## Architectural design:
Defines main components of a system and the relationships between them.

Outputs architectural model

## Architecture in the small:
Architecture of individual programs and their components.

## Architecture in the large:
Architecture of complex enterprise systems that include other systems programs and components.

## Architectural Representations:
Simple informal block diagrams: components & relationships

Based on UML - arch description language

don't show type of relationship and properties of entities (semantics)

## Use of architectural models:
Facilitate discussion about the system design- documentation

## Architectural design decisions:
- Is there a generic application architecture that can be used?
- How will the system be distributed?
- What architectural styles are appropriate?
- How will the system be decomposed into modules?
- How will the architectural design be evaluated?

## Architecture reuse:
Product lines: have a core architecture

## Architecture& System characteristics: MAPSS
Maintainability: replaceable components

Availability: include redundant components for fault tolerance

Performance: minimize communication – localize critical communications

Security: layered architecture

Safety: Localize safety-critical features in a small number of sub-systems.

## Architectural Views 4+1:LPPD
Logical view: key abstractions in the system as objects or object classes.

Physical view: hardware and software components distributed across the processors in the system.

Process view: at runtime, how the system is composed of interacting processes.

Development View: software decomposed for development.

Related using use cases or scenarios.

## Architectural Patterns: to share represent reuse knowledge
A stylized description of good design practice.

Represented tabular/graphical descriptions.

Uses.

### Layered Architecture: presentation – app - data

Model the interfacing of subsystems.

Organizes system into a set of layers each with its own services.

Supports the incremental development of sub-systems in different layers.

### Repository Architecture:

All data in a system is managed in a central repository that is accessible to all system components. Components have their data and do not interact directly, only through the repository.

### Client-server Architecture:

Distributed system model that shows how data and processing are distributed across components.

Set of servers that provide certain services for clients to be accessed through a network

### Pipe and Filter Architecture:

Filters process input data and produce output data that can flow from one filter to another through pipes.

## Application architectures:

Businesses: common => application systems: common architecture

## Application types DELT:

### Data processing:

Data-driven applications that process data in batches without user intervention.

### Transaction processing: e-commerce/reservation

Data-centered applications that process user requests and update information in a database.

### Language processing: compilers/CLI

User intentions specified in a formal language/nat=> processes & interpreted by the system and output some other representation of that language.

### Event processing:

System actions depend on events from the system's environment.

## Information System Architecture:

- layered:
  - UI
  - User communications
  - Information Retrieval
  - Database
- Transaction based

## Web-based Information System:

- Muti-tier client server
- Web server: ui-user communications
- App server: logic-info storage-requests
- Database server:to and from database

## Compiler Components

- Compiler fiya 8 letters w haydek er yaane 6 components
- Lexical analyzer : ->language tokens->internal form
- Semantic analyzer: info from syntax tree and symbol table to check correctness
- Syntax analyzer
- Syntax tree: structure representing code
- Symbol table: var names..
- Code generator: walks the syntax tree->machine code

8:

## Program Testing:

-does what its intended for
-check for defects before putting into use
-execute a program using artificial data
-presence of errors
-part of V&V

### Goals:

-software meets requirements => validation testing- expected system use
1 test for every requirement
-discover incorrectness and undesirables(crashes) => defect testing - unexpected system use

## Verification vs Validation:

### Verification: Are we building the product right?

The software should conform to its specification

### Validation: Are we building the right product?

The software should do what the user requires

### V&V Confidence: MUS

- System is fit for purpose
- Depends on system's purpose: how critical, user expectations: low expectations maybe and marketing environment: getting it to the market may be more important than finding defects.

## Inspections and Testing:

- complementary
- Both used in V&V
- Inspection: conformance with a specification - cant check nonfunctionalities

### Software inspection: static verification

-Static verification involving the static representation of the system before executing it such as source code analysis or design documents.
-Can be one before implementation
-Can be applied to any representation of the system
-Discovering program errors

#### Advantages:

- Errors can mask other errors (we don't have to be concerned with interactions between errors).
- We can inspect incomplete version of the system w/o additional costs
- Checking standard compliance , portability and maintainability

### Software Testing: dynamic verification

Observing product behavior by executing system with test data.

