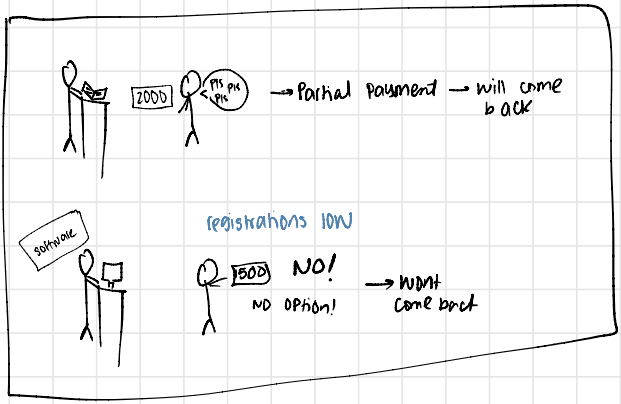
Software Engineering

Farrukh Hassan Syed

Why make software
↳ redundant work
↳ many resources
↳ minimise human error → streamline

Software failu

Feedback differs Person to Person



Missing chp3 , chp4, UI Design

INTRODUCTION

Software Engineering (SE)

- In short, Software engineering is a branch of computer science, which uses well defined engineering concepts required to produce efficient, durable, scalable, in budget, and on-time software products. It is concerned with all aspects of software production, from the early stages of system specification through to maintaining the system after it has gone into use.

Software Crisis

- Systems cost exceeded the initial estimated budget
- Software over-ran delivery schedules
- Software had to be substantially modified
- Projects were unmanageable and code difficult to maintain.
- Software was never delivered.
  - These problems of software development were extensively felt in the beginning of the 1970s and are collectively referred to as the "software crisis".

Reasons for failure [edit]

The project demonstrated a systematic failure of software engineering practices.[3]

- Lack of a strong technical architecture ("blueprint") from the outset led to poor architectural decisions
- Repeated changes in specification
- Repeated turnover of management, which contributed to the specification problem
- Micromanagement of software developers
- The inclusion of many FBI Personnel who had little or no formal training in computer science as managers and even engineers on the project
- Scope creep as requirements were continually added to the system even as it was falling behind schedule
- Code bloat due to changing specifications and scope creep—at one point it was estimated the software had over 700,000 lines of code.
- Planned use of a flash cutover deployment made it difficult to adopt the system until it was perfected.

Why so many issues?

- Requirement of flexibility
  - So flexible that start working with it before fully understanding what needs to be done.
  - Unknowns
  - Changes – policies, requirements, technologies.
- Intangibility/ Invisibility
  - Interfaces undecided
  - Transient hardware, software errors, race conditions, compatibility issues.
  - Visualization (E.g. The program for calculating landing for path finder mission)
- Conformity
  - Systems usually interacts with outside systems.

Stakeholders

- A person, group or organization that has interest or concern in an organization.
- Possible Stakeholders:
  - **Government** taxation, legislation, employment, legalities.
  - **Employees** job security, compensation
  - **Customers** value, quality, customer care
  - **Suppliers** providers of products and services used in the end product for the customer
  - **Creditors** credit score, new contracts, liquidity (banks?).
  - **Community** jobs, involvement, environmental protection, shares, truthful communication.
  - **Trade Unions** quality, worker protection, jobs.
  - **Owner(s)** profitability, longevity, market share, market standing, succession planning, raising capital, growth, social goals.
  - **Investors** return on investment, income.

Software Engineering

Software engineering

- Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.
- Engineering discipline
  - Using appropriate theories and methods to solve problems bearing in mind organizational and financial constraints.
- All aspects of software production
  - Not just technical process of development. Also project management and the development of tools, methods etc. to support software production.

Software project failure

↳ Increasing system complexity
- change in demands rapidly
- hence need to introduce new capabilities
- need to be delived quickly

↳ Failure to use SE methods
- writing programs w/o SE methods cause
  - more expensive software
  - less reliable software

Software Engineering vs Computer Science vs System Engineering

↳ focuses on practicalities of developing and delivering software

↳ focuses on theory and fundamentals

↳ concerned with computer-based system development including
- hardware
- software
- process engineering

Cost of SE

↳ 60% are development cost

↳ 40% are testing cost

↳ software costs more to maintain man to develop → systems with long life have more maintenance cost

↳ maiority of the costs of changing software after it has gone into use

SE Techniques and methods

managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. You can't, therefore, say that one method is better than another.

SOFTWARE PRODUCTS

Generic Products

↳ organisations/individual add features themselves (known market research)

↳ no specific customer requirements → your own requirements

↳ stand alone systems marketed and sold to anyone

e.g diaroz, word processors, calculator

CON

↳ harder to develop as testing is done themselves ??

↳ no investment

↳ UAT → user test themselves

Customized Products

↳ customer tells features

↳ has specific customer requirements

↳ software commissioned by specific customer

e.g NHS

CONS

↳ might become too specific so can't be used somewhere else

BETA

PRODUCT SPECIFICATION

Generic Products

↳ software developer owns

- what product should do
- software changes decisions

Customized Products

↳ customer for the software owns

- what product should do
- software changes decisions

attributes of software

Maintainability → should be modifiable

Dependability → no physical or economic damage when system failure

Security → malicious users don't get acess or damage the system

Efficiency → no wasteful use of system resources → memory, processor cycles

Acceptability → usable (understandable) and compatible with other system that user uses

Imp of SE

↳ people rely on advance software systems

↳ cheaper in the long run to use SE methods → lessens changing costs

Fundamentals of SE activities

↳ Software specification → customers + engineers define software, constraints on operations

↳ Software development → software is designed and programmed

↳ Software validation → software checked to see customer req fulfilled

↳ Software evolution → software is modified

Software Issues

✧ Heterogeneity

- Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices.

✧ Business and social change

- Business and society are changing incredibly quickly as emerging economies develop and new technologies become available. They need to be able to change their existing software and to rapidly develop new software.

✧ Security and trust

- As software is intertwined with all aspects of our lives, it is essential that we can trust that software.

✧ Scale

- Software has to be developed across a very wide range of scales, from very small embedded systems in portable or wearable devices through to Internet-scale, cloud-based systems that serve a global community.

↳ coping with

↳ increased diversity → no universal set of SE methods → depends on
↳ type of application
↳ customer req
↳ background of development team

↳ demands for reduced delivery time

↳ developing trustworthy software

Application types

- Stand-alone applications
  - These are application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to be connected to a network.
- Interactive transaction-based applications
  - Applications that execute on a remote computer and are accessed by users from their own PCs or terminals. These include web applications such as e-commerce applications.
- Embedded control systems
  - These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system.
- Batch processing systems
  - These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs.
- Entertainment systems
  - These are systems that are primarily for personal use and which are intended to entertain the user.
- Systems for modeling and simulation
  - These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects.
- Data collection systems
  - These are systems that collect data from their environment using a set of sensors and send that data to other systems for processing.
- Systems of systems
  - These are systems that are composed of a number of other software systems.

