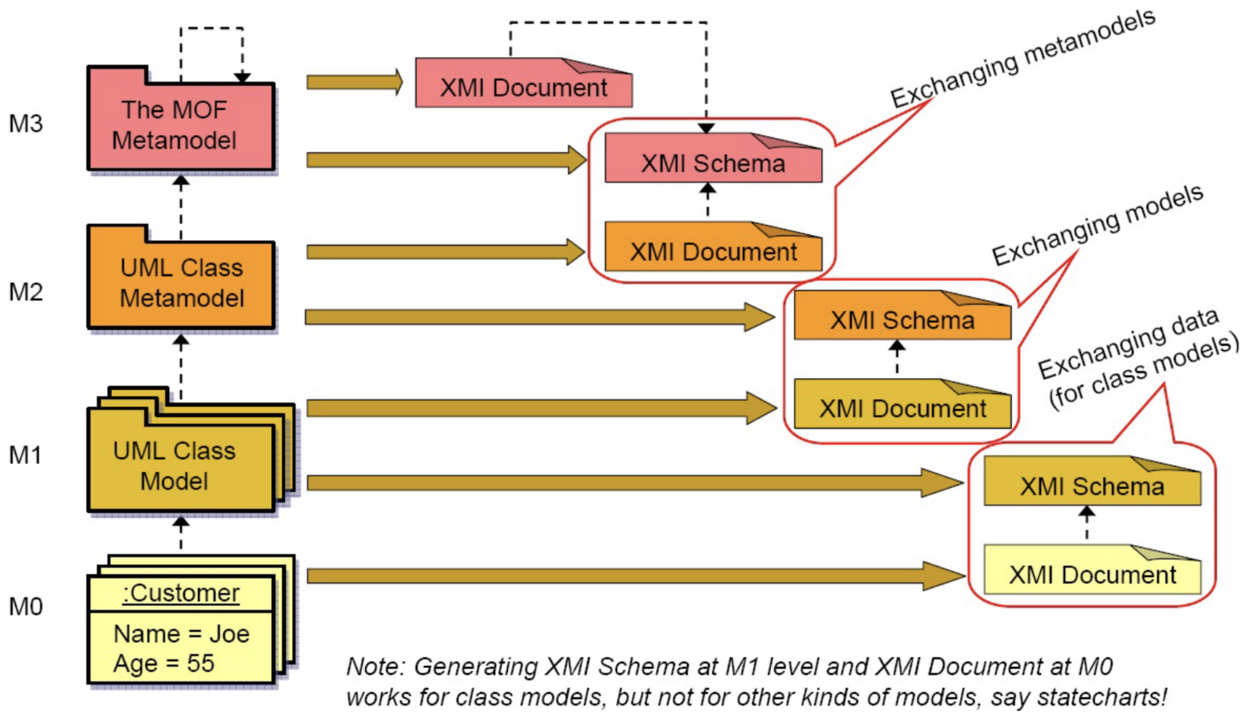
Serialization of a model

- XML mapping – XMI
- Human Usable Textual Notation – HUTN
- …

XML MetadataInterchange(XMI)