### Stages of Testing: DRU

**Development testing: during development for bugs and defects carried by the development team (UCS)**
Release testing: testing team tests a complete version or lease that is intended for other than dev team before releasing to the user. Meets functionality, performace dependability and doesn't fail.
Black-box testing
User testing: users or potential users test the system in their own environment

#### Unit Testing: defect

**individual program units or objects and methods are tested in isolation focusing on their functionality.**
identify sequences of state transitions to be tested

**Object Class Testing:**

-testing operations associated with the object
-setting/integrating object attributes
-exercising objects in all possible states

**Automated Testing:**

-whenever possible unit testing should be automated using a test automation framework to wrote and run your program tests.
Components:
Setup: in out /call: methods to be tested/assertion: comparison with true false

**Effectiveness:**

Test cases should reveal defects
Does what it is supposed to do
2 types of unit test case:
Normal
Abnormal inputs

**Testing Strategies:**

- partition testing:
    - Identify groups of input with common characteristics and should be processed the same way: equivalence partition
    - Test cases should be chosen from each partition
- Guideline-based testing: use testing guidelines to choose test cases. Guidelines should reflect previous experience of errors often encountered by programmers
- buffer overflows: design inputs that cause it, null pointers
- Repeat the same input or series of inputs numerous times
- Force computation results to be too large or too small.
- Choose inputs that force the system to generate all error messages
- For sequences: size 1- varying sizes - size 0

### Component Testing:

Individual units integrated. Testing component interface to verify It behaves according to its specification.

**Interface Testing:**

- To detect faults due to interface errors.
- Interface types:
    - Parameter interface: data passed b/w methods
    - Shared memory
    - Procedural: Sub-system encapsulates a set of procedures to be called by other sub-systems.
    - Message passing: Sub-systems request services from other sub-systems
- Interface errors:
    - Misuse: a calling component makes a mistake in using the interface of another component (wrong order of parameters)
    - Misunderstanding: a calling component makes assumptions about the behavior of the called component that are incorrect. Passed by ref aw kif
    - Timing errors: The called and calling component operate at different speeds and out-of-date information is accessed.
- Interface testing guidelines:
    - Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
    - Always test pointer parameters with null pointers.
    - Design tests which cause the component to fail.
    - Use stress testing in message passing systems (e.g., overloading a message queue).

### System Testing:

Some or all components are integrated. Testing component interactions. Check component compatibility and data transfer.
-involves integration of separately developed(other team) and off the shelf components.

**Testing Policies:**
Exhaustive testing is impossible
Need required coverage
User input: correct and incorrect
All system features accessed thru menus should be tested.

## Test-driven Development TDD:

- Inter-leave testing and code development.
- Test written before the code
- Develop code incrementally along with a test to that increment in which you don't move to the next increment unless your code passes its test
- Agile method: Extreme Programming

### TDD benefits:

- Code coverage: laen ha kun ktbe at least 1 test la kl code
- Regression testing: a regression test suite is developed incrementally as a program is developed.
- Simplified debugging: deghre baarf wen ghalat
- System documentation: tests are form of documentation

### Regression testing:

- Li jdid ma naza3 l adim balla ma yhes BALA MA NHES BALA MA NHESSSSS
- Expensive => automate: all test rerun everytime a change is made to the program
- Test run successfully before commit change

## Release Testing:

A form of system testing

- Differences:
    - Different team
    - System: bugs in the code(defect) -- Release: meets requirements and good enough to release (validation)

## Performance Testing:

- Series of tests where the load is steadily increased until performance is unacceptable
- Stress testing: deliberately overload to test fault behavior

## USERRRR TESTINGGG:

- Customer - user provides input and advice on system testing
- Essential
- User working environment matters w effects reliability performance usability and robustness

### Types of user testing:

**Alpha testing:**
Interactive maa l development team w betsir aa ardon home match yaane w ruhh
**Beta testing:**
Release software to experiment and report issues
**Acceptance testing:**
Customers mfakkar halo shaghle huwe li bi2arrir eza hal shi sar jehiz yenzal aa ardon w sheghlon ha22o bl niheye

- Stages: 6
    - Define acceptance criteria
    - Plan acceptance testing
    - Derive acceptance tests
    - Run acceptance tests
    - Negotiate test results
    - Reject/accept system

9:

## Software Change:

- Inevitable
- New requirements emerge
- Business environments change
- Errors must be repaired
- New computers and equipment added
- Performance and reliability enhancement

## Importance of Evolution:

- software is a huge investment for businesses therefore must be changed and updated
- Majority of software budget is devoted to evolution

## Evolution and Servicing:

### Evolution:

Stage in software lifecycle : in use and is evolving as new requirements are proposed.

### Servicing:

Software remains useful but only changes are to keep it operational. No new functionality is added.

### Phase-out:

No further changes are made to it.

## Change Implementation:

- Iterations of the development process where the revisions to the system are designed implemented and tested
- The first stage of change implementation may involve program understanding.

## Urgent change requests:

- Urgent changes may have to be implemented without going through se process :
    - Serious system fault to allow normal operation to continue
    - Changes to the system environment have unexpected effects
    - Business changes that require a very rapid response

## Agile methods and evolution:

- Based on incremental development so the transition from dev to evolution is a seamless one.
- Simply a continuation of the development process based on frequent system releases
- Changes may be expressed as additional user stories

## Handover Problems:

- Development team used agile methods and evolution team are unfamiliar and prefer plan-based approach where they may expect detailed documentation to support evolution.
- Plan-based approach has been used by dev team but evolution team prefer to use agile where code refactoring and simplification is expected

## Program Evolution Dynamics:

- Program evolution dynamics is the study of the processes of system change.
- After several empirical studies lehman and belady proposed laws which applied to large system as they evolved.