Diff web made to SE

↳ availability of software services

↳ possibility of developing highly distributed service based system

↳ advancement in programming languages

↳ software reuse

Web software engineering

- Software reuse
  - Software reuse is the dominant approach for constructing web-based systems. When building these systems, you think about how you can assemble them from pre-existing software components and systems.
- Incremental and agile development
  - Web-based systems should be developed and delivered incrementally. It is now generally recognized that it is impractical to specify all the requirements for such systems in advance.
- Service-oriented systems
  - Software may be implemented using service-oriented software engineering, where the software components are stand-alone web services.
- Rich interfaces
  - Interface development technologies such as AJAX and HTML5 have emerged that support the creation of rich interfaces within a web browser.

Internet software engineering

- The Web is now a platform for running application and organizations are increasingly developing web-based systems rather than local systems.
- Web services (discussed in Chapter 19) allow application functionality to be accessed over the web.
- Cloud computing is an approach to the provision of computer services where applications run remotely on the ‘cloud’.
  - Users do not buy software buy pay according to use.

Issues of professional responsibility

- Confidentiality
  - Engineers should normally respect the confidentiality of their employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.
- Competence
  - Engineers should not misrepresent their level of competence. They should not knowingly accept work which is outwith their competence.
- Intellectual property rights
  - Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected.
- Computer misuse
  - Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

Ethical principles

1. PUBLIC - Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT - Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES - Software engineers shall be fair to and supportive of their colleagues.
8. SELF - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

SOFTWARE PROCESSES

Software Process

↳ A structured set of activities req to develop a software system

↳ It should involve

- ↳ Specification → defines what system should do
- ↳ Design and Implementation → defines organization and implementation of system
- ↳ Validation → checks it does what the customer wants
- ↳ Evolution → changing system in response to changing customer needs

Software Process Model

↳ an abstract representation of a process

↳ presents a description of a process from some particular perspective

Process Activities

↳ Specifying data model

↳ designing user interface

↳ ordering of activities

Product Description

↳ Products → outcomes of a process activity

↳ Roles → responsibilities of the people involved in the process

↳ Pre and post condition → statement that are true before and after a process activity or product produced

Plan driven processes

↳ all processes activities are planned

↳ Progress is measured against this plan

↳ seperate development phases

Agile Processes

↳ Planning is incremental

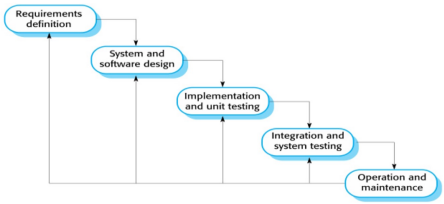
↳ can change plans as per changes in customer req

↳ all phases interlinked

Software Process models

1. The waterfall model → Plan driven
2. Incremental development → plan driven / agile
3. Integration and configuration → plan driven / agile

1. The waterfall model



In principle, a **phase has to be complete before moving onto the next phase.**

Phases:

↳ Req analysis and definition
↳ System and software design
↳ Implementation and unit testing
↳ Integration and system testing
↳ Operation and maintenance

CONS:

↳ cant change requirements after the process is underway
↳ inflexible partitioning of project into stages → difficult to respond to customer req

good when.

↳ req are well understood
↳ very limited changes

• although stable req is very rare

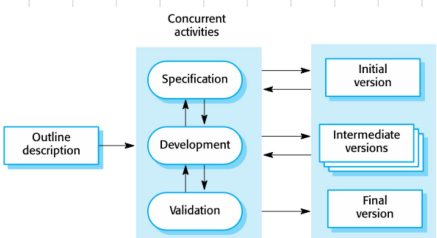
PROS

↳ used for large system engineering projects where system is developed at several sites → as it helps to coordinate the work

2. Incremental model

↳ divide into 3 increments (chunks) → MID 1, MID 2, MID 3

↳ learn while you go



PROS

↳ Reduced changes cost → analysis and documentation that has to be redone is less than waterfall model
↳ Customer feedback ease → on development work → customers can respond to each build
↳ Rapid delivery & development of useful software → as customers are able to use and gain value from software earlier than is possible with a waterfall process

CONS

↳ the process is not visible → to deploy system so quickly, mostly the documentation is not done to reduce development cost → so no documentation is present w.r.t. each version
↳ System degrade → freq changes in req tends to degrade as refactoring is needed → incorporating multiple software changes becomes increasingly difficult and costly

3. Integration and Configuration

↳ based on software reuse

↳ systems are integrated from existing components or application system

↳ reused elements may be configured to adapt user's req

Phases:

↳ Req's specification

↳ Software discovery and evaluation

↳ Req's refinement

↳ Application systems configuration

↳ Component adaptation and integration

PROS

↳ Reduced costs and risks → are less as software is developed from scratch

↳ Faster delivery

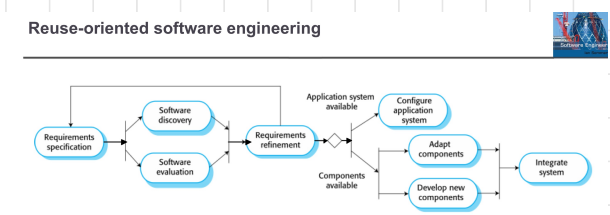
CONS

↳ system may not meet the real needs of client

↳ loss of control over the evolution of reused system elements

Types of Reusable Software

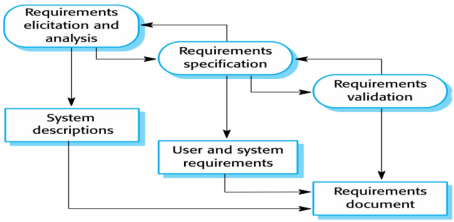
1. Stand alone application systems
2. Package integrated with framework eg .NET
3. Web services



Process Activities

- The **four basic process activities of specification, development, validation** and **evolution** are organized differently in different development processes.
- For example,
  - **waterfall model ->** process activities are **organized in sequence**
  - **incremental development model** -> interleaved.

The requirements engineering process



Software specification

↳ establishesh

- what services are req
- what constraints on operation and development

↳ what stakeholders require or expect from the system

↳ defining req in detail

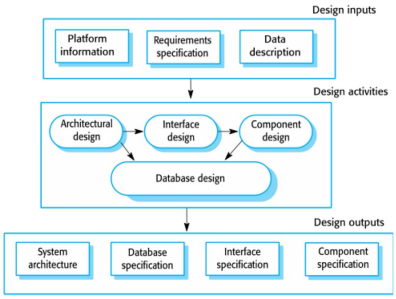
↳ wether the req is valid or realistic

Software design and implementation

↳ converts system specification into executable system

↳ design and implementation activities may be interleaved

A general model of the design process



Design Inputs

- Inputs given to system.
- That could be any **platform** related information like OS , middleware. And any other application systems if needed.
- **Data descriptions** include the data that is to be entered to the system. What could be the appropriate form and so on.
- All the specifications mentioned in requirements elicitation phase are the requirements here i.e. the ultimate functional requirements of software.

