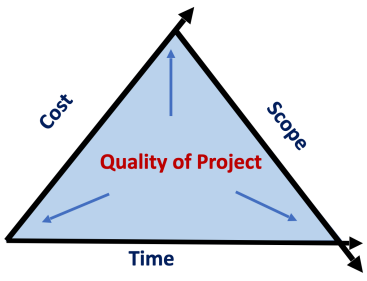
# Unit-2

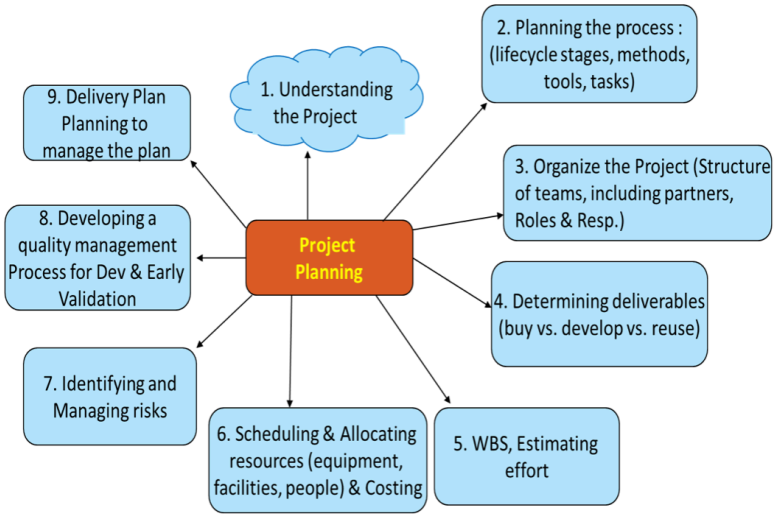
## Software project management fundamentals

- **Need for project management**
  - Large projects have **lots of people** working **together** for **prolonged time**
  - **Coordination** to ensure **integration** and **interoperability**
  - **Business and Environment change frequently** and rapidly
- **Software project**:
  - individual or collaborative enterprises
  - **carefully planned** to **achieve a certain aim** / **create a unique product** or service
  - **Characteristics of a project**
    - Made up of **unique activities** which do not repeat
    - **Goal specific**
    - **Sequence of activities** to deliver end-product
    - **Time bound**
    - **Inter-related activities**
    - **Need adequate resources** (time, manpower ,finance, knowledge-bank)
    - **Intangible;** can claim 90% completion without visible outcomes
- **Software project management:** planning and supervising projects;
  - **planning**, **execution** of plans, **monitoring** and **controlling** projects
- **Project management: planning**, **organising**, **motivating**, **controlling resources** to **achieve certain goals**; project workflow with team collaboration;
  - Planning, scheduling, monitoring, risk management, managing quality and people performance
  - Considers these **4 constraints in equilibrium**



## Software project management lifecycle


- **Initiation and approval**:
  - **Initiation**: happens at **approval or a "go" feasibility study** formally kicking-off
  - Opportunity or reason for project justifies its kick-off
  - A **charter** is created
    - **Purpose** of the project
    - How will it be **structured and executed**
    - **Vision, objectives**
    - **High level scope**
    - **Deliverables**
    - **Responsibilities** of the project teams and stakeholders
  - Initial **project owner/manager**, **budget** along with resources
- **Planning**
  - Focus on **what needs to be achieved and how**
  - **Looking ahead** and **making provisions** for required resources



- **Outcome:** Project plan
  - Level **depends on nature of project** (exploratory, research, development)
  - **Evolves** based on progress, context, risks, etc
  - **Perspectives**

| Sponsors | Customer | Execution Stakeholder |
|---|---|---|
| Projects role in organisation | Team understands problem? | Software Lifecycle to follow |
| Address customer requirement | Time to develop | Project Organisation (roles) |
| Investment vs Revenue | Cost of Solution | Standards, Guidelines, Procedures |
| Time expenditure and Risks | Delivery plans | Communication Mechanism |
| Responsibility and Progress Tracking | Metrics to indicate quality | Criteria to Prioritise requirements |
| Resources and Deliverables | Exit Criteria | Work breakdown and Ownership |
| Exit Criteria | Project support | |
| | Interaction/Review plans | |

