Introduction

Distruptive Innovation	Sustaining Innovation
Inferior solutions that under perform	Sees the market as existing customer base
Create new dimensions of value	Goal to better servces existing customers
Intially appeal to new/less demanding users	Enables high and usage
Then develops a growing flollowing as improved	99% of new technology

Opportunities are endless - Possible future **commercial** and **promotional** opportunities

The design process

- Requirements
  - **What** to be accomplished
  - **Requirements first**
    - Decide what want before design
    - Reduce risks
- Architecture
  - **Vision** how requirements can be realised
  - The **framework** for how to implement
- Detailed Design
  - Make component
  - Fitting together ## Mesauring good and bad
- Transcendental View
  - Recognise but not define
- User View
  - Fit for purpose

- Manufacturing View
    - Conformance to specification
- Product Biew
    - How well it put together
- Marketing View ($$$)

Goal: - Speed of development - Correctness of solution - Ease of comprehension - Elegance of design - Providing leadership - Being a great team player

Preamble

- Six blind men and the elephant: Small fixes can lead to bigger problems
- Six blind men and the elephant: Challenges in creating conceptual design
- The dead parrot sketch

Non-Function Properties

- Non-functional properties challenge
    - Hard to discover
    - Hard to quantify
    - Hard to test
    - Hard to fix
- **NFPs are contraints on the manner that system implements and delivers its functionality.** E.g.
    - Efficiency
    - Complexity
    - Scalability
    - Heterogeneity
    - Adaptability
    - Security
    - Dependability/Reliability
    - Dependability
- Function properties VS Non-Function properties
    - Products are sold based on FPs
    - NFPs play a critical role in preception

- Design guidelines
  - Provide guidelines that support various NFPs
  - Focus on **architectural level**
    - Components
    - Connectors
    - Topologies
- Non-functional requirements
  - Techinical constraints
  - Businss constraits
  - **Quality requirements**: Requirements about **quality** of a deliverable
    - Architectural choice driven by **Trade off** between Quality requrements
    - Quality requirements **Driven** by needs of the system
    - For operations:
      - Availability
      - Protability
      - Deployability
      - Security
    - When executing:
      - Performance
      - Througput
      - Scalability
    - For developers:
      - Simplicity/Readability
      - Testability
      - Maintainability
    - For architects
      - Integratability
      - Extensibility
      - Flexibility (Many ways of being used)
      - Product lines (Many ways of being created)
      - Users:
        - Usefullness
        - Usability
        - Trustability

        - Software Providers:
            - Rist
            - Budget
            - Novelty
            - Profit

Software Project Planning

- **Project Manager**
    - allocate resources
    - estimate feature
    - know (guess) what may go wrong
    - ensures crisis do not happen
- **Project Proposal**
    - more than an idea
        - Clearly descibed
        - Implementatable
        - Task with actions, schedules, people and budget
- **Project Status Report**
    - Motivational
    - Review of progress
    - record of overall development
    - **Should be undertaken in all projects**
    - Headings: Activities, problems, future activity, percentage of work done, planed events
- **Gross scheduling**: Identify the major activities with start times and when must be completed
- **Allocate resources to activities**: who, when, what to work (concurrently), no idle time
- **Funding and costs**
    - Direct labour
    - Direct materials
    - Special equipment
    - Capital expenses
    - Travel & training
    - Subcontracts
    - Support fee

    - Software costs
    - **Tracking costs**:
        - Budgeted cost, actual cost, forecasted cost
        - cumulative totals for these values
        - Sum these costs

- **Tools**:
    - Managing resources and activites: Timeline, MS Project
    - Accounting for expenditures: Excel
    - Timesheets

Software Architecture

- Architecture is
    - all about communication
    - parts
    - how parts fit together

- Architecture is not
    - about development
    - about algorithm
    - about data structure

- Software architecture is the conceptual fabric that **defines a system**
- **focuses on thoses aspects of a system that would be difficult to change once the system is build**
- Capture three primary dimensions:
    1. Structure
    2. Communication
    3. Nonfunction requirements

- Strategires
    - Logical order
    - Collaborative wiki for documenting

- Design process
    - Feasibility stage
    - Preliminary design stage
    - Detailed design stage
    - Planning stage

- **Abstraction**: A concept or idea not associated with a specific instance
    - Top down, Bottom up, Reification
- **Separation of Concerns**
    - Decomposition of problems
    - Concern spread/interacts with many parts (Scattering/Tangling)
- **Components**: Elements that encapsulate processing and data at an architectural level
    - Definition:
        - Architectural entity that:
            - encapsulates a subset of functionality
            - restricts access via explicit interface
            - has explicit environmental dependencies
- **Connectors**: An architectural entity tasked with effecting and regulating interactions between components
    - Connectors are often more challenging than components in larger heterogenous system
    - Often consists of method calls
    - provide application-independent interaction mechanisms.
- **Configurations**: An architectural configuration, or topology is a set of specific associations between the components and the connectors of system's architecture
    - Bind components and connectors together in a specific way
    - Differentiates components and connector from an implementable system
- **Topological Goals**:
    - Minimize **coupling** between components (less component know about each other)
    - Maximize **cohesion** with each component (component responsible for a logical service)
- **Cohesion**:
    - Relationship of code within a routine
    - Routines should do single thing
    - GOAL: Create routines with internal integrity
- **Coupling**:
    - connection between routines
    - loose coupling => small, direct, meaningful, visible, flexible relations

    - criteria
        - Size of interface (global data / complex params => bad)
        - Intimacy (param or return > database > message > gloabl data)
        - Visibility (by param > side effect)
        - Flexibility
        - Clearity (sensible names)
    - levels:
        - Simple-data coupling (simplest/Best)
            - Only non-structured data shared
            - All data passed as params/return result
        - Strong data-structure coupling(OK)
            - Structured data passed between routines
            - All data passes as params/return result
            - Most/all of structures passed relevant
        - Weak data-structure coupling (weak)
            - Little information in structures passed relevant
        - Control coupling (Poor)
            - Params tell called routine how to behave
            - Caller understands internal working of callee
        - Global data coupled (horrible)
            - Two routines operate on same global data
            - connection neither intimate nor visible
        - Pathologically coupled (Disaster)
            - One routine directly uses another routines code
            - One routine directly alters another routines data
- Size of a routine (100 - 150 lines best)

