Topic 1: Introduction to Software Engineering

Software and Software Engineering

- A program or sequence of instructions that tell the computer what tasks it needs to perform and how to perform them
- Can be broken down into:
    1. Application software
    2. System software

Software Development Project

- A project consists of a team of:
    - Business analysts
    - Software:
        - Project manager
        - Designers
        - Developers
        - Testers
- Software product (collective set of entities that includes the program, documentation, data, etc) needs to comply with:
    - Security
    - Performance
    - Reliability

Software Engineering

- The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software
- Software engineer takes a problem definition and applies tools to obtain a problem solution, following specific methodologies for managing the development process
    - Their focus is on understanding the interaction between the system, its users, and the environment, and designing the software based on this
    - The process involves:
        - Understanding the business problem definition
        - Creative formulation of ideas to solve the problem and designing the ‘blueprint’ or architecture of the solution
        - The ‘blueprint’ is then implemented using the programming craft
- Fundamental Law:
    - Software engineering is understanding the problem domain by learning the new environment and its disciplines, communicating with domain experts to extract an abstract model for the problem solution that makes sense in the context of the customer’s business, and designing a software system that will realise the proposed solution and evolve with the business needs in the future
- Difficulties:
    - Software domain - understand what you are building
    - Problem domain - understand the problem and solution
    - Formal domain - programming is formal with fixed inputs and goals, but real world is informal
    - Abstractions - good abstractions wear out, break and get dispersed due to evolving environments

Software Engineering Life-Cycle

The organised process of software engineering can be broken down into:

1. Analysis and Specification
2. Design
3. Implementation
4. Testing
5. Release and Maintenance

- There are also peripheral activities such as feasibility studies, software maintenance, software configuration management, etc.
- Each phase can be accompanied by an artifact or deliverable to be achieved at the completion of the phase

Analysis and Specification

Process of knowledge-discovery about the possible solution and list of features by:

1. Understand problem definition
    a. Delimit its scope
    b. Elaborate system services (behavioural characteristics)
2. Understand requirements
    a. Functional (input/output)
    b. Non-functional requirements (performance, security, quality, maintainability, extensibility)

Techniques include:

1. Use-case modelling
2. User stories

Design

Problem-solving activity that involves a creative process of searching how to implement all requirements to generate engineering blueprints using techniques such as:

1. Entity-Relationship Models
2. UML diagrams (class diagram, component diagram, deployment diagram)

Implementation

The process of encoding the design in a programming language to deliver a software product

Testing

A process of verifying system correctness and usability, and that it realises the intended goals by:

1. Unit tests (individual components are tested)
2. Integration tests (whole system is tested)
3. User acceptance tests (customer requirements met test)

Operation and Maintenance

Running the system, fixing defects, adding new functionality

Software Engineering Design

Due to the intangible nature and complexity of software development it is best to understand and implement a solution by evaluating pilot solutions, which led to the adoption of iterative development

Incremental and Iterative Development Methods

1. Break the problem down into smaller parts and prioritise them
2. Each iteration should progress the development in more depth
3. Seek customer feedback and change course based on further understanding

The goal of an incremental and iterative process is to:

- Get to a working instance as soon as possible
- Progressively deepen the understanding or visualisation of the target product

Topic 2: Software Development Methods

A software development method lays out a prescriptive process by mandating a sequence of development tasks. There are two main types of processes:

1. Elaborate processes (rigid, plan-driven, documentation heavy methodologies)
    a. Waterfall: unidirectional (finish this step before moving to the next)
2. Iterative and incremental processes (develop increments of functionality and repeat in a feedback loop)
    a. Rational Unified Process
    b. Agile Methods (SCRUM, XP)
        i. Aggressive and short iterations
        ii. Heavy customer involvement, rely on user feedback for prioritisation and review of delivered iterations

Waterfall Model

Developed in the 70s, it is a traditional, linear, sequential life cycle model with detailed planning (it is also known as the plan-driven development model)

- Detailed planning - problem identified, documented and designed
- Implementation tasks identified, scoped and scheduled
- Development cycle is followed by a testing cycle