- **Stages of Project Planning**
  - **Understanding expected deliverables** of the project
    - **Customer and stakeholder** expectations
    - **Market forces** driving the project ( new tech, competition)
    - **High-level decisions** (Make-Buy-Reuse)
    - Supported by **feasibility study** and **requirements elicited**
  - **Planning the process**
    - **Choice of lifecycle** is based on: activities, goals, time and so on
      - **Degree of certainty** (high or low) of **product, process, resource**
    - **Models, standards, guidelines and procedures**
      - **Standards**: Technical, interoperability, quality, regulatory
      - Config management, change management, quality plans

| Project Characteristics | Degree of Certainty | | | |
|---|---|---|---|---|
| Product | High | High | High | Low |
| Process | High | High | Low | Low |
| Resource | High | Low | Low | Low |
| | | | | |
| Expected Challenges | Realization Challenges | Allocation Challenges | Design Challenges | Exploration Challenges |
| Primary Focus | Execution focus | Controlling capacity | Designing the project in terms of Milestones, Personnel, Responsibilities etc. | Commitment of stake holders |
| | Optimize resource, efficiency and schedule | | | Maximize results |
| Development Model | Waterfall, V, CBSE | Waterfall, V, Iterative,CBSE | Iterative, Incremental | Incremental, Prototype, Agile |

- **Organize the project**
  - **Structure:** organise structure in terms of **people, team and responsibilities**
    - Eg: project manager, programmer, architects and so on
    - **Hierarchical , Flat, Functional, Matrix, Line orgs**
  - **Partners**
    - For project build, install, localisation, documentation, product management, product marketing, sales, pre-sales, support
    - **Downstream or Upstream**
- **Determine deliverables**
  - **Buy, develop, reuse**
- **Work break down (WBS)**
  - **Project activities** split into **smaller deliverable components** (iterative)
  - Split till **enough granularity**
  - **Hierarchical tree** structure: WBS (Work breakdown structure)
  - Aggregate tasks into **phases**

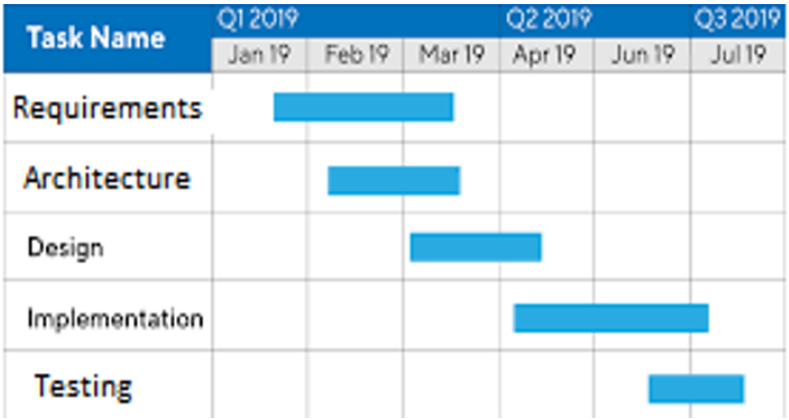
- **Milestone, checkpoints** and so
- **Entry and exit criteria**: Phases, milestones, etc
- Identification of **work packages for final product**
- **Estimation of tasks/activities** (effort and time)
  - Estimation helps in **efficient and effective control**
  - **WBS is estimation** of tasks in project
  - Estimation is done for **size and effort of WBS tasks** (time, cost and so on)
  - **Common estimates:**
    - **Lines of Code** (LOC)
    - **Function points**: business functionality in terms of functions
  - Top down or bottoms up
  - Estimation could be
    - **Experience based**: expert judgement, comparative studies
      - **Delphi, Modified Delphi**
    - **Empirical estimation:** Formula derived from past projects (size, process char, experience)
      - **CoCoMo (Constructive Cost Model)**
        - Three categories of projects
          - **Organic:** Small team, problem well understood, experienced people
          - **Embedded:** Large team, complex problem, need people with sufficient experience
          - **Semi-detached**: In between
        - **Estimation approaches**

$$\text{Effort } E = a_b(KLOC)^b \text{ Person months, Where}$$

$$\text{Time } T = c_b*(efforts)d_b \text{ Months}$$

- **Types of CoCoMo**
  - **Basic:** rough calculations
  - **Intermediate:** considers cost drivers
  - **Detailed:** factors dependencies

- **Scheduling and allocating resources**
  - Based on **outcomes of WBS and Estimation**
  - **Calendarization** of work activities
  - **Activities**
    - Brings **concerned individuals** to participate in **building schedule**
    - **Identification and allocation of resources** to WBS tasks
      - Hardware, software, humman
      - **Rework estimates;** schedule based on competencies
    - **Validate** upstream and downstream dependencies
    - **Organize concurrentl**y to **optimise people** and **sequentially** for **dependencies**
    - **Graphic tools**: gantt chart