Design activities → 4 components

- **Architectural design**
  - identify the overall structure of the system,
  - the principal components (subsystems or modules), their relationships and how they are distributed.
- **Database design**
  - design the system data structures
  - how these are to be represented in a database.
- **Interface design** → system interface not UI
  - define the interfaces between system components. Q) why
  - Must encapsulate all the complexities
  - Should be user friendly
- **Component selection and design**,
  - you search for reusable components.
  - If unavailable, then you design the component from scratch.
  - Includes the list of changes to be done in the off the shelf component
  - or the proper technical UML diagram (if designing from scratch)

System implementation

- The software is **implemented either by developing a program or programs or by configuring an application system** depending upon the scale of software application.
- **Design and implementation are interleaved activities** for most types of software system.
- **Programming** is an individual **activity with no standard process. (e.g., variable definition) ~ depends on programmer feasibility.**
- **Debugging** is the activity of finding program faults and correcting these faults.

Software validation

- Verification and validation (V & V) is intended to show that a **system conforms to its specification and meets the requirements of the system customer.**
- Involves checking and review processes and system testing.
- **System testing** involves **executing the system with test cases that are derived from the specification of the real data to be processed by the system**.
- Alpha testing: testing & reviewing by the development staff
- Beta testing: testing & reviewing by the real time user

Testing stages

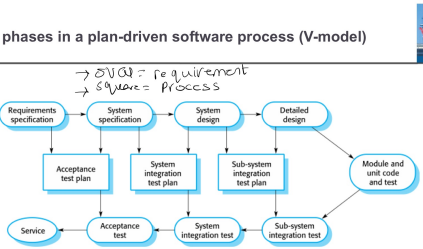
- Component testing
  - Component tested by developers working on the system.
  - Individual components are tested independently;
  - Components may be functions or objects or coherent groupings of these entities.
  - Test automation tools like Junit for java, VS Code plugin for JS or python could be used for unit/component testing.
- System testing
  - Testing of the system as a whole.
  - Testing of errors after different module interaction.
  - In large systems multiple subsystems are integrated first and then on whole integrated as a final system.

Stages of testing

Testing stages

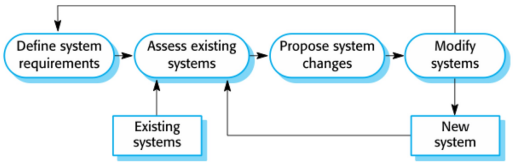
- Customer testing
  - Beta testing
  - Testing with customer data to check that the system meets the customer's needs.
  - It might reveal error in software requirements omission if any.

Testing phases in a plan-driven software process (V-model)



Software evolution

- Hardware changes are very expensive, as compare to that software is inherently flexible and can be modified to larger & complex systems.
- As requirements change through changing business circumstances, the software that supports the business must also evolve and change.
- Although there has been a demarcation between development and evolution (maintenance) this is increasingly irrelevant as fewer and fewer systems are completely new.



Agile

Scrum

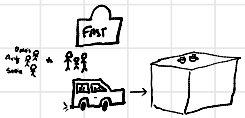
↳ project activies added to agile

↳ plans first

↳ gives feedback after deviliery

XP

↳ Plans after every itteration



DESIGN CONCEPTS

Software Design

↳ transform user req into suitable form

↳ first step in SDLC

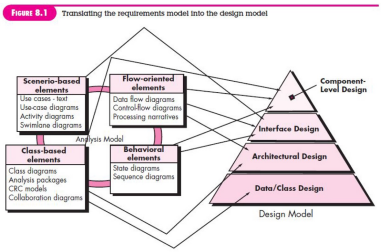
↳ simplifies how to fullfill req mentioned in SRS

↳ done by software Engineer OR UI/UX Designer

- Good software design should exhibit:
- *Firmness:* A program should not have any bugs that inhibit its function.
- *Commodity:* A program should be suitable for the purposes for which it was intended.
- *Delight:* The experience of using the program should be pleasurable one.

Requirement model feeds the design task

Analysis Model -> Design Model



Important Questions

As per previous year question papers

1. Explain different design concepts. 8M
2. What is the relationship between modularity & functional independence. 7M
3. What do you mean by term cohesion & coupling in the context of software design? How are the concepts are useful for good design system. 7M

Phases of Design

1. Data/class Design
2. Architechtical Design
3. Interface Design → interaction b/w system and user devices
   Senario and Behavioral Digram
4. Component Design

Evaluation of Good Design

↳ The Design must

1. Implement all explicit req in the analysis model
2. accomodate all implicit req desired by customer
3. Readable → FOR THOSE WHO → generate code, test code, subsequently support the software
4. Provide complete picture of software → ADDRESSING THE → data, functional, behavioral domain → from an implementation Perspective

Quality Guidelines

↳ A design should

1. Exhibit an architecture
2. be modular → logically partioned into sub elements → 1. Registration 2. bosm
3. contain distinct representation of → data, architecture, interface, componat
4. carry appopriate ds and recognisable data patterns
5. Design componat must show independendt functional characteristics
6. The designed interface should reduce complexity of connections

7. derived using repeatable method

8. have notations to effectively comm. its meaning

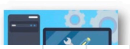
Software Quality Attributes

The attributes of design name as 'FURPS' are as follows:

1. **Functionality:** It evaluates the feature set and capabilities of the program.
2. **Usability:** It is accessed by considering the factors such as human factor, consistency and documentation.
3. **Reliability:** It is evaluated by measuring parameters like frequency and security of failure, output result accuracy, recovery from failure and program predictability.
4. **Performance:** It is measured by considering processing speed, response time, resource consumption, throughput and efficiency.
5. **Supportability:** It combines the ability to extend the program, adaptability, serviceability. Testability, compatibility and configurability are the terms using which a system can be easily installed and found the problem easily.

   Supportability also consists of more attributes such as modularity, reusability, robustness, security, portability, scalability.

UR SPF



FUNDAMENTAL CONCEPTS

- Abstraction—data, procedure, control
- Architecture—the overall structure of the software
- Patterns—"conveys the essence" of a proven design solution
- Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces
- Modularity—compartmentalization of data and function
- Hiding—controlled interfaces
- Functional independence—single-minded function and low coupling
- Refinement—elaboration of detail for all abstractions
- Aspects—a mechanism for understanding how global requirements affect design
- Refactoring—a reorganization technique that simplifies the design
- OO design concepts—Appendix II
- Design Classes—provide design detail that will enable analysis classes to be implemented

M A D P A I R SOA

R

F

1. Abstraction

- Abstraction is used to hide background details or unnecessary implementation about the data.
- So that users see only required information.

Type 1: Procedural Abstraction:

- There is collections of subprograms.
- One is hidden group another is visible group of functionalities. **Example:**

Private:
Fuel_machine()
Set_top_speed()
Develop_engine()

Public:
Turn_on() , Turn_off()
Accelerate(), Break()

Type 2: Data Abstraction:

- Collections of data that describe data objects.
- Show representation data & hide manipulation data.
- **Examples:** Data Structure Programs directly used Push(), Pop(), Top() and Empty() methods.

2. Architecture

- The architecture is the structure of program modules where they interact with each other in a specialized way.

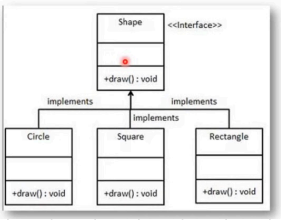
➢ **Structural Properties:** Architectural design represent different types of components, modules, objects & relationship between these.