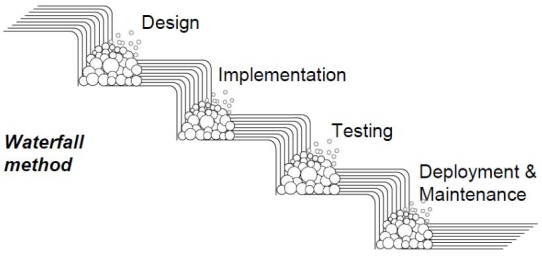
Advantages and Use Cases

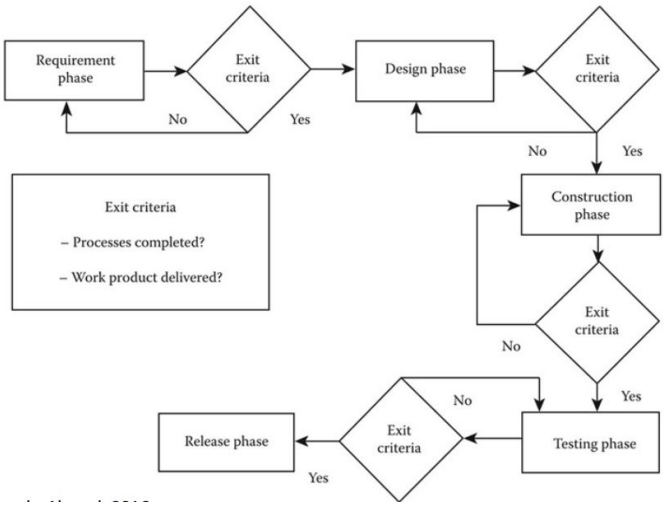
Best suited for:

- Simple, risk-free projects with stable product statement
- Clear, well-known requirements with no ambiguities
- Clear technical requirements and ample resources
- Mission-critical applications (think NASA)

Project visibility

- Makes the design simple to understand and manage because of clear visibility over all the phases in the project





Waterfall Model ft. Quality Gate System

A variation of the waterfall model that ensures that quality is maintained throughout the life-cycle

- A completion criteria check is done that does a quality assurance check to see if all the necessary artifacts are generated for that phase and the artifacts meet the quality standards

Drawbacks

Disadvantages:

- No working software produced until late into the software life-cycle
- Inflexible
    - Does not support fine-tuning or refinement
    - Ideas must be identified upfront
    - Typically all requirements are frozen at the end of the requirements phase, making it difficult to retract or change something that was not well thought out in the concept or design phase
- Heavy documentation is required
- Not suitable for projects where requirements are at a moderate risk of changing
- Typically incurs a large management overhead

Rational Unified Process (RUP)

An iterative software development process with four phases:

1. Inception
    a. Scope the project, identify major players, resources required, architecture and risks, and estimate costs
2. Elaboration
    a. Understand the problem domain, analysis, evaluate in detail required architecture and resources
3. Construction
    a. Design, build and test software
4. Transition
    a. Release software to production

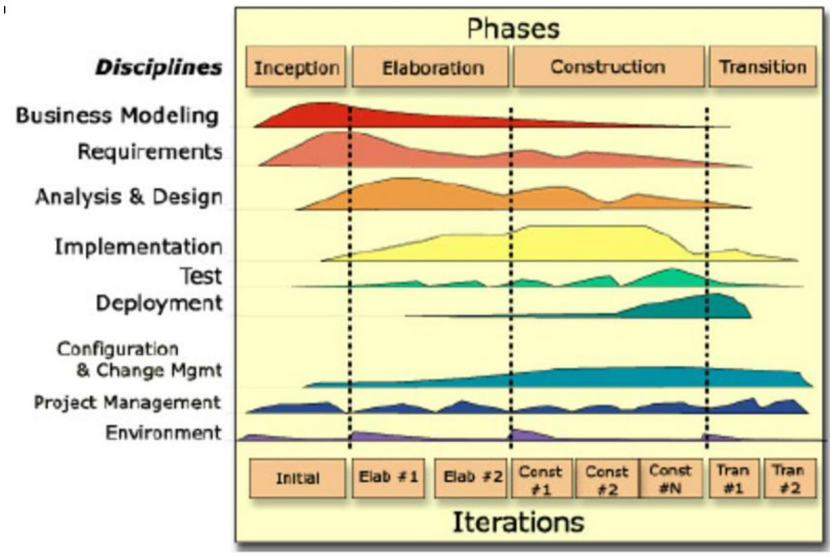
RUP is serial in the large and iterative in the small