Programming tips

- Wrap malloc/realloc/free
- single exit routine
- single abort routine
- trap routine() for debugging
- Log routine

- Use assertions (without side-effects)
- Document changes and use version control
- Memory leakage
- Implement interfaces as interfaces
- Check for error returns
- Macros to support/faciliate change
- Nest macros
- Order params
- Prefix global and object state variables
- Distinguish pointers from non-pointers
- Use comments
- Code should be
    - re-entrant
    - thread safe
    - interrupt safe

Mythical

- The man-month as a unit for measuring the size of job is a dangerous deceptive myth
- **Brooks law**: Adding manpower to a late project makes it later
- Learn from your own mistakes and from the mistakes of others
- Rushing things may be counter productive
- Give people some time to excel
- Don’t assume everyone can do the same task in the same time
- Always be asking how you can get 10 times more out of people by giving them tasks they are good at
- Respect the architectural rules in place
- Too many cooks making different soups spoil the broth
- You can go a long way with small steps
- Small mistakes here, and small mistakes there, quickly add up to a big mess
- The Second Systems Effect: The second system is the most dangerous ever designed
- Talk is cheap
- Have to be able to monitor progress
- Not all who make promises keep them
- There are limits to power
- Teams that communicate succeed
- Teams that fail to communicate fail
- You can learn more by doing than reading
- What you don’t learn might cost you

- Know what is important and what is not
- It is not the volume of paper that is important, but what is written on it
- An early start might not be helpful but not starting never is
- Recognise your mistakes before they become costly ones
- Don’t reinvent wheels
- Knowing the tools can be as important as knowing the job
- Saying you’ll do something, and doing it are two very different things
- Small slippages tend to become bigger ones
- Success has many friends, but failure is an orphan
- Knowing must come before doing
- You can’t claim ownership of code you don’t understand
- Much of the challenge in design, is not the design but understanding how to design

Design

- Why design?
  - Communicate high level ideas
  - Backbone of the implementation
  - Provides top down view
  - Implementation is bottom up

- Retionalism VS Empiricalism
- Structured model:
  - **Waterfall model**: The waterfall model is a sequential (non-iterative) design process, used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of conception, initiation, analysis, design, construction, testing, production/implementation and maintenance.
  - **Co-evolution**
    - Structured sequence of better and prototypes
    - Early modelling reduces risk
  - **Open source**: The open-source model is a decentralized development model that encourages open collaboration
    - Chaotic see a need solve it, see a problem fix it
  - **Agile development**: software development under which requirements and solutions evolve through the collaborative effort of self-organizing cross-functional teams

        - Focus on the communication, not the policy

- The design team
    - Most great works are the product of one mind
    - Two person teams are magical
        - Ideas traded rapidly

    - Many person teams
        - Risk of offending the majority
        - Lack of conceptual integrity
        - A camel is a creature designed by committee

- Design aim
    - to translate requirements into deliverable
    - uncovering requirements
    - recoginising options

- **Good architectural design**:

    - Improve comprehension through metaphors
    - Consistency
        - Orthogonality: Do not link what is independent
        - Propriety: Do not introduce the immaterial
        - Generality

Architectural

- Conception:
    - Refinement and deduction process
    - Style of software

- Separation of concerns
    - Components
    - Communication
    - External software
    - User interace
    - Configurability
    - Security
    - ...

- Architectural Analysis
  - Goal: (completenenss/consistency/compability/correctness)
  - Scope: (different level/data exchange)
  - Concerns: (struct/behavior/linteraction/non-function)
  - Models
  - Type: (Static/Dynamic)
  - Automation level: (manul vs automated)
  - Stakehodlders

- Connectors
  - Procedure Call
    - Params:
      - Data transfer (reference/value)
      - Semantics
      - Return value

    - Entry point (multi/single)
    - Invocation:
      - Explicit (method/macro/inline/syscall)
      - Implicit (exception/callback/delegation)

    - Synchronicity (asynch/synch)
    - Cardinality
    - Accessibility (private/protected/public)

  - Event:
    - Cardinality (producers/observers/event patterns)
    - Delivery (best effort/exactly once/at most once/at least once)
    - Priority
    - Synchronicity (synch/asynch)
    - Notification (poll/publish sub/central/queue)
    - Causality (absolute/relative)
    - Mode (hardware/software)

  - Data Access:
    - Locality (thread/process/global)
    - Access (accessor/multator)
    - Availability
      - transient (register/cache/memory)
      - persistent (file/db)

        - Accessibility (private/protected/public)
        - Lifecycle
        - Cardinality

    - Linkage:
        - Reference (implicit/explicit)
        - Granularity (unit/syntactic/semantic)
        - Cardinality
        - Binding (compile-time/run-time/pre-compile-time)

    - Stream
        - Delivery
        - Bound (bounded/unbounded)
        - Buffering (buffered/unbuffered)
        - throughput
        - State (stateless/stateful)
        - Identity (named/unnamed)
        - Locality (local/remote)
        - Synchronicity (synch/synch)
        - Format (raw/structured)
        - Cardinality

    - Arbitrator
        - Fault handling
        - Concurrency
        - Transactions (nesting/awareness/isolation
        - Security (authentication/authorization/privacy/integrity/durability)
        - Scheduling (time/weight)

    - Adaptor
        - Invocation conversion
        - Packaging conversion
        - Protocol conversion
        - Presentation conversion

    - Distributor
        - Naming
        - Delivery
            - semantic (exactly/most/lest once)
            - mechanism (unicast/multicast/broadcast)

        - Routing
            - membership (bounded/ad-hoc)
            - path (static/cached/dynamic)

- **Hardware Architecture**

    - **RISC** (Reduced instruction set computer): emphasize small fast instruction set
    - **Pipelined / multi-processor**: emphasize the configuration of pieces of the hardware and overlapped flow of intructions and operation
    - **Micro-coded machines**: employ a fast interpreter to translate from user intruction set to underlying instruction set

- **Network Architecture**:

    - Topology: (Star/Token ring/Ethernet)
    - synch / asynch
    - unreliable / reliable
    - Layered communications protocol
    - ISO/OSI ethernet reference model

- **Data Flow Architecture**

    - **Pipes and Filters**: enforce dataflow, partition into step wise processes

        - **Pipelines**: restrict topologies to linear sequences of filters
        - **Batch Sequential**: each filter processes all of its input data before producing any output
        - Advantages:
            - Easy to understand
            - Support **reuse**
            - easily **maintained and enhanced**
            - permit **specialized analysis**
            - support **concurrent execution**

        - Disadvantages:
            - Not good for handling **interactive systems**

- **Abstract Layer Architectures**

    - **Domain Specific Software Architectures (DSSA)**: an assemblage of software components, specialized for a particular type of task (domain), generalized for effective use across that domain, composed in a standardized structure (topology) effective for building successful applications
    - **Software Layer Architecture**: localize changes to single components, seperate concerns into

different components.