➢ **Extra-Functional Properties:** How design architecture achieve requirements of Performance, Capacity, Reliability, Security, Adaptability & other System Characteristics.

➢ **Families of related systems:** The architectural design should draw repeatable patterns. They have ability to reuse repeatable blocks.

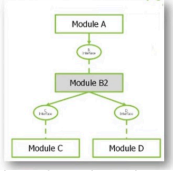
3. Design Patterns

- The pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern.
- **Example:**



4. Modularity

- Modularity simply means dividing the system or project into smaller parts to reduce the complexity of the system or project.
- After developing the modules, they are integrated together to meet the software requirements.
- Modularizing a design helps to effective development, accommodate changes easily, conduct testing, debugging efficiently and conduct maintenance work easily.



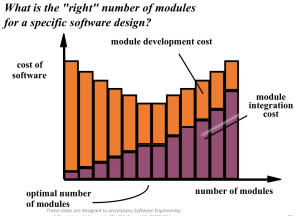
Patterns

"A proven solution to a recurring problem within a certain context amidst competing concerns"

Design Pattern Template

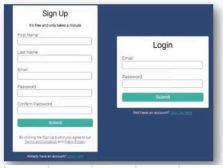
Pattern name—**describes the essence of the pattern in a short but expressive name**
Intent—**describes the pattern and what it does**
Also-known-as—**lists any synonyms for the pattern**
Motivation—**provides an example of the problem**
Applicability—**notes specific design situations in which the pattern is applicable**
Structure—**describes the classes that are required to implement the pattern**
Participants—**describes the responsibilities of the classes that are required to implement the pattern**
Collaborations—**describes how the participants collaborate to carry out their responsibilities**
Consequences—**describes the "design forces" that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented**
Related patterns—**cross-references related design patterns**

Modularity: Trade-offs



5. Information Hiding

- Modules should be specified and designed in such a way that the data structures and algorithm details of one module are not accessible to other modules.
- They pass only that much information to each other, which is required to accomplish the software functions.
- The way of hiding unnecessary details in modules is referred to as information hiding.



6. Functional Independence

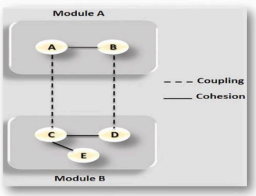
- The functional independence is the concept of separation and related to the concept of modularity, abstraction and information hiding.

Criteria 1: Coupling

- The degree in which module is "connected" to other module in the system.
- Low Coupling necessary in good software.

Criteria 2: Cohesion

- The degree in which module perform functions in inner module in the system.
- High Cohesion necessary in good software.



7. Refinement

- Refinement is a top-down design approach.
- It is a process of elaboration.
- A program is established for refining levels of procedural details.
- A hierarchy is established by decomposing a statement of function in a stepwise manner till the programming language statement are reached.

Example:

INPUT	INPUT
Get number 1 (Integer)	Get number 1 (Integer)
Get number 2 (Integer)	Get number 2 (Integer)
PROCESS	While (Invalid Number)
OUTPUT	EXIT

8. Refactoring

- Refactoring is the process of changing the internal software system in a way that it does not change the external behavior of the code still improves its internal structure.
- When software is refactored, the existing design is examined for
  - redundancy
  - unused design elements
  - inefficient or unnecessary algorithms
  - poorly constructed or inappropriate data structures
  - or any other design failure that can be corrected to yield a better design.

Why Information Hiding?

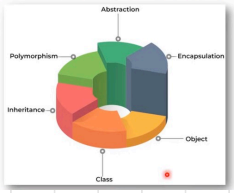
- reduces the likelihood of "side effects"
- limits the global impact of local design decisions
- emphasizes communication through controlled interfaces
- discourages the use of global data
- leads to encapsulation—an attribute of high quality design
- results in higher quality software

Coupling
indicates interdependance b/w modules

Cohesion
indicates functional strength of a module

9. Object Oriented Design Concepts

- Object Oriented is a popular design approach for analyzing and designing an application.
- Advantage is that faster, low cost development and creates a high quality software.



- Design classes
  - Entity classes
  - Boundary classes
  - Controller classes
- Inheritance—all responsibilities of a superclass is immediately inherited by all subclasses
- Messages—stimulate some behavior to occur in the receiving object
- Polymorphism—a characteristic that greatly reduces the effort required to extend the design

Separation of Concerns

- Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- A *concern* is a feature or behavior that is specified as part of the requirements model for the software
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.
- P1 is harder to solve than P2. Which one to solve first?
- P1+P2 may be harder to solve together than one by one.

Aspects

- Consider two requirements, *A* and *B*. Requirement *A crosscuts* requirement *B* "if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account. [Ros04]
- An *aspect* is a representation of a cross-cutting concern.

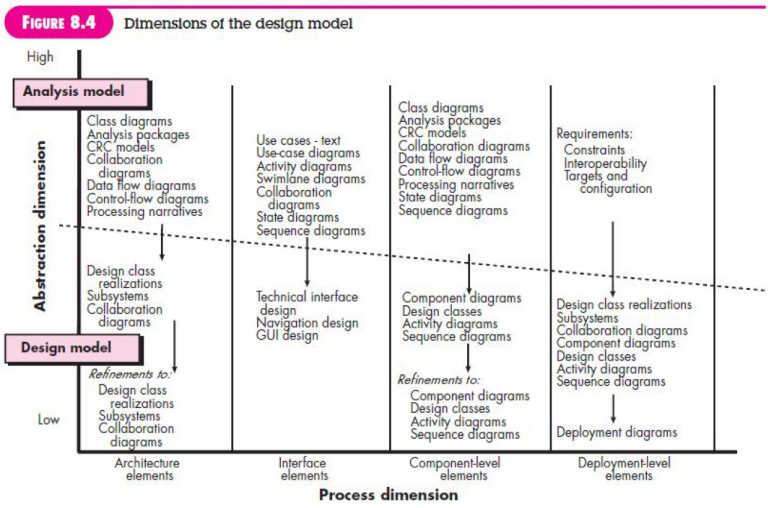
1. a particular part or feature of something.
   "personal effectiveness in all aspects of life"

   Similar: feature facet side characteristic

Design Classes

- *User interface classes* define all abstractions that are necessary for human-computer interaction (HCI). In many cases, HCI occurs within the context of a *metaphor* (e.g., a checkbook, an order form, a fax machine), and the design classes for the interface may be visual representations of the elements of the metaphor.
- *Business domain classes* are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
- *Process classes* implement lower-level business abstractions required to fully manage the business domain classes.
- *Persistent classes* represent data stores (e.g., a database) that will persist beyond the execution of the software.
- *System classes* implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

- Four characteristics of a well-formed design class:
- **Complete and sufficient.** Complete encapsulation of all attributes and methods that can reasonably be expected
- Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.
- **Primitiveness.** Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing
- **High Cohesion**
- **Low coupling**



Design Model Elements

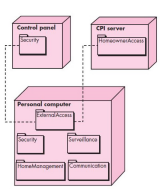
- Data elements
- E.g What information elements exist?
  - Data model --> data structures
  - Data model --> database architecture
- Architectural elements
- E.g. Floor plan of house
  - Application domain
  - Analysis classes, their relationships, collaborations and behaviors are transformed into design realizations
  - Patterns and "styles" (Chapters 9 and 12)
- Interface elements
- E.g. How doors, windows etc will look like
  - the user interface (UI)
  - external interfaces to other systems, devices, networks or other producers or consumers of information
  - internal interfaces between various design components .
- Component elements
- E.g. Set of detailed drawings, (and specifications) for each room in a house, i.e. wiring, switches, etc.
- Deployment elements

Architectural Elements

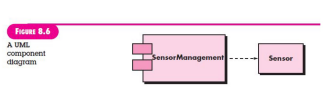
- The architectural model [Sha96] is derived from three sources:
  - information about the application domain for the software to be built;
  - specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
  - the availability of architectural patterns (Chapter 12) and styles (Chapter 9).