- The four phases occur in a serial manner over time, however, work in an iterative manner on a **day-to-day basis**

RUP Disciplines

All work in RUP is organised into disciplines:

1. Development disciplines
    a. Business modelling (understand domain and develop high level requirement model), develop:
        i. Use-case model
    b. Requirements (identify, model and document vision and requirements), develop:
        i. Use-case model & domain model (class or data diagram)
        ii. Business process model (data flow diagram & activity diagram)
    c. Analysis and design (engineer the blueprint)
    d. Implementation (encode the design)
    e. Test (testing throughout the project)
    f. Deployment (product releases, software delivery)
2. Support disciplines
    a. Configuration and change management
    b. Project management
    c. Environment



Agile Methodology

Agile Manifesto:

1. Individual and interactions > processes and tools
2. Working software > comprehensive documentation
3. Customer collaboration > contract negotiation
4. Responding to change > following a plan

Agile Principles

1. Highest priority → satisfy the customer through early and continuous delivery of valuable software
2. Welcoming of changing requirements at all stages of development
3. Working software is delivered frequently
4. Business people and developers work together daily
5. Build projects around motivated individuals → trust them to get the job done
6. Face-to-face communication is the most efficient and effective method of communicating information
7. Working software is the primary measure of progress
8. Agile processes promote sustainable development and constant working pace
9. Continuous attention to technical excellence and good design enhances agility
10. Simplicity - maximising the amount of work not done - is essential
11. The best architecture, requirements and designs emerge from self-organising teams
12. Regularly, the team should reflect on how to become more effective, and responds to this accordingly

Agile Drawbacks

1. Difficulties for clients:
    a. Frequent in-person communications can be difficult in that:
        i. Clients and developers are separated geographically
        ii. Clients do not have the interest or manpower to invest in the process
    b. Estimates and timetables are adaptive and are not rigid
        i. Clients may prefer firm answers
2. Difficulties for developers:
    a. Small self-organised teams can struggle with:
        i. Adaptation to large software projects with many stakeholders
        ii. Finding leadership
    b. Lack of comprehensive documentation can make it hard to:
        i. Maintain or add to the software for new developers leading to inconsistent features and interfaces
3. Difficulties for project managers:
    a. Agile development depends on the ability to recruit very experienced engineers who can work independently and interface effectively with business users

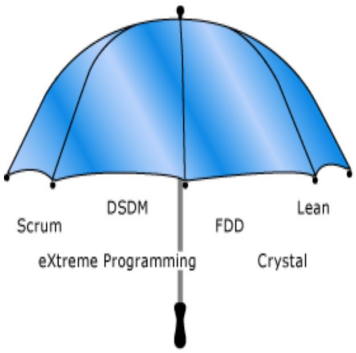
Which Methodology?

The customer wants the **newest software today**, with the most advanced features, **at the lowest possible cost**

There is no best methodology - consider all three and what the business requirement is - but be agile in the approach to re-evaluation and reflection

Topic 3: XP and Scrum

The Agile Umbrella



Extreme Programming (XP)

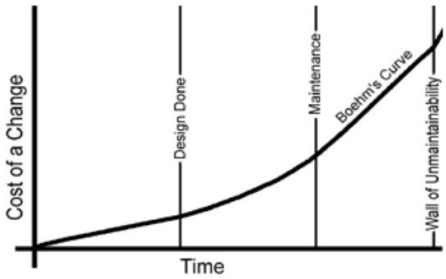
A prominent agile software engineering methodology that:

1. Focuses on providing the highest value for the customer in the fastest possible way
2. Acknowledges changes to requirements as natural and inescapable
3. Values adaptability over predictability
4. Aims to lower the cost of change by introducing a set of basic values, principles and practices allowing more flexibility to changes

XP Values

Boehm's Cost Curve

- As software proceeds through its lifecycle, the cost of making a change becomes larger



Values of XP

1. Communication
    a. Increased collab between customers and developers as members of a team
2. Simplicity
    a. Focus on design and coding on the simplest solution for today's needs
3. Feedback
    a. Via the customer and team, or testing (unit and integration)
4. Courage
    a. Design and code for today, be comfortable refactoring code, be truthful about progress and estimates, don't be fearful, take on problems proactively
5. Respect
    a. Never commit changes that break compilation or fail unit-tests, strive for high-quality