        - Advantage:
            - Simple, divide and conquer
            - Clean, inependent layers

        - Potential problems
            - tight binding of code
            - Synchronous
            - Lack of hierarchical flow
            - Encapsulation denies access to important detail
            - Interrupts and Exception

    - **Overlaid Layered Architecture**:

        - Dynamically load code as needed
        - Free memory for reuse when no longer needed (Garbage collection)

    - **Microkernel architecture pattern**

        - consists of two types of architecture components: a core system and plug-in modules.
        - Application logic is divided between independent plug-in modules and the basic core system, providing extensibility, flexibility, and isolation of application features and custom processing logic
        - Minimal capabilities in a much reduced Kernel

    - Layered Style Examples:
        - Network commpunication protocols
        - Unix

    - Layered Style Advantages:
        - **Design**: based on increasing levels of abstraction
        - **Enhancement**: changes to function affects at most two other layers
        - **Reuse**: layer can be used interchangeably

    - Layered Style Disadvantages:
        - Not all systems works in layered fashion
        - Performance requirements may force the coupling of high-level functions to heir lower-level implementations
        - Adding layers increases the rist of error

- **Implicit Invocation Architecture**

- **MVC (Model View Controller**
  - Model expresses the application's behavior in terms of the problem domain
  - View responsible for presentation of model
  - Controller responsible for handling input and maintains model consistency
- **MVP (Model View Presenter)**
  - Model is an interface defining the data to be displayed
  - View is a passive interface that displays data
  - Presenter is responsible for querying the model and updating the view, reacting to user interactions updating the model
- **Why avoid 3-tier**
  - Putting all view management in the presenter makes the presenter ever fatter
  - Risks breaking presenter
  - Presenter doesn’t care about the views
- **Implicit Invocation Style**
  - Suitable for applications that involve loosely-coupled collection of components
  - Component announce event(s)
  - other components in system subscribe to event
  - When events fired, broadcasting system(connector) notify subed components
  - Example:
    - Debugger breakpoint
    - enforce integrity constraints in DB management system
    - used in forms to allow generic logic
  - Advantage:
    - Strong support for reuse
    - Eases system evolution
    - Easy to add new view
    - Eases system development
    - Case tools
    - Asynchronous
  - Disadvantages
    - Component unkown about which component respond to it
    - Feedback involves callback routines
    - No single defined flow of logic

        - Hard to consider all possible events
        - Hard to maintain and debug

- **Event Driven Architectures**

    - Promoting the production, detection, consumption of, and reaction to events.
    - Asynch behaviour model
    - Parallelism
    - Logic:
        - detail is performed at the highest level
        - Communication is via event generation

    - Example:
        - Mouse
        - AJAX
        - Windows O/S
        - State machine

    - Co-routines:
        - Parts cannot easily decomposed into sequential layerd operations
        - Seperate parts communicate with each other without loosing stack state
        - May run in separate theads.

- **Table Driven Architectures**

    - simplify and generalize applications by separating the program control variables and parameters from the code and placing those in separate external tables
    - Precompute behaviour in data at compile time
    - Imrpves performance of system
    - look up information in a table rather than using logic statements
    - pros&cons:
        - clean solution to difficult problems
        - Hard to grasp what is going on
        - Hard to making the transition from conventional code to table driven code
        - Hard to debug

- **Interpreter Architecture (Virtual Machines)**

    - Interpreter Style
        - Suitable for applications in which the most appropriate language/machine for executing the solution is not available

        - Interpreter implements how to do it
        - Preserves high level semantics
        - **Components**: include one state machine for the execution engine and three memories
            - current state of the execution engine
            - program being interpreted
            - current state of the program being interpreted
        - **Connectors**:
            - procedure calls
            - memory access
        - Examples:
            - Compilers (Java, clang)
            - Rule Based Systems (Coral, Prolog)
            - Scripting Languages (Awk, Perl, JavaScript)
            - Database plan
            - Micro coded machine
            - Cash register / calculator
        - Advantages:
            - Simulation of non-implemented hardware
            - Facilitates portability of application or languages across a variety of platforms
            - Behaviour defined by a custom language or data strucutre
            - Separates how do this from how do we say what it is we want to do
        - Disadvantage
            - Extra level of indirection slows down execution
            - Can't step outside language as interpreted

- **Message Oriented Architectures**
    - **Client Server**
        - Advantage:
            - Distribution of data is straightforward
            - Transparency of location
            - Heterogeneous platforms
            - Easy to add/upgrade servers
            - Functional client server interfaces
            - Simplifies distant levels of recursion

        - One server can support multiple clients

    - Disadvantages
        - Security
        - No central register of names/services
        - Hard to asynch communicate with server
        - Server can't initiate communication with clients
        - Overhead of packing and unpacking data in message
        - Potential restrictions on data types and structures
        - Uncertain server lifetime
        - Unpredictable server load

- **Remote Procedure Calls (RPC)**

    - IDL: Interface Definition Language
    - RPCGEN (client/server stub, communication, param unmashall/mashall)

- **Component Object Model (COM)**

    - Dynamic Linking of Objects at runtime
    - All method invocation is via obtained interface
    - **DCOM**: Provides a transparent remote COM interface