Deployment Elements

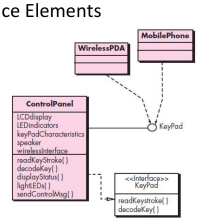
- Defines how software functionality and subsystems will be allocated within the physical computing environment that will support the software



Component Elements



Interface Elements



ARCHITECTURAL DESIGN

Software Architecture VS Software Design

↳ skeleton and infrastructure of software

↳ code level design
eg what each module does
the classes scope

Design Pattern

↳ Common problem solution

↳ Template

↳ reusable solution to a common occuring pattern

↳ categorized into 3

1. Creational Patterns
(creates object problems)

↳ object creation

↳ gives more flexibility to decide which objects to create

e.g creating a score
Singleton
factory
abstract factory
Prototype

2. Structural Patterns
(diff things joined)

↳ identifies ways to realize rs. among entities

↳ how to combine objects, classes to form large system

↳ r/s between entities

e.g using API
bridge
facade
Adapter

3. Behavioral Patterns
(how a function is working)

↳ identifies common comm. Pattern b/w objects and realize these Patterns

↳ comm. b/w entities

e.g chain of responsibility
Command
Interpreter
Iterator

Architectural Patterns

↳ serves as a blueprint for the software systems

Architectural Design

↳ how a software system should be organised and designing the overall structure of that system

↳ It is the critical link b/w design and req engineering

↳ as it identifies

↳ main structural components

↳ rs. b/w them

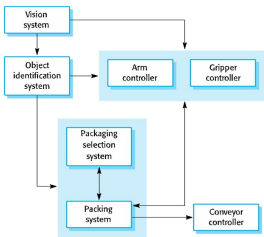
Agility and Architecture

↳ early stage of agile process is to design an overall system architecture

↳ Refactoring the sys. archi. is expensive

↳ as it affects so many components in the system

The architecture of a packing robot control system



Architectural Abstraction

1. Arch. in the small is concerned with

↳ arch. of indivisual programs

↳ ways the indivisual program is decomposed into components

2. Arch. in the large is concerned with

↳ arch. of complex enterprise systems → other systems, programs, program components

↳ these enterprise systems are distributed over different computers which may be owned/managed by diff companies

Adv of Explicit Architecture

1. Stakeholder comm.

↳ Arch used as a focous of discussion by system stakeholders

2. System analysis

↳ analyses wether system can meet non-functional req

3. Large-scale reuse

↳ the system arch is often same for systems with similar req so arch may be resuable across a range of systems

↳ product-line arch may be developed

(Product lines are applications that are built around a core architecture with variants that satisfy specific customer requirements).

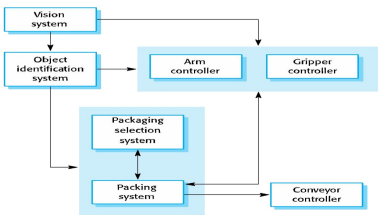
Architectural Representation

→ most used for documenting software arch

↳ simple, informal block diagrams

↳ showing entities + rs.

BOX LINE DIAGRAM



CONS

↳ lack semantics

↳ doesn't show

↳ type of rs. b/w entities

↳ visible properties of entities in arch

PROS

↳ useful for comm. with stakeholders and project planning

Practical Uses of Architectural Models

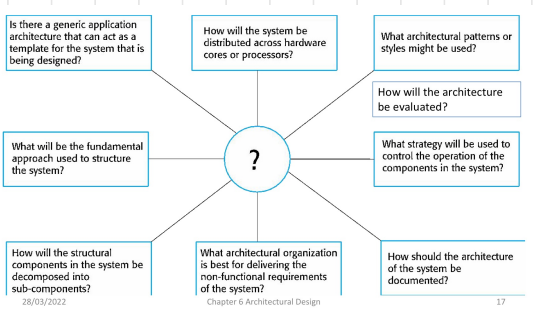
1. As a way of facilitating discussion about the system design
   - A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.
2. As a way of documenting an architecture that has been designed
   - The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections.

Architectural Design Decisions

↳ involves a seresie of decisions to be made that impact non-functional req of system

↳ while specific process may vary based on type of system

↳ common decisions are universal



Architecture reuse

↳ system in same domain often have similar arch.

- Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
- The architecture of a system may be designed around one of more architectural patterns or 'styles'.
  - These capture the essence of an architecture and can be instantiated in different ways.

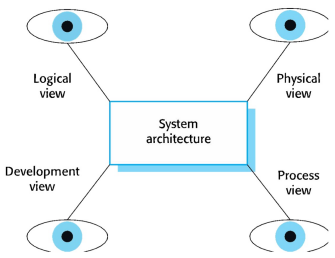
Architecture and system characteristics

1. Performance
   - ↳ localise critical operations
   - ↳ minimise comm.
   - ↳ use large rather than fine grain components
2. Security
   - ↳ use layered arch
   - ↳ with critical assets in the inner layers
3. Safety
   - ↳ localise safety critical features in a small no of sub systems
4. Availability
   - ↳ include redundant components and mechanisims for fault tolerance
5. Maintainability
   - ↳ use fine grain, replacable components

Architectural Views

↳ appropriate notation for describing and documenting system arch

- What views or perspectives are useful when designing and documenting a system's architecture?
- What notations should be used for describing architectural models?
- Each architectural model only shows one view or perspective of the system.
  - It might show how a system is decomposed into modules, how the run-time processes interact or the different ways in which system components are distributed across a network. For both design and documentation, you usually need to present multiple views of the software architecture.



4+1 View Model

1. Logical View
   ↳ shows key abstractions in the system as objects / classes

LPDP

2. Process View
   ↳ shows how, at run time, the system is composed of interacting processes

3. Development View
   ↳ shows how software is decomposed for development

4. Physical View
   ↳ shows the system hardware
   ↳ shows how software components are distributed across the processors in the system

Representing Architectural Views

↳ UML: Unified Modeling Language
   ↳ doesn't include abstractions appropriate for high level system description

↳ ADLs: Architectural Description Language
   ↳ have been developed but not widely used

Architectural Patterns

↳ means of representing, sharing and reusing knowledge

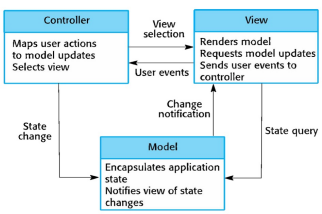
↳ tells into when and when not they are useful

- An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- Patterns should include information about when they are and when the are not useful.
- Patterns may be represented using tabular and graphical descriptions.