XP Principles

<u>Whole Team</u>

- Managers, developers and customers work closely as a single team, collaborating to solve communal problems
- Customer is a part of the team - responsible for defining and prioritising features
    - Proximity to the development team makes it easier to facilitate this
    - Customer representatives may step in to fill the geographical gap

User Stories

- A user story is:
    - A mnemonic token of an ongoing conversation about a requirement and is essential to the planning of XP
    - Developers and customers discuss the requirements of the software and build user-stories, a short simple description of a feature narrated from the perspective of the person who desires that capability
        - As a <type of user>, I want <some goal> so that <some reason>
    - Often written physically on index cards and arranged on walls which serves as a planning tool to schedule the implementation of a requirement, based on its priority and estimated cost

Short Cycles

- Using the user stories an XP team creates a:
    - Project Release Plan
        - A release is usually 3 months' work and represents a major delivery that can be deployed to production and often maps out the next six or so iterations
        - Process:
            - Customer selects stories for the release (subject to total estimate not exceeding developer budget)
            - Customer determines the order for completing the stories
            - The stories are broken down into tasks for developers
    - Iteration plan
        - XP projects deliver working software every two weeks
        - Iteration plan represents a minor delivery of some user stories
        - Can not change the definition or priority of the user story once an iteration has been started
        - At the end of the two weeks, the system is demonstrated to the stakeholders
- Plans are considered temporary artifacts in XP
    - Every time the customer gains insight, or if the team falls behind or moves ahead of the current schedule, then a new plan will be made

High Quality

- Pair programming
    - Two programmers work together, one coding, the other reviewing. Roles change frequently and anyone can check out any module and improve it
- Continuous integration
    - Programmers check their code in and integrate several times per day (merge if not first to check in)
- Sustainable pace
- Open workspace
- Refactoring
    - New features, requirement changes, bug fixes (software rot) is avoided by refactoring - making a series of tiny transformations to improve the structure of the system without affecting the behaviour

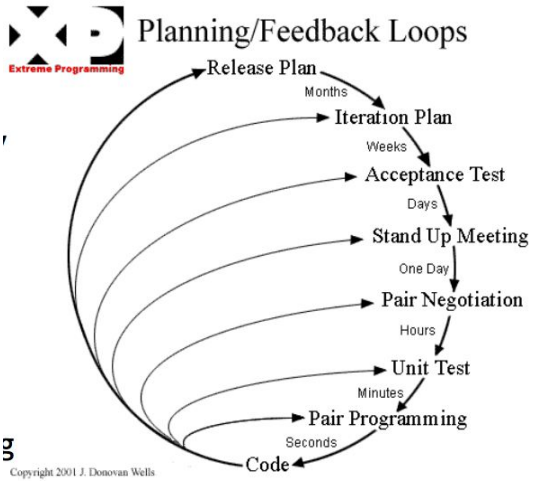
Test-driven development (TDD)

- Unit Testing
    - A complete body of test-cases evolve along with the code
    - Unit tests encourage developers to "decouple" modules, so that they can be tested independently
- User Acceptance Testing
    - Written by customers/analysts, they are the "true" requirements document of the project
    - Serve as final authority to determine correctness of implementation
    - Once an acceptance test passes, it is added to a body of passing acceptance tests, and is not allowed to fail again. This is achieved by running acceptance test body and if a new build fails these tests, the build is considered a failure

Simple Design

- Focus on the stories in the current iteration
- Design mantras:
    - Consider simplest possible design, resist adding unnecessary detail, don't duplicate code

Continuous Feedback

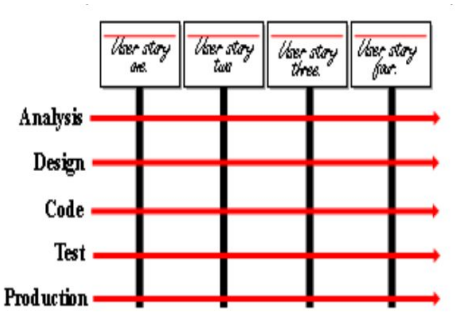


System Metaphor

- A story that everyone - customers, programmers and managers - can tell about how the system works
- It is necessary for:
    - A common vision (understanding the system)
    - Shared vocabulary (define the system)
    - Architecture (shaping of the system)
    - Generativity (new ideas about the system)

Life-Cycle

XP flips the linear lifecycle of software development on its axis: from Analysis → Design → Code → Test → Production for the overall system, to:



XP Planning

1. Initial exploration
    a. Developer/customer have conversations about the system-to-be and identify significant features
    b. Features broken down into user stories
    c. Developers estimate the user story in user story points
        i. Avoid stories that are too large or too big
        ii. Velocity - the sum of the estimates of the completed stories each week
        iii. Average velocity - the average of the weekly velocities, a useful tracking and planning metric for future work
2. Release plan
    a. Created in the release planning meeting, a release date in which customers specify which user stories are needed and the order for the planned date
    b. Use the project velocity to plan by time or scope (how long a set of stories will take to finish)
    c. Four major ideas to consider
        i. Scope - how much is to be done
        ii. Release - how many people are available
        iii. Time - when the release is due
        iv. Quality - how good the software will be

3. Iteration planning
    a. Created using the release plan
    b. Developers and customers choose an iteration size (1 or 2 weeks)
    c. Customers choose the user stories for the iteration if they fit with the current velocity, but the order of implementation within each iteration is a technical decision
    d. The iteration ends on the specified date, at which point estimates for the iterations velocity are made, even if certain stories aren't done
        i. "Done" means all its acceptance tests pass
4. Task planning
    a. Developers and customers arrange an iteration planning meeting at the beginning of each iteration
    b. From the customers' selected user stories, developers break them down into programming tasks, and select an order for implementation of programming tasks
    c. Tasks should be within .5, 1, 2 or 3 days of ideal programming days, and if it exceeds the project velocity, then user stories will be removed, or if below the project velocity, user stories added
    d. Iteration planning velocity overrides the release planning velocity because it is more accurate

Planning

- User stories are written.
- Release planning creates the release schedule.
- Make frequent small releases.
- The project is divided into iterations.
- Iteration planning starts each iteration.

Managing

- Give the team a dedicated open work space.
- Set a sustainable pace.
- A stand up meeting starts each day.
- The Project Velocity is measured.
- Move people around.

eaks.

Designing

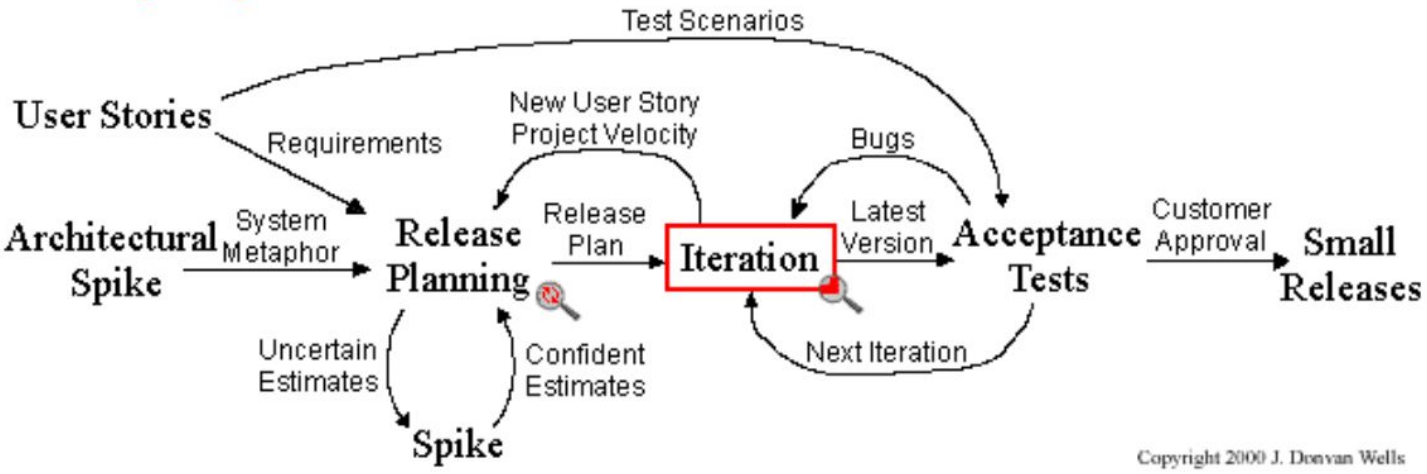
- Simplicity.
- Choose a system metaphor.
- Use CRC cards for design sessions.
- Create spike solutions to reduce risk.
- No functionality is added early.
- Refactor whenever and wherever possible.