- **Broker Architecture**

    - Client shouldn't know how to reach
    - Security
    - Dynamic services
    - Broker act as an adaptor (Bridge)
    - load balancing
    - **Common Object Request Broker (CORBA)**

        - Language/OS independence
        - Interface mismatch translation (bridging)
        - Handle multiple servers(services)
        - Security
        - Encapsulate

    - **Message Bus Architecture**

        - For peer-peer connections, reduce application dependencies
        - All peers communicate with the bus

        - Bus provides share infrastructure

    - **Asynchronous Messaging**:

        - Peer-to-Peer:
            - Peers talk to other peers one at a time
            - Asynchronous handled by a callback

        - Publish-Subscribe
            - Provider communicates with Topic
            - Subscribers dynamically subscribe to topics
            - Topics forward messages to servers

    - Application Servers
        - Client/Browser Side:
            - HTML, JavaScript, Ajax

        - Web Tire (eg. Apache)
            - Receives request
            - Invokes web server-hosted components

- **Message Oriented Choices**

    - Delivery: Best effort, Persistent, Transactional
        - UDP, TCP/IP, Message Queues

    - Blocking / Non-Blocking / Asynchronous Sockets
    - Interface: Sockets, RPC, HTTP, SOA, Ajax, Stream

- **Representational State Transfer (ReST)**

    - URL + Params returns Document (GET, POST, PUT)
        - Stateless (except for cookies)
        - Server indicates cacheability

    - AJAX: Small document transfers
    - **RESTfull Web Services**

        - Web services only uses HTTP as transport layer
        - Representational State Transfer (REST)
            - Resources identified by a URL
            - Messages passing mechanism (GET, POST, PUT, DELETE)

        - ReST Constraints
            - Client/Server -> Parallel evolution
            - Stateless -> Scalable
            - Cacheable -> efficient
            - Layered -> Transparency
            - Mobile Apps -> Powerfull
            - Uniform Interface
                - Identification of all universal resources
                - Manipulation of resources (read, change, delete a named resource)
                - Self descriptive messages
                - Hypermedia

- **Semantic Web**: Efforrt to give XML entities real semantics

    - Namespaces disambiguate entity names
    - XMLSchema types the contents of an entity
    - RDF describes semantic information as graph
    - OWL defines ontological relationships
    - SPARQL permits queries about semantics

- **Service-oriented Architecture** (Web Services)

    - **A service-oriented architecture (SOA) is a style of software design where services are provided to the other components by application components, through a communication protocol over a network**
    - The basic principles of service-oriented architecture are independent of vendors, products and technologies
    - Fix interoperability problems
    - Provide internet-scale distributed systems
    - Design Considerations
        - Boundaries are explict
        - Services are autonomous
        - Services simply receive messages and reply

    - Guiding Principles: Discovery, Standardized Service Contract, Service abstraction, RPC
    - Implementation Goals: Stateless, Service reusability, Service autonomy

- **Web Services**

    - Common standard to support SOA
    - Offers:

        - Find out about a service (WSDL)
        - Find an instance of that service (UDDI)
        - Ask a service to do something (SOAP: Simple Object Access Protocol)
        - Assisting standards (WS-*)
    - XML based

REST	SOA
A simple client server interface	More flexible
What server does encoded in the message	Network of interacting components
Familiar well understood established standard	Access rich set of WS-* standards and features
Transparency with respect to client request	Reusable tool based architecture
Must know URL	Supports distribution/Replication
Circumvents firewall	Offers better security, reliability, interoperability
If you want it you create it	Complex evolving set of standards
Message paradigm	Layered architecture paradigm

- **N-tier Architectures**
    - **Three Tire**: Display <-> Process <-> State
        - Presentation tier: User interface, to translate results to something user can understand
        - Logic tier: application, process commands, make logical decisions and evalutions and performs calculations
        - Data tier: Information is stored and retrieved from a database or file system
    - **Multi Tier**:
        - More than tree tiers (Routing, Distributed processing)
        - Multi Tier VS Layered
            - Layered: based on logical division
            - Multi tier: Leverages client server, based on physical divison, more dependent on infrastructure
        - Advantage:
            - Code located whre most efficiently run

        - Scalability (distribution/replication)
        - Multi-threading support
        - Leverages existing technology

    - Disadvantages:
        - Communication overhead
        - Hard to perform regression testing
        - Hard to debug
        - Dependent on remote resources

- **Data Centric Architecture**

    - **Repository Style**

        - **Use a repository to separate the logic that retrieves the data and maps it to the entity model from the business logic that acts on the model. The business logic should be agnostic to the type of data that comprises the data source layer. For example, the data source layer can be a database, a SharePoint list, or a Web service.**
        - Suitable for applications in which hte central issue is establishing, augmenting, and maintaining a complex central body of **gloabl** information
        - Typically the information must be manipulated in a variety of ways. Often long-term persistence is required
        - **Components**: A central data structure representing the correct state of system, and a collection of independent components that operate on the central data structure
        - **Connectors**: Procedure calls or direct memory accesses.
        - **Blackboard Architecture**

            - **The Blackboard arrchitectural pattern is useful for problems for which no deterministic soluion strategies are known. In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution**

        - Consistency of global data: policed, controlled through locking
        - Database (SQL, OMDG, ...)
        - Data structure in memory / disk
        - Atomic Transaction
        - Concurrent computations and data accesses
        - Ruble based / Expert system
            - Repository is a collection of facts(rules), can be added, deleted
            - Satisfiability solvers

        - Example:
            - Information Systems
            - Graphical Editors
            - AI Knowledge Bases
            - Ipod

        - Advantage:
            - **Efficient** way to store data
            - **Sharing** modelCen is available as a shema
            - **Centralized management** (backup, security, concurrency control)
            - Ability to **easily extend** the data schema
            - Reliable

        - Disadvantage
            - Must **agree on a data model**
            - Difficult to **distribute data**
            - Data evolution is **expensive**
            - Scalability (read once write everywhere)

- **Distributed Repository**

    - **Scalable Data Architectures**

        - **Vertical Scaling**: Faster CPU, More memory, Larger Disks
        - **Horizontal Scaling**: More servers, Parallelism, Routing

    - Usage of Data
        - On-line Trasaction Processes (OLTP)
            - Large volumns of insert/update/delete
            - Simple select statements
            - GOAL: storage/retrieval: Transaction per seconds
            - operational data for control and run fundamental business tasks
            - short and fast inserts and updates by end users, relatively standardized and simple queries
            - Processing very fast and relative small if historical data is archieved

        - On-line Analytical Processing (OLAP)
            - Warehoused data
            - Very complex operations on the data
            - Goal: Knowledge: Reponse time per query
            - consolidation data, for problem solving and decision support
            - Periodic long-running batch jobs refresh the data, complex queries involving aggregations,

Space larger due to existence of aggregation structures and history data
        - Processing speed on amount of data involved

- **RDBMS SQL systems**

    - MySQL, Oracle, DB2, SQL Server, etc.