    - **Identify schedule risks** and **mitigation plans**
    - **Minimise task dependency** (avoids delays)
    - Factor **working conditions** (holidays, work shifts, etc)
    - **Multiple iterations:** Due to feature requirements, Schedule, Resources, people or cost

        - **Cost** (Budget, capital and expenses )

- **Identify and Manage risks**
    - **Risk**: **unexpected** event that might **impact** people, tech and resources
    - Can be for **Products, projects, orgs**, etc
    - **Steps**
        - **Identify** risk (what may go wrong)
        - **Assessment and analysis** (probability of occurrence and impact)
        - **Mitigation**/Fall back plan
        - **Identification** and **catching** the **trigger**
        - **Execute** and **monitor mitigation** plan
- **Develop Quality management process**
    - Plans for **tracking progress**
    - **Communicatio**n plans
    - **Quality assurance** plans
    - **Test completion criteria**
    - **Early verification**/customer validation (Beta testing)
- **Plans for Tracking Project Plan and Delivery plan**
    - Plans for **managing project plan**
    - **Procedure for release** to customer
    - **Activities towards**
        - Staffing and people engagement
        - Compensation and benefits

- **Outcomes and Roles**

| **Outcomes** | **Roles** |
|---|---|
| Project Plan | Project Manager (at multiple levels) |
| Work Breakdown (to required granularity) | |
| Schedule (calendarized) | |

| Outcomes | Roles |
|---|---|
| Resource Management (Allocation, Balancing, Orchestrate for efficiency, dependency) | |
| Communication, Metrics, Checkpoints, Exit Criteria | |
| Risk Management and Quality Plan | |

- **Contents of Project Plan**
  - Introduction
  - Deliverables
  - Process model
  - Organisation
  - Standards, Guidelines, Procedures
  - Management activities
  - Risks
  - Staffing
  - Methods and Techniques
  - Quality Criteria/Assurance
  - Work Packages
  - Resources
  - Budget and Schedule
  - Change control Process
  - Delivery Means

- **Monitoring and Control**
  - Processes performed to **observe project execution** so that **potential problems** can be **identified** in a **timely manner** and **corrective action** can be taken
  - Begins once plan is created, runs **in parallel with execution**
  - **Continuously performed**; tasks, measures and metrics to **ensure project is on track** (time, budget, risk)
  - **Quantitative data**

- **Checkpoints, milestones, toll-gate**s
- **Project Management and Control dimensions**
  - **Time:** in terms of **effort** (man-months) and **schedule**
    - **Brooks law**: adding people to a late project delays it further
    - **Development models** help in managing time
  - **Information** (availability, propagation): **communication** including documentation
    - **Current state** and **changes agreed** upon
    - **Agile:** focuses on tacit knowledge held by people instead of docs
  - **Organisation** and structure, **roles and responsibilities**
    - Building a **team**
    - **Reorganising** structures
    - Clarify and manage **expectations**
    - **Reorient** roles and responsibilities
  - **Quality is built-in**
    - Designed in, not an afterthought
    - Frequent **interaction with stakeholders**
    - **Quality requirements** may cause **conflict**
  - **Cost, infrastructure and personnel** (capital and expense)
- **Includes**
  - **Monitoring and controlling** project work: **collect measures** and make **corrective/preventive actions**
  - Ensure **change controls** are meticulously followed
  - Scope, deliverables, documents are **updated**
  - **Control quality triangle** (Recollect triangle from page 1)
  - **Manage** team, performance and communication

- **Closure**
  - **Close project** and **report success** to sponsor
  - **Steps involved**
    - **Handover deliverable**s to customer and **obtain project/UAT** (User acceptance testing) sign-off from client
    - **Complete** and **pass on documentation**; **cancel supplier** contracts
    - **Release staff**, equipment; inform stakeholders
    - **Post mortem:** determine projects success and lessons learned