### Lehman's laws:

1. Continuing change: program must change
2. Increasing complexity: structure becomes more complex
3. Large program evolution: overtime properties usch as size time and erros is invariant for each system release
4. Organizational stability: rate of development is constant and independent of the resources devoted to the system development
5. Conservation of familiarity: incremental change in each release is approximately constant
6. Continuing growth: functionality continually increase.
7. Declining quality: quality will decline
8. Feedback system:
9. significant product improvement

## Types of Maintenance:

- Maintenance to repair faults:
  Correct bugs
- Maintenance to add or modify functionality
- Maintenance to adapt to new environment :

Diff computer  OS

## Maintenance cost:

- Greater than development cost
- Affected by tech and non-tech factors
- Increases as software us maintained since it corrupts the software structure
- Ageing software can have higher support cost: old lang

### Maintenance cost factors:

- Team stability: same staff=> less cost
- Nb of changes
- Contractual responsibilities: developers didn't make toom for future change
- Staff skills: inexperienced
- Program age and structure: age^ hard to understand and change

## Maintenance prediction:

Assessing which parts of the system may cause problems and have high maintenance cost

## System Re-engineering:

- Re-structuring or re-writing part or all of a legacy system without majorly changing its functionality.
- Applicable where some components need maintenance
- Re-engineering includes effort to make them easier to maintain.
- Major overhaul
- Takes place after the system has been maintained for a whileand costs are ^

### Advantages of re-engineering:

- Reduced risk: higher risk in developing new software
- Reduced cost: reengineering costs<<< new software

### Re-engineering cost factors:

- Quality of the software
- Extent of data conversion required
- Availability of experts especially with old technology

## Preventative maintenance by refactoring:

- Refactoring is the process of making improvements to a program
- Preventative
- Reduces complexity
- Make easier to understand
- Improve structure
- Don't add new functionality
- Minor overhaul
- Continuous process of improvement

## Legacy System Management:

- Scrap system completely
- Continue maintaining
- Re-engineer
- Replace with new system

Strategy depends on business value and system quality.

## Legacy System Categories:

- Low quality, low business value:
  scrapped
- Low quality, high business quality:
  Re-engineer or replaced
- High quality, low business value:
  COTS, scrap or maintain
- High quality, high business value:
  Maintain

## Business Value assessment:

- Business customer
- It manager
- Line manager
- Senior manager
- Other stakeholders

### Issues in business value assessment:

- Use of the system:
  If used occacionally only or by small number of people => low business value
- Output of the system:
  Business depends on it => high business value
- Dependability of the system:
  Not dependable and affects customer => low value

## Factors used in environment assessment:

| Supplier stability | Is the supplier still in existence? |
|---|---|
| Failure rate | Does the hardware have a high rate of reported failures? Does the support software crash and force system restarts? |
| Age | How old is the hardware and software? The older the hardware and support software, the more obsolete it will be. |
| Performance | Is the performance of the system adequate? Do performance problems have a significant effect on system users? |
| Support requirements | What local support is required by the hardware and software? |
| Maintenance costs | What are the costs of hardware maintenance and support software licenses? Older hardware may have higher maintenance costs than modern systems. Support software may have high annual licensing costs. |
| Interoperability | Are there problems interfacing the system to other systems? |

## Factors used in application assessment:

| Factor | Questions |
|---|---|
| Programming language | Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development? |
| Configuration management | Are all versions of all parts of the system managed by a configuration management system? |
| Test data | Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system? |
| Personnel skills | Are there people available who have the skills to maintain the application? Are there people available who have experience with the system? |

## System Measurement:

- The number of system change requests;
- The volume of data used by the system.

23:

## Project Planning:

- Recording work to be done
- Breaking down the work into parts
- Assign work to team members
- Development schedule for carrying out the work
- Resources available
- products
- Anticipate and prepare for possible problems
- Communicated to the team and customer
- Help assess progress

## Planning Stages:

- Proposal stage: bidding for a contract
- Project startup phase: planning who - increments - resource allocation
- Periodically throughout, modifying the plan upon gaining insight

## Proposal Planning:

- Software requirements
- Provide info to set price

Project pricing: developmental costs: staff- hardware- software

## Project Startup Planning:

- More about system requirements
- No design or implementation info
- Create a plan: budget staffing resource allocation
- Project monitoring
- Agile development

## Development Planning:

- Schedule
- cost-estimate and risk revised
- Plan amended as project progresses

## Factors Affecting Software Pricing:

**Contractual terms:** owenership of sourcecode=> less price

**Cost estimate uncertainty: eza wahad menno y2akkad bizid l price shway kermel potential risks**

**Financial health:** Organizations in financial difficulty may lower their price to gain a contract.

**Market opportunity:** sacrificing immediate profit for long-term market share.

**Requirements volatility:** An organization may lower its price to win a contract and charge high prices for changes to requirements

## Pricing Strategies:

### Underpricing:

- To gain a contract to own code
- Gain access to market

### Increased pricing:

- Buyer wants fixed-price to allow for unexpected risks.