- **NoSQL systems**

    - **Apache Hadoop (Distributed File System)**

        - Open source in JAVA
        - Replication, Load Balance, Multiple Servers
        - Replicates code execution at each server
            - Map: (grouping in SQL)
                - Select, Transform, Distribute to servers for reducing
                - Compute form each data recorded a key
                - Sort data by generated key on each machine

            - Reduce
                - Each server aggregates/reduces its own data
                - Results form one or more servers merged
                - Compute a summary value for each key
                - Then reduce to summary value across machiens

        - Advantage:
            - Divide and couquer
            - Highly parallel and scalable
            - Robust / Reentrant
            - Replication

        - Disadvantage:
            - Must write Map and Reduce Logice from scratch
            - Have to wait for all parallel activity to complete
            - Not a database solution

    - **MongoDB** (NoSQL)

        - Distributed database system
        - Object Oriented DB
            - Each record encodes Binary JSON (JOSON is a way of describing nested structure/content)

        - No Meta Schema

    - Indexed searching supported
    - Replication, load balancing, sharding
    - Advantage:
        - Uses JSON while still supports transaction
        - Uses Map Reduce like Hadoop

    - Disadvantages:
        - Custom query language
        - Complexity of stored data structures
        - Not as flexible as RDBMS
        - No clear meta schema
        - Risk of storeing erroneous data

- **Redis**

    - Main meory storage
    - Pros:
        - Supports both fast memory read and update
        - Can store objects referenced by name
        - Can be distributed across many machines

    - Cons:
        - Database must fit in memory
        - Recovery is by periodically back of chagnes to a disk log

- **Cassandra**

    - Pros:
        - Data duplication across nodes
        - Improves reliability and recovery
        - Row content can be distributed across nodes
        - Parallel searching of row content
        - Fast Map/Reduce capabilities

    - Cons:
        - Record Read/write grows linearly with distribution
        - Doesn't support relational joins

- **Elastic Search**

        - Pros – Can search HTML text – Supports indexing of that text – Supports distribution of the indices
        - Cons – Lacks distributed transactions – If documents change indices must be rebuilt?

    - **CouchDB**

        - Databases are documents
        - Pros – Documents can change over time – Documents may be offline – Changes are timestamped – Atomic, consistent, isolation, durability (ACID)
        - Cons – Applications responsible for enforcing ACID

    - **Accumulo**

        - Built on top of Hadoop
        - Pros – Big table store – Different security levels for column info – Map Reduce functionality
        - Cons – Uncertain future

- Other data mining strategies
    - Genetic Programming: Try to find random formula's that signal something interesting
    - Gradient Descent: Construct a cost formula for how close answer is to desired answer, find optional parameterization
    - Regression - Fourier Transform: Prediction of future patterns

- Methodology: Filter (clean up), Train (optimise logic on data given truth), Validate (prove that not over fitting or under-fitting), Use
- CAP: Consistent, Available, Partition Tolerant
    - Consistent: the data (when read) will always be consistent
    - Available: system is always available
    - Partition Tolerant: system can handle lost messages, split, down nodes
    - can only choose two
    - Basic Proof
        - Consider two partioned nodes:
            - Write new record to one node
            - Then read this same record from the other node

        - If partitioned other node can't know latest version of record to be read
            - Either return earlier version
            - Or wait this node to obtain latest version (blocking)
            - Or insist both nodes must remain up all the time (not partition tolerant)

- **Process Control Architectures**:

- **Process control is an engineering discipline that deals with architectures, mechanisms and algorithms for maintaining the output of a specific process within a desired range**
- **Process-Control Style**
    - Suitable for applications whose purpose is to maintain specified properties of outputs of the process at given reference values
    - Components:
        - **Process Definition** includes mechanisms for manipulating some process variables
        - **Control Algorithm** for deciding how to manipulate process variables
    - Connectors:
        - **Process Variables**: *Input variable* measures an input to the process, *Manipulated variable* whose value can be changed by the controller, *Controlled variable* whos value the system is intended to control
        - *Set Point* is the desired value for a controlled variable
        - *Sensors* to be obtain values of process variables pertinent to control
- **Open-Loop Control System** (Non-feedback System)
    - Information about process variable is not used to adjust the system
    - Lack of feadback problems:
        - No regulation of software behaviour
        - Fixed behaviour in response to chagne requet
        - Disturbances in behaviour not considerd
        - No check that actual close desired
- **Feed-Back Control System**
    - The controlled variable is measured and the result is used to manipulate one or more of the process variables
    - Benefits:
        - Self checking
        - Self improving
    - Example:
        - Prediction
        - Monitor accuracy of prediction
        - Employ mixture of experts
- Process Control Examples
    - Real-Time System Software to Control: Automobile anti-lock brakes, Nuclear power plants,

Automobile Cruise-Control
    - Hardware circuits that implements clocks
    - Logic circuits: employ feedback
    - Quantum circuits

- **Rule Based Architectures**

    - **Interative enhancement style**

        - Start by writing a very dumb program, keep adding logic which makes it less dumb, terminate when can't improve behaviour of resulting logic.
        - Pros:
            - Allows concurrent design and development
            - Can lead to surprising intelligence

        - Cons: Hard to predict how successful exercis will be
        - Example:
            - Bridge program:
                - deal hand,enforce basic rules of play
                - add sensible rules for how to play well
                - Logic identifies the least worse card to play based on huge number of empirical rules drawn from observation of codes prior behaviour