---

## Software architecture

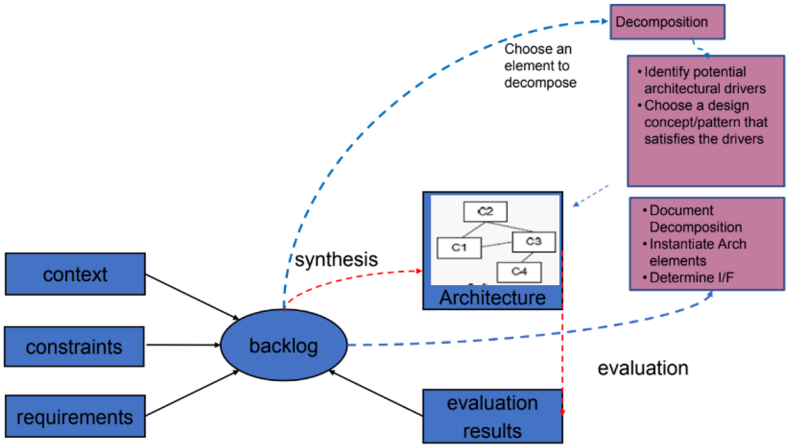
- **Architectural design: decomposition and organisation** of software into **components**
- **Detailed design: specific behaviour** of components in Architectural design
- Well written software is **more maintainable**
- To become a **master of software development**
  - Need to **learn the rules**
    - Algorithms, data structures, languages of software
  - **Learn the principles**
    - Structured programming, modular programming
    - Object oriented programming, generic programming
  - **Study architecture and designs of other masters**
    - Contain **patterns** that must be understood, memorised and used
- **Software architecture**
  - Top level **decomposition** into **major components** with a characterisation of **how they interact**
  - **Big picture depiction** of the system
    - Used for **communications among stakeholders**
    - **Blueprint** for system and project development

    - **Negotiations and balancing** of **functional and quality goals**

- **Importance of architecture**
    - **Manifests earliest design decisions**
        - **Constraints** on implementation
        - Dictates **organisational structure**
        - **Inhibits or enable quality attributes**, supports WBS
    - **Reuse** at architectural system level
    - **Helps in WBS** (reduce risk, enable cost reduction)
    - **Changes** to architecture **is expensive in later stages** of SDLC
- **Characteristics of software architecture**
    - Address **variety** of stakeholder **perspectives**
    - **Realises all of the use cases and scenarios**
    - Supports **separation of concerns**
    - **Quality driven**
    - **Recurring styles**
    - **Conceptual integrity**
- **Factors that influence Software Architecture**
    - Functional requirements
    - Data profile
    - Audience
    - Usage characteristics
    - Business priority
    - Regulatory/Legal obligations
    - Architectural standards
    - Dependencies and Integration
    - Cost constraints
    - Initial state

- Architect and staff background
- Technical and organisational environment
- Technical constraints
- Types of user experience

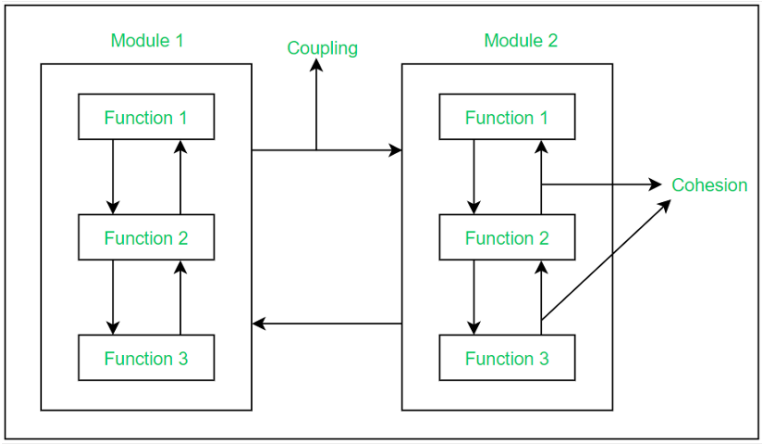
- **Architect**
  - Makes **high-level design choices** and **dictates tech standards**
    - Coding standards, tools and platforms
- **Architectural views, styles and patterns**
  - **Views:** ways of **describing software architecture**, enables **different stakeholders** to **view** it from the **perspective** of their interests
    - **Eg**: UI view, Process view
  - **Styles:** how the **subsystems and elements are organised**
    - Way of **organising code**; (Pipe & Filters, Client-Server, Peer-to-Peer)
  - **Pattern: known or proven approach** of **structuring and functioning**
    - Exists to “solve” a problem
    - **Eg:** MVC solves problem of separating UI from the rest
- **Architectural conflicts**
  - **Large-grain components improves performance** but **reduces maintainability**
  - **Redundant data**: improves **availability**; makes **security and integrity** difficult
  - **Localising** improves **safety** but **communication** degrades performance
- **Generalised model for architecting**