The Model-View-Controller (MVC) pattern

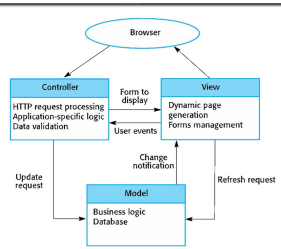
Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

The organization of the Model-View-Controller

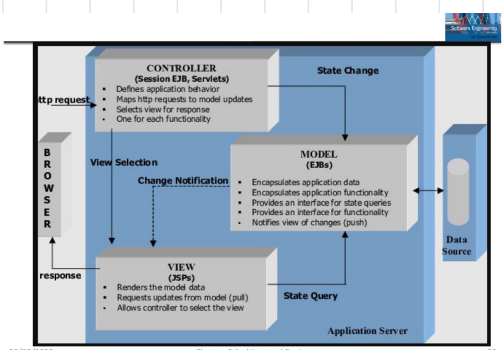


Conceptual view

Web application architecture using the MVC pattern



One possible process view



used:

↳ Multiple ways to view and interact with data

↳ future req are unknown

Adv

allows data to change independently of its rep.

CON

↳ additional code and complexity

e.g web based application

Layered architecture

- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often artificial to structure systems in this way.

The Layered architecture pattern

Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

used:

- building new facilities on top of existing
- development is spread across several teams
- each team responsible for a layer functionality

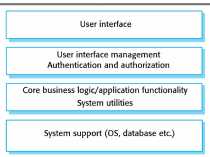
PROS

- can replace entire layer
- redundant facilities can be provided in each layer

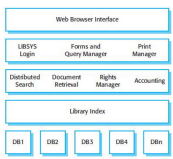
CONS

- hard for high ll to interact with low ll (need layer)
- difficult to provide clean seperation
- low performance

A generic layered architecture



The architecture of the LibSys system

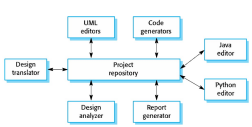


e.g Library System

Repository architecture

- Sub-systems must exchange data. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used a this is an efficient data sharing mechanism.

A repository architecture for an IDE



- May be active or passive.
- There is no need to transmit data explicitly from one component to another
- Components must operate around an agreed repository data model. Inevitably, this is a compromise between the specific needs of each tool and it may be difficult or impossible to integrate new components if their data models do not fit the agreed schema.

The Repository pattern

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

e.g IDE

USE

- large amount of sharing done
- large volumes of data generated and stored for long time
- data driven systems

PROS

- components can be independent
- change in 1 component can be propagated to all components
- data managed consistently
- no overhead of comm.

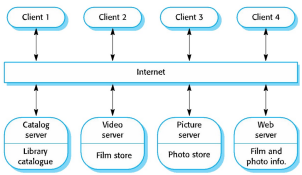
CONS

- Problems in repository affect all systems
- difficult to distribute repository across several computers

Client-server architecture

- Distributed system model which shows how data and processing is distributed across a range of components.
  - Can be implemented on a single computer.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.

A client–server architecture for a film library



eg film library

The Client–server pattern

Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

- It is easy to add a new server and integrate it with the rest of the system or to upgrade servers transparently without affecting other parts of the system.

use
↳ data to be accessed across diff locations

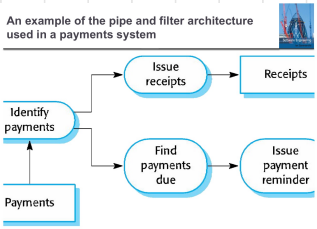
pros
↳ servers can be distributed across networks
↳ general functionality avl to all clients

cons
↳ server failure / Denial of service attacks
↳ unpredictable performance
(based on wifi)

Pipe and filter architecture

- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.

An example of the pipe and filter architecture used in a payments system



The pipe and filter pattern

Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.13 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

use
↳ data processing applications
↳ inputs are processed in seperate stages

cons
↳ format of data transfer has to be agreed upon
↳ high system overhead

pros
↳ easy to understand
↳ supports transformation reuse

e.g payments systems

Application architectures

↳ designed to meet an organisational needs

↳ due to shared characteristics among bussiness they often adopt a common architecture tailored to their req.

↳ It serves as a blueprint for a software system type adjustable to meet specific demands.

Software Architecture

↳ skeleton and infrastructure of software

Use of application architecture

↳ as a

- ↳ starting point for arch design
- ↳ design checklist
- ↳ means of assesing components for reuse
- ↳ vocabulary for talking about application types

Example of Application types

1. Data processing applications

↳ data driven applications that process data in batches w/o explicit user intervention during me processing

2. Transaction processing applications

E-commerce

↳ Data-centered applications that process user requests and updates info in a system database

3. Event processing applications

↳ Applications where system actions depend on interpreting events from the systems enviorment

4. Language processing applications

Compilers

↳ applications where users intentions are specified in a formal language that is processed and interpreted by me system

Application type example

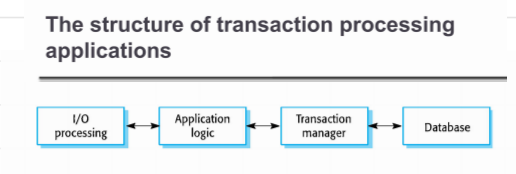
1. Transaction Processing Systems
   - ↳ E-commerce systems
   - ↳ Reservation systems
2. Language Processing Systems
   - ↳ compilers
   - ↳ Command interpreters

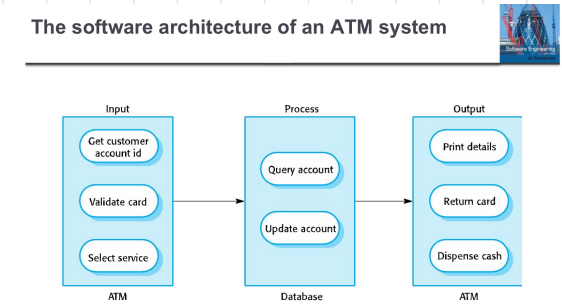
WIDLY USED Generic Application Arch

Transaction Processing Systems

- ↳ the system handles user req for info retrival or to update database
- ↳ Users Perspective for transaction
  - ↳ any coherent sequence of operations that satisfies a goal

e.g find the times of flights from London to Paris





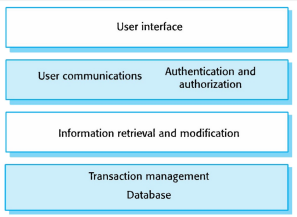
Information systems Architecture

↳ have generic arch that can be organised as a layred arch

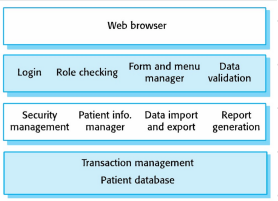
↳ these are transaction-based system → as interaction with these involves database transaction

↳ Layers include:
- ↳ user interface
- ↳ user comm.
- ↳ information Retrival
- ↳ system database

Layered information system architecture



The architecture of the Mentcare system



Web-based information systems

↳ where user interfaces are implemented using web browser

e.g Information / Resource management systems

- For example, e-commerce systems are Internet-based resource management systems that accept electronic orders for goods or services and then arrange delivery of these goods or services to the customer.
- In an e-commerce system, the application-specific layer includes additional functionality supporting a 'shopping cart' in which users can place a number of items in separate transactions, then pay for them all together in a single transaction.