            - Cribbage
                - Table driven discard logic

            - Pegging
                - For each card can play
                - For every possible card played in response

- **Model Driven Architectures**

    - Architecture is formalised in a (computer readable) model
    - A computer tool produces the implementation
    - Pro:
        - Consistent mapping from design to implementation
        - Efficient approach to software development

    - Cons:
        - Limited by power of the model and translation tool
        - Hard to test, maintain and debug

- Model driven Examples:
  - RPCgen / DCOM: Marshalling/unmarshalling interfaces
  - Object Management Group
  - Liquil>

- **Other Architectures**

  - C2:
    - Architecture built using components & connectors
    - Both have a top and a bottom – Top and bottom of component has ≤ 1 connector – Opposite ends of component/connector connect – Other end of connector connects to anything – All communication is via messages
    - Requests travel up
    - Connectors are responsible for:
      - Routing, broadcasting, message filtering – Domain translation when necessary

    - Objectives:
      - Substrate independence
      - Message based architecture
      - Different components share no state
      - Components are compartmentalized

  - Aspect oriented architecture
    - Cross cutting concerns – Runtime assertions – Tracing (event logging) – Profiling (performance issues) – Error handling – Testing – Debugging – Reliability (buffer overflow etc) – Encryption mechanisms
    - Extend the language to specify aspects
      - Can be implemented statically during compilation
      - And/or dynamically at runtime via request

Death March

- Death march projects are rarely billed as such. They can be hard to identify
- Why happen?
  - Politics
  - Naive promises
  - Cuts in project schedules
  - Start up mentality
  - Outside pressures
  - Unexpected crises

- Procrastination

Design Patterns

A design pattern is a general solution to a common problem in a context

Creational	Structural	Behavioural
Factory Method	Adapter	Template
Abstract Factory	Bridge	Strategy
Builder	Composite	Command
Singleton	Decorator	State
Multition	Facade	Visitor
Object pool	Flyweight	Chain of Responsibility
Prototype	Front controller	Interpreter
	Proxy	Observer
		Iterator
		Mediator
		Mememto

- Good design principles – Program to interfaces not to an implementation – Separate what changes from what does not – Encapsulate what varies behind an interface – Favor composition over inheritance – Loosely couple objects that interact – Classes should be open for extension, but closed for modification – Each class should have one responsibility – Depend on abstractions, not concrete classes
- Good use of Design Pattern Principles
  - Let design patterns emerge from your design, don't use them just because you should
  - Always choose the simplest solution
  - Always use the pattern if it simplifies the solution
  - Know all the design patterns out there

- **Creational Design Patterns**

  - **Factory Design Pattern**

        - **The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate**
        - Goal:
            - Separate what changes from what does not
            - Encapsulate what varies behind an interface
            - Depend on abstractions, not concrete classes
            - Loosely couple objects that interact
            - Design should be open for extension but closed to modification
        - Problem of myClass = new MyClass():
            - may be replaced by MyBetterClass()
            - many subclasses might exists
            - may not know what to create until runtime
        - Factory Solution
            - Wrap code that creates objects inside a method
            - make createMyClass static
        - Advantages:
            - Protects higher level code from lower level detail
            - Any change to class created need once
            - Create object dynamicly
            - Override create method in subclasses
    - **Abstract Factory Pattern**:
        - **The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes**
        - **Abstract Factory patterns work around a super-factory which creates other factories**
        - create a factory to create the object but form a very different framework.
        - Advantage:
            - Protect from creating components from many frameworks that cannot talk to each other
            - Easy to decide at run time which framework to use
            - Framework chosen is transparent to all higher level code
            - Easy to add new frameworks
        - Example:
            - createPizza() for both Domino's pizza and Pizza Hut's pizza which is a subclasses of PizzaStore class
    - **Builder Design Pattern**

        - **The Builder Pattern encapsulates the construction of a product and allows it to be constructed as a sequence of steps**
        - An bstract super class have interface for building a planner, concrete builder class creates real products
        - Advantage:
            - Programming to an interface
            - Objects built according to actions, not code
            - The composite objects are be constructed differently whitn the build transparently
            - Useful structures such as indices, hashing can be implemented inside builder
            - Extra layer of indirection

- **Singleton Design Pattern**

    - **The Singleton Pattern ensures that a class has only one instance, and provides a global point of access to it**
    - Ensure the assignment of singlement instance is thread safe
    - Use a stati factory method to construct it, the factory return it if already constructed. Constructor and destructor is private to the class.

- **Multition Pattern**

    - **Generalisation of the Singleton Design Pattern, create a singleton object associated with a key enforcing one-to-one mapping**

- **Object Pool Design Pattern**

    - **Factory maintains a list of contructed object, creation involves returning an object from pool if it exists else creating a new object. When delete, add it to pool for later reuse**
    - Advantage:
        - Malloc/new are expensive operations
        - Reducing the need to create can gain significant performacne improvement
        - Can return elements of a large array instread of creating each element independently
        - Manage the maximum number of objects of a type created
        - Control number of threads created

- **Prototype**

    - **Collection of proto-typical instance of a class exist. To create a new instance, cloned from its prototype. If class has no state, it can be reused in many places**
    - Advantages:
        - How class instance is created is determined by an example

        - Simplify the construction of a complex object
        - New prototypes can be added
        - Choose the prototype they want
    - Disadvantage:
        - Copying an object can be complicated

- **Structural Design Patterns**
    - **Adapter**
        - **Adapter pattern works as a bridge between two incompatible interfaces. Involves a single class taht responsible to join functionalities of independent or incompatible interfaces**
        - **Object adapters**: use composition
        - **Class adapters**: user multiple inheritance
        - Participantes (example)
            - Client: calls the adapter which is isolated from the adaptee
            - Adapter: Forwards calls between client and adapter, it may have to interact with multiple classes
            - Adaptee: A component being adapted
        - Have to watch for potential exceptions when there is **no match** betwen old and new interfaces
    - **Bridge Design Pattern**
        - **This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other.**
        - Separates abstraction from its implementation, abstraction can be different from the implementations, use code to bridge the differences
        - Useful when the implementation might be selectable orswitchable at runtime.
    - **Composite Design Pattern**
        - **allows you to compose objects into tree structures to represent part- whole hierarchies, lets client treat individual objects and composition of objects uniformly, irrespective of how the objects within the hierarchy differ**
        - **This pattern creates a class that contains group of its own objects. This class provides ways to modify its group of same objects.**
        - Subclass from this common functionality
    - **Decorator Design Pattern**

- **Decorator attaches additional responsibilities to an object dynamically, it provide a flexible alternative to sub classing for extending functionality**
- **This pattern creates a class that contains group of its own objects. This class provides ways to modify its group of same objects.**
- Advantages:
  - Provides an alternative to sub classing
  - Decorators change behaviour of their component by adding functionality before/after/in place of method calls to their child.
  - Component can have any number of decorators
  - Add state to the thing decorated
  - Presenccce of decorators are transparent
  - Easy to modify behaviour of a decorator, not affect other things.