- **Common themes of architecture**
  - **Decomposition**: problem -> individual modules/components
    - **Approaches**:
      - **Based on layering**
        - Order **system into layers**; each layer **consumers service from lower** layer and **provides service to higher layers**
        - **Ordering of abstractions**
        - **Eg:** TCP/IP model
      - **Based on distribution** (computational resource)
        - **Dedicated task** owns **thread of control;** process need not wait
        - Many clients need access
        - **Greater fault isolation**
      - **Based on exposure**
        - How is the **component exposed** and **consumes other components**
        - Service offered, logic and integration
      - **Based on functionality**
        - **Grouping with problem domain** and s**eparate based on functions**
          - **Eg:** login module, customer module, so on
        - Mindset of operational process
      - **Based on generality**
        - Components which **can be used in other places** as well
      - **Based on Volatility**
        - Identify **parts which may change** and **keep them together** (UI)
      - **Based on Configuration**
        - Look at **target** for features needing to **support different configurations**
        - **Eg:** security, performance, usability

- **Based on Coupling:** keeping things together; Cohesion: things that work together
  - **Low coupling and high cohesion (good software always obeys this)**
- **Other approaches**
  - **Divide and conquer**
  - **Stepwise refinement**: simple solution and enhance
  - **Top-down approach:** overview of system, detail the subsystems
  - **Bottom-up approach:** specify individual elements and compose
  - **Information hiding**

- **Architectural views**
  - **4 ways to view**
    - **Structure of modules** (Module view point)

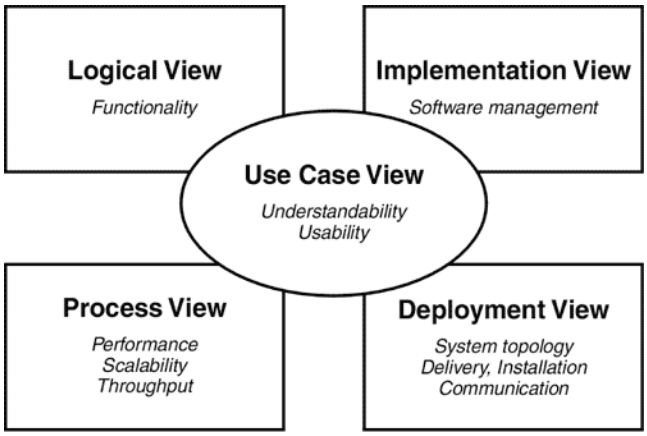



      - **Module:** unit of code with a functional responsibility
      - **Structure system as set of code units**
      - Architect enumerates **what units of software will have to do** and assigns **each item to a module**
      - **Larger module** may be divided into **smaller modules**
      - **Less emphasis** on how **software manifests at runtime**
    - **Component-and connector structure** (View point)
      - **Dynamic view** of system in **execution/runtime**

  - **Eg:** process view includes set of processes connected by sync links
- **Components/processing element**: software structure that **converts input to Output**; series of processes
  - **Computational**: does computation; Eg: function, filter
  - **Manager:** contains state+operations; State retained between invocations
  - **Controller:** governs time sequence of events
- **Connecting elements: glue** to component and data element
  - **Communication and sync link**
  - **Eg:** Procedure calls, RPCs, comms protocols, etc
- **Data element: information** needed **for processing/to be processed**
  - Memory in data; persistent

- **As allocation structure**
  - **Deployment structure: software** assigned to **hardware and communication paths**
    - **Relations:**
      - **“allocated-to”**: shows on which **physical units software elements reside**
      - **“Migrates-to”**: if allocation is **dynamic**
    - Allows to reason about **performance, data integrity, availability, security**
    - Important in **distributed or parallel systems**
  - **Implementation structure**
    - How **software is mapped onto file structures** in systems development, integration or config control envs
  - **Work assignment structure**
    - **Who is doing what** and the knowledge needed
- **Krutchens (4+1 view)**
  - **Use case view:** exposing requirements or scenarios

- **Design view:** exposes vocabulary of problem and solution space
- **Process view:** dynamic aspects of runtime behaviour
  - Threads and processes; addresses performance, concurrency
- **Implementation view**: realisation of the system; UML diagrams
- **Deployment view**: focus on system engineering