### Pricing to win:

- According to what the buyer is willing to pay
- Less than developmental costs=> reduced functionality in initial releases.
- Additional costs add per change at a higher level

## Plan-driven Development:

- Development process planned in detail
- Traditional

### Advantages:

- Handling organizational issues
- Potential problems discovered before the project starts

### Disadvantages:

- Early decision have to revised due to changes

## Plan sections:

- Introduction
- Project organization
- Risk analysis
- Hardware and software resource requirements

- Work breakdown
- Project schedule
- Monitoring and reporting mechanisms

### Project Plan Supplements:

- Configuration management plan
- Deployment plan
- Maintenance plan
- Quality plan
- Validation plan

### Planning process:

- Iterative process
- Plan changes unavoidable:
    - More insight about project team schedule risks requirements
    - Changing business goals

### Planning Assumptions:

- Realistic
- Description problems=> delay
- Take unexpected problems into account
- Contingency so that schedule ma ktir byet2assar

### Risk Mitigation:

Actions taken to reduce the risk of project failure, these actions include:

- renegotiating project constraints and deliverables
- Picking a new schedule of when work should be completed

### Project Scheduling:

- How will tasks organized as separate tasks
- When and how will they be executed
- Estimate time to do each task
- Effort required and who
- Estimate resource needed

#### Project scheduling activities:

- Split project into tasks and estimate time and resources
- Organize tasks concurrently to make optimal use of workspace
- Minimize task dependencies to avoid delays
- Dependent on project manager intuition and experience

#### Scheduling problems:

- Estimating difficulty of problems and cost
- Productivity is not proportional to the number of people working on the project
- Unexpected always happens
- Adding people to a late page project makes it later because of communication overhead

### Project Activity: task

- Basic planning element
- Each activity has:
    - Duration in days or months
    - Effort estimate - nb of person-day or person month to complete
    - Deadline by which it should be complete
    - Defined end-point: document - meeting - execution of tests

### Milestones and deliverables:

- points you assess progress
- Work products delivered to customer

### Agile planning:

**List and explain the 2 stages of agile planning.**

**Release planning:** Decides on features to include in a release of a system (several months) and the order in which the user stories will be implemented

**Iteration planning:** Planning for next increment (2-4 weeks). Stories selected (reflecting time)

**What is the planning game (story-based planning)? Explain**

- based on user stories reflecting system features
- Project team rank them in terms of amount of time
- Stories are assigned 'effort points' : size and difficulty
- Number of effort point/day = team's velocity
- Allows total effort required to be estimated

### Task Allocation:

- During the task planning stage the developers break down stories into developmental tasks:
  - All development task should take 4-16 hrs
  - All tasks to complete stories in iteration shall be listed
  - Developers sign up for taks that they will implement
- Benefits of this approach:
  - Team gets overview
  - Developers have sense of ownership

### Software Delivery:

- Increment delivered at end of iteration
- Delivery schedule never extended instead scope is reduced

### Agile planning difficulties:

- Reliant on customer involvement and availability
- familiarity with traditional project plans

### Agile planning applicability:

- Small stable development team that can get together
- Collaborative

## Estimation techniques:

### Experience based techniques:

- Based on manager's experience of past projects. Informed judgement.
- Identify deliverables
- Document on spreadsheet to estimate them individually and compute the total effort required.
  - Difficulties:
    - New project not common with something old
    - SD changes quickly and new techniques are introduced

### Algorithmic cost modeling:

Formulaic approach to compute project effort based on estimates of size process characteristics experience and staff involved.

COCOMO! (ACM) Application composition model - (EDM) Early Design Model - (RM) Reuse Model - (PM) Post-architecture Model

ACM formula: PM = (NAP (1 - %reuse/100))/PROD:

EDM formula PM = A * Size^B * M

RM formula:

- Blackbox: PM = (ASLOC*AT/100)ATPROD
- White box: PM = ASLOC*(1-AT/100)*AAM

PM same formula as EDM

Exponent term is based on 5 scale factors:

- Precedenteness
- Development Flexibility
- Architecture/Risk Resolution
- Team Cohesion
- Process Maturity

DATPP!

TDEV = 3* PM^(0.33+0.2*(B-1.01))