- **Facade Design Pattern**

  - **hides the complexities of the system and providea unified, higher-level interface to a whole module making it easier to use**
  - **involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.**
  - Participantes: Subsystem, classes
  - Collaborations: Clients interact subsystem via Facade
  - Consequence: Shilds clients from subsystem components, promotes weak coupling
  - Adapter VS Facade:
    - Adapter: change an interface so it matches one the client needs
    - Facade: Provide a client with a simplified interface to subsystem

- **Flyweight Design Pattern**

  - **Flyweight pattern tries to reuse already existing similar kind objects by storing them and creates new object when no matching object is found**
  - **Extract what changes from what does not**
  - Flyweight is stateless permitting sharing/reuse
  - State can be stored in array, on disk, or in database

- **Front Controller Design Pattern**

  - **provide a centralized request handling mechanism so that all requests will be handled by a single handler**
  - **This handler can do the authentication/ authorization/ logging or tracking of request and then pass the requests to corresponding handlers**

        - Front Controller: Single handler for all kinds of requests coming to the application
        - Dispatcher: Front Controller may use a dispatcher object which can dispatch the request to corresponding specific handler
        - View: Views are the object for which the requests are made
    - Advantages:
        - Avoids having to repeat code
        - Centralises routing and so isloates low level logic from client
        - Simplifies navigation issues
        - Can spawn thread for bulk of work
        - Can preserve state on client
- **Proxy Design Pattern**
    - **create object having original object to interface its functionality to outer world**
    - Uses:
        - Remote: transparent communication with object
        - Virtual: creates real object only when needed
        - Protection: refuses access unless permitted
        - Stub generation in RPC

- **Behavioural Design Patterns**
    - **Template Design Pattern**
        - **Abstract class exposes defined way(s)/template(s) to execute its methods. Subclasses may override the method implementation but the invocation is to be in the same way as defined in abstract class**
        - **Template Method defines the skeleton of an operation with many steps. Template Method lets subclasses redfined theses steps without changing the behaviour.**
        - Code to be reusable but details may differ
        - Advantages:
            - Separates what chagnes from what does not
            - Bulk of unchanging code can be reused
            - Dailed difference are that subclass worry about
    - **Strategy Design Pattern**
        - **The Strategy Design Pattern defines a family of algorithms, encapsulates each one, and (by using a common interface) makes them interchangeable. Let algorithm vary independently of the clients that use it**

        - **create a Strategy interface defining an action and concrete strategy classes implementing the Strategy interface. Context is a class which uses a Strategy**
        - Advantages:
            - Assignment of behaviour done at run time
            - Easily increase types of behaviour with no new class
            - Code need to know how to invoke behaviour but not how actual behaviour is implemented
            - Behaviours can be shared, reused and dynamically changed
            - Can modify object without touching behaviour
            - Can modify behaviour without impacting object

- **The Command Design Pattern**

    - **The Command Pattern encapsulates a request as an object letting parameterize other objects with different requests, queue or log requests, and supports repeat and undoable operations.**
    - **A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the object which can handle the command and passes the command to object to execute.**
    - Advantages:
        - Create no-op if no action required
        - A concrete command may itself be targeted at different receivers
        - Can queue up future commands
        - Can cache commands to disk and repeat
        - Have a common interface with an object passed containing parameters or actual parameters to execute

- **State Design Pattern**

    - **In State pattern a class behavior changes based on its state. This type of design pattern comes under behavior pattern. Create objects which represent various states and a context object whose behavior varies as its state object changes.**
    - Represent each state by a class having interface
    - Current state is referenced using strategy pattern
    - Advantage:
        - object have behaviour predicated on state
        - behaviour is a given state encapsulated within a single class
        - less risk of failing to handle an action when in an arbitrary state
        - Easy to change or extend the state

- **Bisitor Design Pattern**

        - **Visitor class changes the executing algorithm of an element class. Execution algorithm of element can vary as and when visitor varies. Element object accept the visitor so visitor object handles the operation on the element object.**
        - **add capabilities to a composite of objects and encapsulation within these objects is not important, undesirable, or not possible**
        - Advantages:
            - Separate what changes from what does not
            - Classes open for extension; closed for modification
            - Each class should have one responsibility

    - **Chain of Responsibility Design Pattern
        - **creates a chain of receiver objects for a request. This pattern decouples sender and receiver of a request based on type of request**
        - give a sequence of objects in turn a chance to handle a request, Each object in chain delegates to next object in chain if it can't handle request
        - Advantages:
            - Decouples a request from all of the thing might handle it
            - Allows the chain of things called to be changed at run time
            - Need a catch all if going to be certain command is handled

        - Disadvantages:
            - Harder to debug

    - **Interpreter Design Pattern**

        - **Provides a way to evaluate language grammar or expression, involves implementing an expression interface which tells to interpret a particular context.**
        - Have the nodes in this tree be instances of classes. Have an interpret function in each node that computes the actions to be performed by the command
        - Advantages:
            - Easy to implement language from grammar
            - Easily change/extend the language
            - Extend plan using composite design pattern

    - **Observer Design Pattern** (implicit invocation)

        - **In one-to-many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically**
        - Invokes all registered objects on the interface when relevant events occur
        - Advantages
            - Decouples observation of an action from consequences