Server implementation

↳ often multi-tier client/server arch

↳ web server is responsible for all user comm.,
with the user interface implemented using a web browser

↳ application server is responsible for implementing
- ↳ application-specific logic
- ↳ information storage
- ↳ retrival requests

↳ database server moves into to and from database and handles transaction management

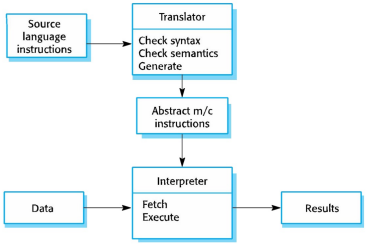
Language processing system

↳ accepts a natural or artificial language as input and generates some other representation of that language

↳ may include an interpreter to act on the instructions in the language that is being processed

↳ used in situations where describing an algorithm or system data is the simplest solution to a problem

The architecture of a language processing system

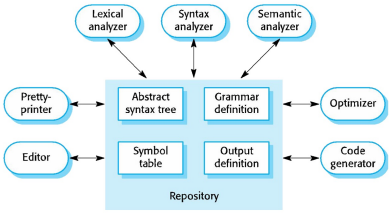


Compiler components

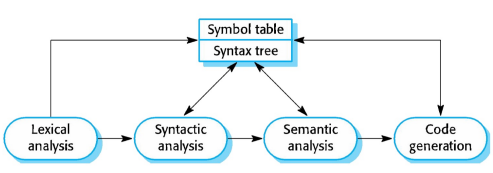
↳ a semantic analyzer that uses info from the syntax tree and symbol table to check the semantic correctness of the input language text

↳ a code generator that 'walks' the syntax tree and generates abstract machine code

A repository architecture for a language processing system



A pipe and filter compiler architecture



ESTIMATION

Estimation

↳ Project manager sets time req to complete the software among stake holders, the team, organizations management

↳ If expectations aren't realistic

Stakeholders won't trust the team or project manager

Elements of a sound Estimate

↳ To make a sound estimate, the project manager must have

↳ WBS → work break down structure

↳ effort estimate of each task

↳ list of assumptions made for the estimate → to deal with incomplete info

↳ consensus among the project team that estimate is accurate

Assumptions Make Estimates More Accurate

- Team members make *assumptions* about the work to be done in order to deal with incomplete information
  - Any time an estimate must be based on a decision that has not yet been made, team members can assume the answer for the sake of the estimate
  - Assumptions must be written down so that if they prove to be incorrect and cause the estimate to be inaccurate, everyone understands what happened
  - Assumptions bring the team together very early on in the project so they can make progress on important decisions that will affect development

Wideband Delphi

↳ a process to generate an estimate

↳ It's a repeatable process

↳ as it has steps that are performed the same way each time

↳ Project manager chooses an estimation team and gain conscensus among that team on the results

Steps

↳ specification and estimation form given

↳ Group meeting
to discuss estimation issues

↳ fill out forms anonymously

↳ distribute summary of estimates

↳ Group meeting
to discuss points where estimates vary

↳ fill out forms anonymously

Repeat till appropriate

1. Coordinator presents each expert with a specification and an estimation form.
2. Coordinator calls a group meeting in which the experts discuss estimation issues with the coordinator and each other.
3. Experts fill out forms anonymously.
4. Coordinator prepares and distributes a summary of the estimates
5. Coordinator calls a group meeting, specifically focusing on having the experts discuss points where their estimates vary widely
6. Experts fill out forms, again anonymously, and steps 4 to 6 are iterated for as many rounds as appropriate.

The Wideband Delphi Process

- Step 1: Choose the team
  - The project manager selects the estimation team and a moderator. The team should consist of 3 to 7 project team members.
    - The moderator should be familiar with the Delphi process, but should not have a stake in the outcome of the session if possible.
    - If possible, the project manager should not be the moderator because he should ideally be part of the estimation team.

- Step 2: Kickoff Meeting
  - The project manager must make sure that each team member understands the Delphi process, has read the vision and scope document and any other documentation, and is familiar with the project background and needs.
  - The team brainstorms and writes down assumptions.
  - The team generates a WBS with 10-20 tasks.
  - The team agrees on a unit of estimation.

- Step 3: Individual Preparation
  - Each team member independently generates a set of preparation results.
  - For each task, the team member writes down an estimate for the effort required to complete the task, and any additional assumptions he needed to make in order to generate the estimate.

Estimation form

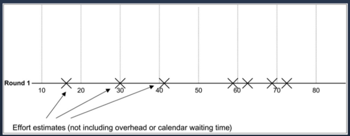
Name	Mike					Date 4/3/2004	Estimation form 1/1
Goal statement	To estimate the time to develop prototype for customers A & B						Units days
Category	☑ goal tasks ☑ quality tasks ☐ waiting time ☐ project overhead						
WBS# or priority	Task name	Est.	Delta 1	Delta 2	Delta 3	Delta 4	Total	Assumptions
1	Interview customers (A+B)	3	+2	+1				Needs off-site trip
2	Develop requirements docs	6	+5	-2	+1			Start from scratch
3	Inspect requirements docs	1	+2	+2	-2			Team of 4 BSAs
4	Do rework	1	+4					
5	Prototype design	20	-3	4	-2			Includes DB
6	Test design	5	+3					20% exists now
	Delta		+13	+5	-3			
	Total	36	49	54	51			

10

- Step 4: Estimation Session
  - During the estimation session, the team comes to a consensus on the effort required for each task in the WBS.
  - Each team member fills out an estimation form which contains his estimates.
  - The rest of the estimation session is divided into rounds during which each estimation team member revises her estimates based on a group discussion. Individual numbers are not discussed.

- Step 4: Estimation Session (continued):
  - The moderator leads the team through several rounds of estimates to gain consensus on the estimates. The estimation session continues until the estimates converge or the team is unwilling to revise estimates.
- Step 5: Assemble Tasks
  - The project manager works with the team to collect the estimates from the team members at the end of the meeting and compiles the final task list, estimates and assumptions.
- Step 6: Review Results
  - The project manager reviews the final task list with the estimation team.