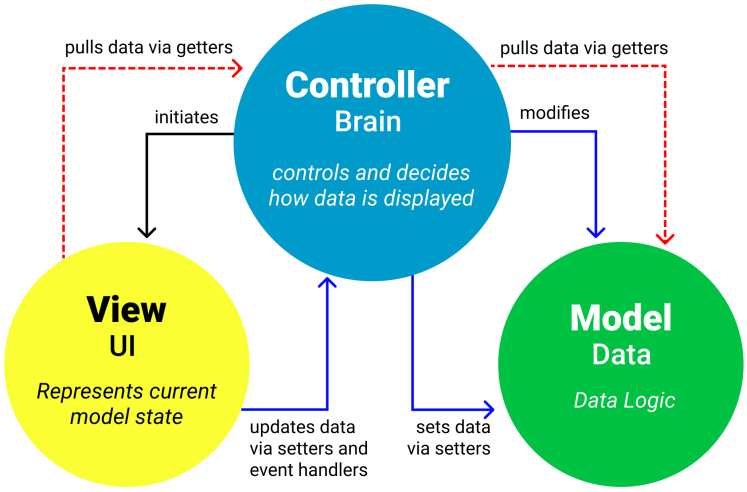



- **Architectural styles**
  - Way of **organisation of components**, characterised by **features** that make it **notable**
  - Used to **construct software modules**
  - **Structure and behaviou**r of the system
  - **Provides four things:**
    - **Vocabulary:** set of **design elements** (Pipes, filters, client, severs,)
    - **Design rules:** constraints that **dictates** how **processing elements** would be **connected**
    - **Semantic interpretation**: meaning of connected design elements
    - **Analysis**: performed on the system (Deadlock detection, scheduling)
  - **Recognised architectural styles**
    - **Main-program with subroutines**
      - **Generic:** traditional language-influenced style
      - **Problem:** system is hierarchy of functions; natural outcome of functional decomposition

        - **Top-level module is the main program** that invokes the rest
        - **Single thread of control**
    - **Context:** language with nested procedures
    - **Solution**
        - **System model:** procedures and modules are defined in a **hierarchy**
            - **Higher level modules call the lower level**
            - Hierarchy: strict or weak
        - **Components:** procedures resigning in main program; local and global data
        - **Connectors**: procedure call and shared access to global data
        - **Control structure:** single centralised thread of control
- Implicit invocation
- Pipes and filters
- Repository
- Layers of abstraction
- Client server
- Component based system
- Service oriented architecture
- Object oriented arch

- **Architectural pattern**
    - **Proven solution** to a **recurring architectural problem** (structuring, function and solutioning of subsystems)
    - Named collection of architectural designs that
        - Has **resulted in a successful solution** in a given development context
        - **Constrain** such **design decisions** that are specific to a particular system
        - Elicit **beneficial qualities** in each resulting system,
        - **Good starting point** for solution
        - **Number of layers** between user and data

- **Tiered architecture**
  - **Single tiered or monolithic**
    - Single app layer that supports UI, business rules and manipulation of data
    - **Eg:** microsoft word
    - Used in **client server apps**
  - **Two tiered arch**
    - **Client app:** business rules + UI
    - **Server app:** data retrieval + manipulation; physically separate system
    - Eg: SQL server
    - Uses in **traditional client-server App**
  - **Three tiered arch/Model - View - Controller**




    - **Model**
      - Central component; **application data**, **business rules**, logic and functions; **abstraction layer for features**
      - True domain object; no knowledge of UI
      - **Establish and collect information** to be displayed
      - Notify associated views and controllers of about change in state
    - **View**
      - **Output representation** of information (chart or diagram)

    - Graphics design and layout; get **information from model** as prompted by the **controller**

  - **Controller**

    - **Accepts input** and **converts it to commands** for model/view

  - **Use cases**

    - **UI logic** tends to **change more frequently** than **business logic**

    - App needs to **display same data in different ways**

    - Developing UI and Business logic require **different skill sets**; separate development teams easily

    - **UI code** is more **device-dependent** than business logic

---

## Software design

- **Design principles**

  - **Further decomposition** post architecture If necessary

  - **Description of behaviour** of components/sub-systems as identified in Architecture design

  - How **interfaces will be realised** (data structures+algorithms)

  - System will **facilitate interactio**n with user

  - Use of apt **structural and behavioural design patterns**

  - Maintenance and reuse

- **Techniques that enable design**

  - **Abstraction:** focus on essential properties

    - Expose only relevant functions

    - Procedural/data abstraction

  - **Modularity, coupling and cohesion**

    - **Modularity:** degree/extent to which large module is decomposed

      - Best to be self-contained

    - **Cohesion:** extent to which components are dependent on each other

    - Strong is good
    - **Could be**
        - Adhoc
        - Logical (input routines)
        - Temporal (initialisation sequence)
        - Sequential
        - Procedural (read and print)
        - Functional (contribute to same function)
  - **Coupling:** how strongly modules are connected
    - Loose is good
    - **Types**
        - **Content:** one component directly impacts another
        - **Common**: two components share overall constraints
        - **External:** components communicate through external medium
        - **Control:** one component controls the other (passes info)
        - **Stamp:** complete data structures are passed
        - **Data:** only one type of interaction
- **Information hiding**
  - **Need to know**; each module has a secret
  - Done via
    - **Encapsulation**: hides data and only allows access via specific functions
    - **Separation of interface and implementation:** define a public interface but separate details of how it is realised
        - Enables independence
- **Limiting complexity**
  - **Effort required to build** the solution (lines of code, depth of nesting)
  - Criterion to asses a design