        - New consequences can be added without impacting the Observable code
        - Observable code not cluttered up with all the code concerning every consequence of an action
        - Code ends up where it belongs
        - Complexity is replaced by simplicity

- **Iterator Design Pattern**

    - **Provide a way to access the elements of a collection object in sequential manner without any need to know its underlying representation**
    - Problems:
        - Adds a potentially needless layer of complexity
        - Has to carry where it is
        - User need to know how to use the interator

    - Composite Iterators
        - Have to remember where we are
        - Easy to directly traverse a composite tree structure
        - Harder to write an iterator

- **Mediator Design Pattern**

    - **Reduce communication complexity between multiple objects or classes. Provides a mediator class which handles all the communications between different classes and supports easy maintenance of the code by loose coupling.**
    - Everything reports state change to mediator
    - Everything responds to request from mediator
    - Advantages:
        - All communication between objects is described in one central object
        - Objects do not need to know all other objects talk with
        - Reduce exponential explosion in how things communicate
        - Easier to test
        - Easier to change

- **Memento Design Pattern**

    - **Used to restore state of an object to a previous state. Save this state to a new memento object**

        - Memento: contains state of an object to be restored.
        - Originator: creates and stores states in Memento objects.

        - Caretaker: is responsible to restore object state from Memento.

    - Advantages:
        - Compartmentalizes backup/restore
        - Can be implemented using serialization
        - Can have many saved check points
        - Can restore to any state

Operational

- **Operational Analysis**

    - Basic ABC of quantities
        - A: Total number of Arrivals
        - B: Total number of Busy
        - C: Total number of Completions
        - T: Total time spent monitoring above

    - Basic derived quantities
        - Arrival rate λ: A/T (arrivals/second)
        - Departure rate X (exit rate): C/T (completions/second)
        - Server utilization U: B/T (fraction)
        - Mean service time S per task: B/C (seccond/completion)

    - Utilization law:
        - Utilzation = Completion Rate * Service Time
        - U = B/T = (C/T) * (B/C) = XS

    - Job flow balance assumption
        - Total arrivals = Total Completions
        - Reasonable since (A-C)/C -> 0 as T -> 0
        - λ = X
        - U = λS (steady state limit theorem)

    - Generalizing to network
        - n >= 1 services (servers)
        - Arrival λi = Ai/T (at server i)
        - Departure rate Xi = Ci/T (from server i)
        - Server utilization Ui = Bi/T
        - Mean service time Si = Bi/Ci

- Routing frequency (Cik task goes i -> k)
  - qik = Cik/Ci if i = 1..n

- Queuing at device
  - Wi = ∑t=0..n queue lenth(t)
  - Qi = Wi/T (Average queue length)
  - Ri = Wi/Ci (mean waiting time at i)
  - Qi = XiRi (Little's law)

- Visit ratios
  - Assume Xi = ∑k=0..n XkQki
  - Vi = Xi / Xo = Ci / Co
  - Xi = ViXo

- Interactive Response Time
  - M users using terminal
  - Mean wait time per user at terminal R
  - Mean think time per user Z
  - Mean thinking and waiting time (R+Z)
  - (R+Z) Xo = M = mean number of users
  - R = (M/Xo - Z)
  - R = (∑i=1..n Qi)/Xo

- Markov models:
  - System views as a set of states S0,...Sn, S-1
  - S0 start state, and S-1 final state
  - Probability Pik transition between Si and Sk
  - Probabilities constant for lifetime of model
  - ∑i=-1..n Pik = 1
  - P-1-1 = 1 (Once finished keep finishing)
  - Goals:
    - Determine probability of entering Si
    - Determine expected number of transitions before arriving in State Si
    - Flag certain state transition as signficant
    - Counting achieved by multiplying state transition probability by dummy variables

  - Markov modelling reduction
    - For all i,k reduce multiple transition Si->Sk
    - Select Si that does not move directly to Si
    - Eliminate Si from the model using:

            - For all h,k: Sh->Si->Sk add Sh->Sk with Phi*Pik
        - When all states loop eliminate Pii if i!= -1
            - For all k!=i Pik = Pik*1/(1-G) where G generating function describing Pii
        - Terminate when have G for S0->S-1
    - Equisions – Utilization law: Ui = Xi Si { Ui = λi Si if assume λi = Xi }
        - Little's law –Qi =Xi Ri
        - Forced flow law (Very useful) –Xi=Vi Xo
        - General response time law: R = ∑ i=1..n Vi Ri
        - Interactive response time law (Very useful): R = (M/X0) - Z

DevOps

- Goal of DevOps
    - On the rise
    - Reduce time to market for new features
    - Impack
        - Team organization
        - How systems are built
        - Structure of system
- DevOp Side Effects
    - Enhanced customer experience
        - responsive to customer feedback
    - Improved capacity to innovate
        - Efficient test strategies
        - Roll out better software
        - Recall worse software
    - Faster time to value
        - Tools and culture
- Core Principles
    - test against production-like systems
    - Deploy with repeatable, reliable processes
    - Monitor and validate operational quantities
    - Amplify feedback loops

- The DevOps Culture
  - High degree of collaboration across roles
  - Focus on business
  - Trust and reciprocal support
  - Value placed on learning through experiment
  - Lean agile transformation practices

- The challenge
  - Development: Rewarded for improvements
  - Operations: Rewarded for system uptime and stability
  - Solution: share both reponsibilities
  - Improve visibility of tasks/actions/problems

- Cloud Service Models
  - **Software as a Service (SaaS)**
    - Email, online games, customer relations ...
  - **Platform as a Service (PaaS)**
    - Web services, database, dev tools ...
  - **Infrastructure as a Service (IaaS)**
    - Virtual machines, storage, load balances, networks ...
  - Hybrid Cloud Problems
    - Combination of cloud and physical systems
    - On and off premises cloud systems
    - Combination of IaaS and PaaS
    - Portability across more than one cloud provider

- 10 Myths about DevOps
  - Is only for “born on the web” shops
  - Is operations learning how to code
  - Is just for development and operations
  - Isn’t for ITIL (standards compliance)
  - Isn’t for regulated industries
  - Isn’t for out sourced development
  - No cloud means no DevOps
  - Not for large complex systems
  - Is only about communications

- Involves continuous change deployment