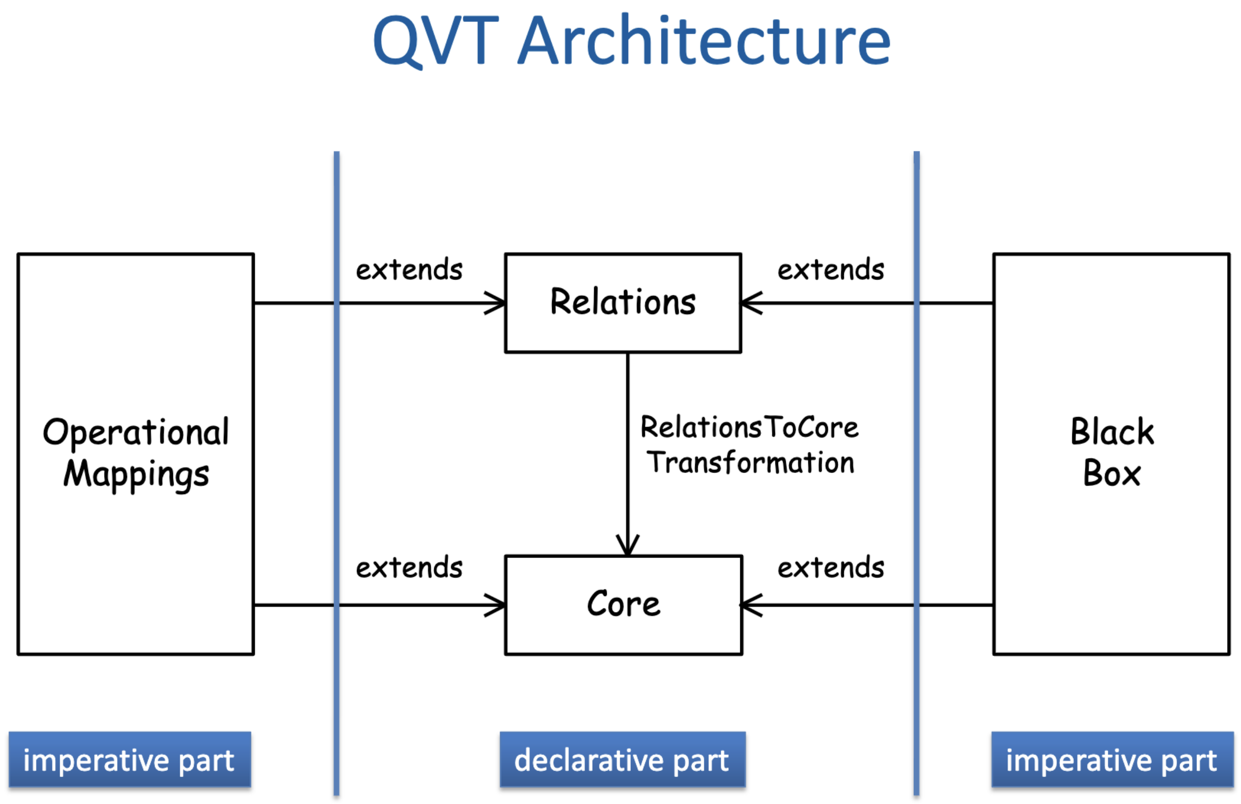
- A standard way of mapping objects to XML
    - XMI allows for defining, interchanging, manipulating and integrating XML data and objects
        - Defines rules for creation of schema from metamodel and document from model
    - XML isn't object oriented
    - May be used for serializing objects at meta-levels

Serialization at different meta-levels



-
- XMI in Eclipse
    - Eclipse uses the Eclipse Modeling Framework (EMF) as its Meta Object Facility
    -

QVT
Query / Views / Transformations

- QVT (Query/View/Transformation) is a standard set of languages for model transformation defined by the Object Management Group.
    - the **OMG** standard covers transformations, views and queries together
    - 

-
- The QVT standard defines three model transformation languages. All of them operate on models which conform to Meta-Object Facility (MOF) 2.0 metamodels. the transformation states which metamodels are used.
    - **QVT-Operational** is an imperative(important) language designed for writing unidirectional transformations.
        - Extends Relations language with imperative constructs
    - **QVT-Relations** is a declarative language designed to permit both unidirectional and bidirectional model transformations to be written.
        - Specification of relations over model elements
        - **Transformationsand ModelTypes**
            - A transformation between candidate models is specified as a set of relations that must hold for the transformation to be successful

            - A candidate model is any model that conforms to a modeltype
            - A **domain** is a distinguished typed variable that can be matched in a model of a given model type
                - A domain has a pattern, which can be viewed as a graph of object nodes, their properties and association links originating from an instance of the domain's type
                - A pattern can be viewed as a set of variables, and a set of constraints that model elements bound to those variables must satisfy to qualify as a valid binding of the pattern
        - A transformation contains two kinds of relations:
            - Top-level: requires that all its top-level relations hold
            - Non-top-level: old only when they are invoked directly or transitively
        - The Relations Metamodel is structured into three packages:
            - **QVTBase** contains a set of basic concepts, many reused from the EMOF and OCL specifications that structure transformations, their rules, and their input and output models
            - **QVTTemplate**
            - **QVTRelation**
    - **QVT-Core** is a declarative language designed to be simple and to act as the target of translation from QVT-Relations.
- **BlackBox Implementations**
    - Allow complex algorithms to be coded in any programminglanguage