- **Higher value => higher complexity => higher effort = worse design**
- Kinds of module complexity
  - **Intra-modular:** complexity of single module; based on size (LOC)
  - **Inter-modular:** between module
    - Based on size and structure
    - Eg: local flow, global flow and so on

- **Hierarchical structure**

- **Issues to be handled**
  - **Concurrency:** parallel execution of more than one program; deadlocks/race conditions
  - **Event handling**: messages sent between objects
  - **Distribution of components**
    - Distributed apps are supported by middleware
    - Consider communication breakdown
  - **Non functional requirement:** may have system-wide impact
  - **Error, exception handling, fault tolerance**
    - **Mistake diverts program execution** or **create incorrect result/action**
    - **Exception** could lead to **termination**
    - **Ensure that Faults** do not need **lead to errors** which **lead to system failures**
    - **Approaches:** fault avoidance, fault detection and removal
  - **Interaction and presentation**
    - React to user input **effectively and efficiently**
  - **Data persistance**
    - Storage of information **between executions**

- **Differences between architecture and design**

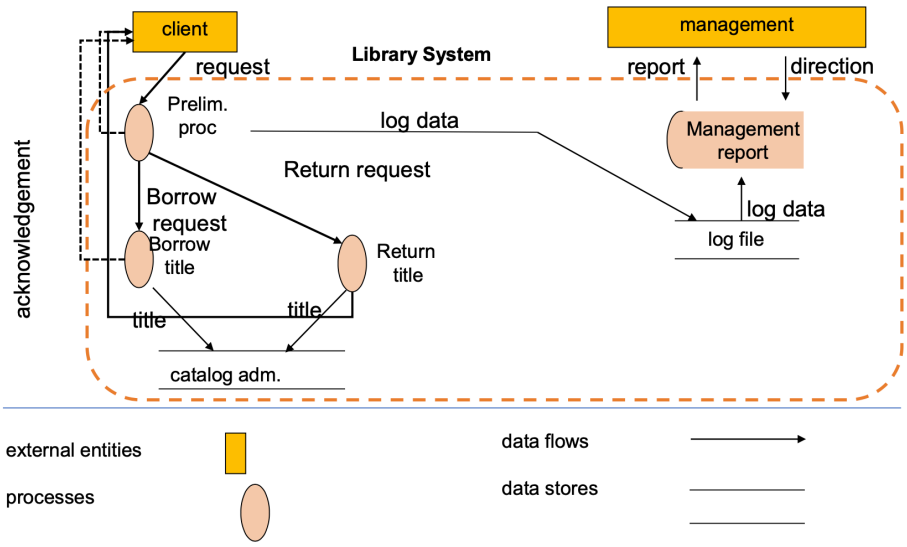
| Architecture | Design |
|---|---|
| Bigger picture; frameworks, tools, languages, scope, goals, etc | Smaller picture; local constraints, design patterns, programming idioms, code org |
| Strategy, structure and purpose | Implementation and practice |
| Software components, visible properties and relationships | Problem solving and planning for internal software |
| Harder to change | Simpler and have less impact |
| Influence non-functional requirements | Influence functional requirements |

- **Design methods:** support in **decomposing components** and representing **system requirements as components**
  - **Module hierarchy**
  - **Data flow design**
    - **Two step** process
      - **Structured analysis: logical** design; **data flow** diagrams
        - Logical consisting of set of DFD's augmented by minispecs and data dictionary
      - **Structured design**: transform **logical design into program structure**; **structure charts**
        - Heuristics based on **coupling and cohesion**
        - Transform centered
    - **Data flow diagram**
      - **External entity**
        - **Source and destination** of transaction
        - Located **outside** the domain
        - **Squares** in the diagram
      - **Processes**
        - **Transform** the data; depicted as **circle**
      - **Data stores**

- Lie **between processes** and places where **data structures** reside
- **Two parallel lines**

• **Data flows**

- **Data travels** between processes, entities, data stores
- Depicted as an **arrow**




- **Minispecs**: process in DFDs become sufficiently straight forward and **doesn't warrant further decompositions**
- **Data dictionary: contents** of DFDs after we are at **logical decomposed state**
  - Precise description of the structure of data