- The moderator collects the estimation forms and plots the sum of the effort from each form on a line:

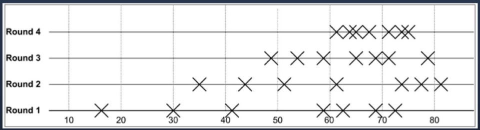


Summarized result of estimation

Goal statement	To estimate the time to develop prototype for customers A & B								
Estimators	Mike, Quentin, Jill, Sophie							Units days	
									Shaded items must be discussed
WBS# or priority	Task name	M.	Q.	J.	S.	Best-case	Worst-case	Avg.-hi & lo	Notes
1	Interview customers (A+B)	6	4	3	3	3	6	3.5	
2	Develop requirements docs	5	10	2	5	2	10	5	Discrepancy between Q. and J.
3	Inspect requirements docs	7	5	6	5	5	7	5.5	
4	Do rework	8	7	9	7	7	9	7.5	
5	Prototype design	28	23	31	25	23	31	26.5	
6	Test design	9	7	6	6	6	9	6.5	
	Total	63	56	57	51	46	72	54.5	

15

- The team resolves any issues or disagreements that are brought up.
  - Individual estimate times are not discussed. These disagreements are usually about the tasks themselves. Disagreements are often resolved by adding assumptions.
- The estimators all revise their individual estimates. The moderator updates the plot with the new total:



Other Estimation Techniques

- PROBE, or Proxy Based Estimating
  - PROBE is based on the idea that if an engineer is building a component similar to one he built previously, then it will take about the same effort as it did in the past.
  - Individual engineers use a database to maintain a history of the effort they have put into their past projects.
  - A formula based on linear regression is used to calculate the estimate for each task from this history.
- COCOMO II
  - In Constructive Cost Model, or COCOMO, projects are summarized using a set of variables that must be provided as input for a model that is based on the results of a large number of projects across the industry.
  - The output of the model is a set of size and effort estimates that can be developed into a project schedule.

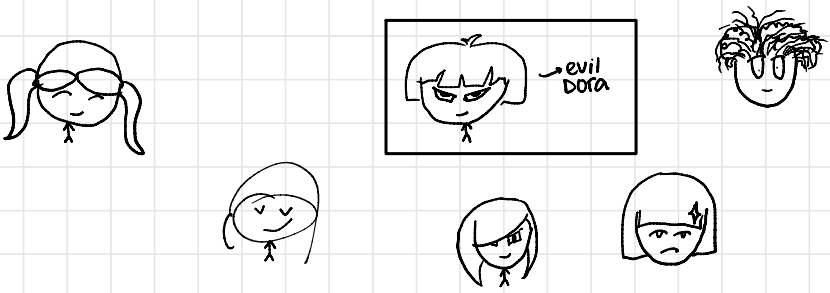
Key points

- A software architecture is a description of how a software system is organized.
- Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used.
- Architectures may be documented from several different perspectives or views such as a conceptual view, a logical view, a process view, and a development view.
- Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used and describe its advantages and disadvantages.

Key points

- Models of application systems architectures help us understand and compare applications, validate application system designs and assess large-scale components for reuse.
- Transaction processing systems are interactive systems that allow information in a database to be remotely accessed and modified by a number of users.
- Language processing systems are used to translate texts from one language into another and to carry out the instructions specified in the input language. They include a translator and an abstract machine that executes the generated language.

User Interface Analysis and Design



MISSING

Quality Management

CHP 24

Software Quality Management

↳ ensures required level of quality is achieved

↳ there are 3 principal concerns

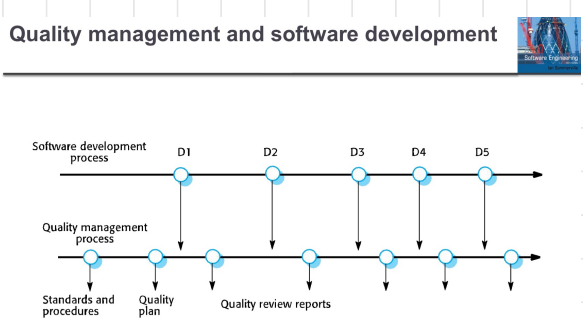
1. establishing a framework of organizational processes and standards that lead to high quality software
   ↳ a concern at the organizational level
2. application of specific quality processes and checking if these planned processes have been followed
   ↳ a concern at the project level
3. establishing a quality plan for a project
   The quality plan should
   ↳ set out quality goals
   ↳ define what processes and standards to be used
   ↳ a concern at the project level

Quality Management Activities

↳ provides an independent check on the software development process

↳ It checks the project deliverables to ensure they are consistent with organizational standards and goals

↳ quality team should be independent from the development team so they can report quality issues w/o biasnes



Quality Planning

- It sets out desired product qualities
- how these are assessed
- defines most significant quality attributes

Quality Plans

- It should define quality assessment process
  - which organisational standards should be applied
  - where necessary
  - define new standards to be used
- They should be short, succint docs → if too long no one will read them

Outline for quality Plans

- Product Introduction
  - description of the product
  - its intended market
  - quality expectations for the product
- Product Plans
  - the critical release dates and responsibilities
  - Plans for distribution and Product servicing
- Process descriptions
  - development and service processes to be used
  - Standards to be used for Product development and management
- Quality goals
  - quality goals and Plans for the produt
  - including identification and justification of critical product quality attributes
- Risks and risk management
  - key risks that might affect Product quality
  - actions to be taken to address these risks

scope of quality management

↳ It is imp for large complex systems

↳ for smaller systems

↳ quality management needs less documentation

↳ should focus on establishing a quality culture

quality documentation

↳ is a record of progress

↳ It supports continuity of development as the development team changes

Software Quality

Non functional characteristics

↳ defines subjective quality of a software system

↳ It reflects practical user experience

- This reflects practical user experience – if the software's functionality is not what is expected, then users will often just work around this and find other ways to do what they want to do.

↳ If the software is unreliable or too slow then it is impossible to achive their goals

Software fitness for purpose

- Has the software been properly tested?
- Is the software sufficiently dependable to be put into use?
- Is the performance of the software acceptable for normal use?
- Is the software usable?
- Is the software well-structured and understandable?
- Have programming and documentation standards been followed in the development process?

Quality conflicts

↳ Its impossible for a system to be optimized for all these attributes

e.g increase robustness ⟶ loss of performance

↳ Hence quality plan should define

↳ most imp quality attributes for the software

↳ include definition of quality assessment process

↳ an agreed way of assessing some quality

e.g maintainibility/ robustness is present in product

Software quality attributes

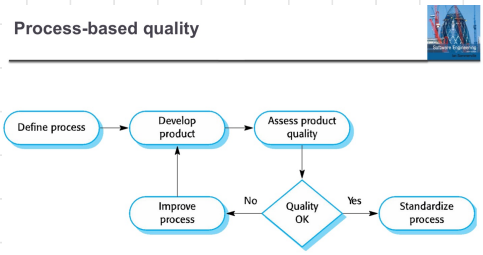
Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

Process and Product Quality

↳ Product quality is influenced by process quality

↳ It is imp as product quality attributes are hard to assess

- However, there is a very complex and poorly understood relationship between software processes and product quality.
  - The application of individual skills and experience is particularly important in software development;
  - External factors such as the novelty of an application or the need for an accelerated development schedule may impair product quality.



SOFTWARE STANDARDS

Software Standards

↳ defines required attributes of a product/process

↳ standards may be
  ↳ international
  ↳ national
  ↳ organizational/project standards

Importance of Standards

↳ Encapsulation of best practices
  ↳ avoids repetion of past mistakes

↳ Provide continuity
  ↳ new staff can understand the organisation

↳ Provides a framework defining what quality means
  ↳ organizations view of quality