- **Design patterns**

• **Procedural patterns**

- **Analyse** problem **prior** to and **during construction**

| | |
|---|---|
| **Structural decomposition pattern** | Breaks down a large system into subsystems and complex components into co-operating parts, such as ***a product breakdown structure*** |
| **Organization of work pattern** | defines how components work together to solve a problem, such as ***master-slave and peer-to-peer*** |
| **Access control pattern** | describes how access to services and components is controlled, such as through a ***proxy*** |
| **Management pattern** | defines how to handle homogeneous collections in their entirety, such as a ***command processor and view handler*** |
| **Communication pattern** | defines how to organize communication among components, such as a ***forwarder-receiver, dispatcher-server, and publisher-subscriber*** |

- **Object oriented pattern:** Gang of Four solutions
  - **Creational patterns** that focus on creation of objects (Singleton, builder)
  - **Structural patterns** that deal with composition (Adapter, Bridge)
  - **Behavioural pattern** that describe interaction (Command, interpreter)
  - **Distribution patterns** that deal with interface for distributed systems
  - **Singleton pattern**
    - I**ntent**: only **one instance of class is created;** global access point to object
    - Motivation: **one object to coordinate** actions across the system
      - One class responsible to instantiate itself
      - **Global point of access** to the instance
    - Eg: centralised management of global resources

- **Anti Patterns**
  - Describes **situations** a developer should **avoid**
  - **In agile** approaches, **refactoring** is **applied when anti pattern** is **introduced**

- **Contrasting structural approach vs object oriented approach**

| Comparison Factor | Structural Approach | Object Oriented approach |
|---|---|---|
| **Abstraction** | Basic abstractions are real world functions, processes and procedures | Basic abstraction are not real world functions but data representing real world entities |
| **Lifecycles** | Uses SDLC methodology | Incremental or Iterative methods |
| **Function** | Grouped together, hierarchically | Functions grouped based on data |
| **State information** | In centralised shared memory | State information distributed among objects |
| **Approach** | Top Down | Bottom up approach |
| **Begin basis** | Considering use case diagram and scenarios | Begins by identifying objects and classes |
| **Decompose** | Function level decomposition | Class level decomposition |
| **Design approaches** | Use Data flow diagram, structured English, ER diagram, Data dictionary, Decision tree/table | Class, component and deployment for static design; Interaction and State for dynamic |

| Comparison Factor | Structural Approach | Object Oriented approach |
|---|---|---|
| **Design techniques** | Design enabling techniques need to be implemented | Communicates with objects via message passing and has design enabling techniques |
| **Design Implementation** | Functions are described and called; data isn't encapsulated | Components have attributes and functions; class acts as blueprint |
| **Ease of development** | Easier; depends on size | Depends on experience of team and complexity of program |
| **Use** | Computation sensitive apps | Evolving systems that mimic a business or business case |

## Service oriented architecture

- Make **software components reusable** via **service interfaces**
  - Utilise **common communication standards** and can be **rapidly incorporated** into new applications without deep integration
- Each **service** embodies **code and data integrations** to execute a **complete, discrete business function**; provide **loose coupling**
- Exposed using **standard network protocols** (SOAP - Simple object access protocol/HTTP or JSON/HTTP) to send **requests** to read or change data
- **Structured collections of discrete software modules** that collectively provide functionality
- **Benefits**
  - **Greater business agility, faster time to market**
  - Ability to **leverage legacy functionality** in new markets
  - **Improved collaboration** between business and IT
  - **Service reusability**
- **Service**
  - **Logical representation of a repeatable business activity** that has **specified outcome**
  - Discrete pieces of software written in **any language**
  - **"Callable entities"** accessed via **exchange of messages**
  - **Application functionality with wrappers**

- **Service characteristics**
  - Services **adhere to service contract**
    - Adhere to **communications agreement**
  - Services are **loosely coupled**: minimise dependencies
    - Maintain awareness of each other
  - Services are **stateless**
    - **Minimise resource consumption**
  - Services are **autonomous**
  - Service **abstraction**
  - Services are **reusable**
  - Services use **open standards**
  - Services **facilitate interoperability**
  - Services can be **discovered:** metadata
  - **Composed to form larger services**
- **SOA roles**
  - **Service provider: creates web services** and provides them to a **registry**
    - Responsible for terms of service
  - **Service broker/registry**: responsible for **providing information** about the service
  - **Service requester/consumer: finds a service** in the broker and **connects** to the provider
- **SoA vs Microservices**

| SoA | Microservice |
|---|---|
| Enterprise-wide approach to architecture | Implementation strategy with app dev teams |
| Communicates with components using Enterprise service bus (ESB) | Communicate statelessly using APIs |
| Less tolerant and flexible | More tolerant and flexible |