Coding

- The customer is always available.
- Code must be written to agreed standards.
- Code the unit test first.
- All production code is pair programmed.
- Only one pair integrates code at a time.
- Integrate often.
- Set up a dedicated integration computer.
- Use collective ownership.

Testing

- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often and the score is published.



Project Tracking

Recording the results of each iteration and using those results to predict what will happen in the next iteration. A variety of charts are used to record this:

1. Velocity Chart
    a. Tracks how many story points (y-axis) were completed in each iteration (x-axis)
    b. Story points are stories that passed the user acceptance tests
    c. An average project velocity can be deduced from this
2. Burn-down Chart
    a. Tracks story points remaining (y-axis) over the project timeline (x-axis)
    b. New stories can be added to the project and old stories may be re-estimated and so can increase the y-height between periods
    c. The slope of the chart can be used to predict the end-date

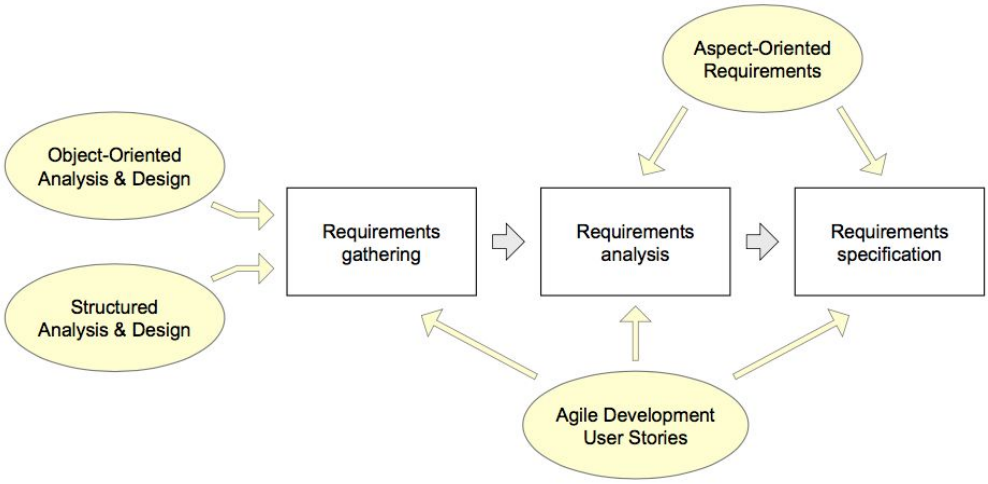
When to use XP?

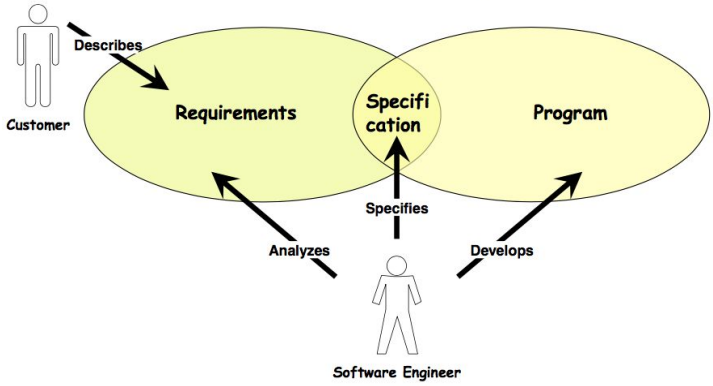
1. When the problems requirements are expected to change
2. When the customer does not have a firm idea of what they want or the system functionality is expected to change
3. When a project is determined to be a risky project - XP was developed as a risk mitigator
4. Suitable for project group sizes of 2-12
5. When a team has managers, developers and customers all collaborating closely

Topic 4: Requirements Engineering and Use-Case Modelling

Requirements Process

1. Gathering
    a. Helps the customer to define what is required:
        i. What needs to be accomplished
        ii. How the system will fit into the needs of the business
        iii. How the system will be used on a day-to-day basis
2. Analysis
    a. Refining and modifying the gathered requirements
3. Specification
    a. Documenting the system requirements in a semiformal or formal manner to ensure clarity, consistency, and completeness





Types of Requirements:

1. Functional
2. Non-functional (quality requirements)
    a. Security, usability, reliability, performance, supportability
3. On-screen appearance
    a. Hand-drawing the proposed interface forces you to economise and focus on the most important features and should only be invested in when a consensus on a design is reached

Acceptance Tests

Acceptance tests provide a means of assessing that the requirements are met as expected

1. Conducted by the customer
2. An acceptance test describes whether the system will pass or fail the test

Project Estimation using User Story Points

Size points assigned to each user story as a means to measure the time involved to complete a user story

1. Total work size estimate = sum of each user-story points
2. Velocity = estimated from number of user-story points completed per iteration
3. Project duration = path size / travel velocity

Use Cases

A use case is a step-by-step description of how a user will use the system-to-be to accomplish business goals. A common structure is the RGB method:

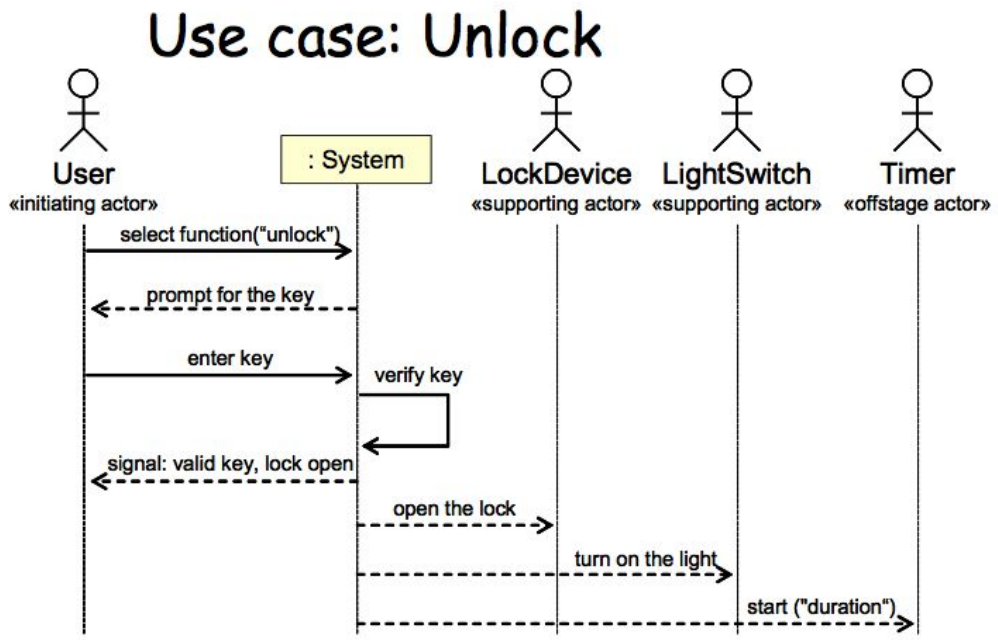
- Role (which user) → Goal (what the user wants to achieve) → Benefits (why the user wants to achieve this)

Actors

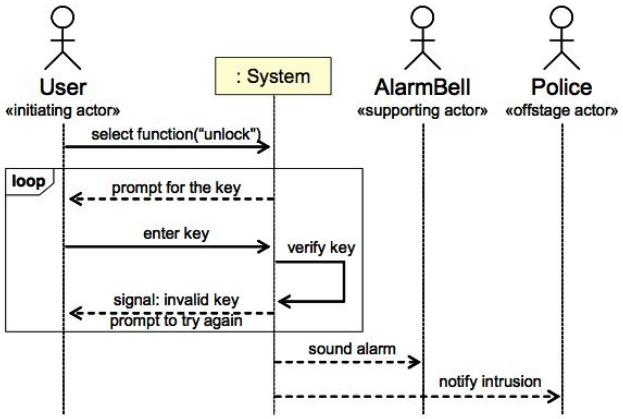
- Initiating actor (primary actor or user)
    - Initiates the use case to achieve a goal
- Participating actor (secondary actor)
    - Participates in the use case but does not initiate it
        - Supporting actor - helps the system-to-be to complete the use case
        - Offstage actor - passively participates in the use case

ID	S1
Name	**User Story Name**
Description	As a ... I would like to ... so that I can ...
Acceptance Criteria	1. Acceptance Criteria #1 2. ...
Priority	Essential/Future/Optional
Size	2 SP

System Sequence Diagram



Main success scenario



Alternate scenario (burglary attempt)

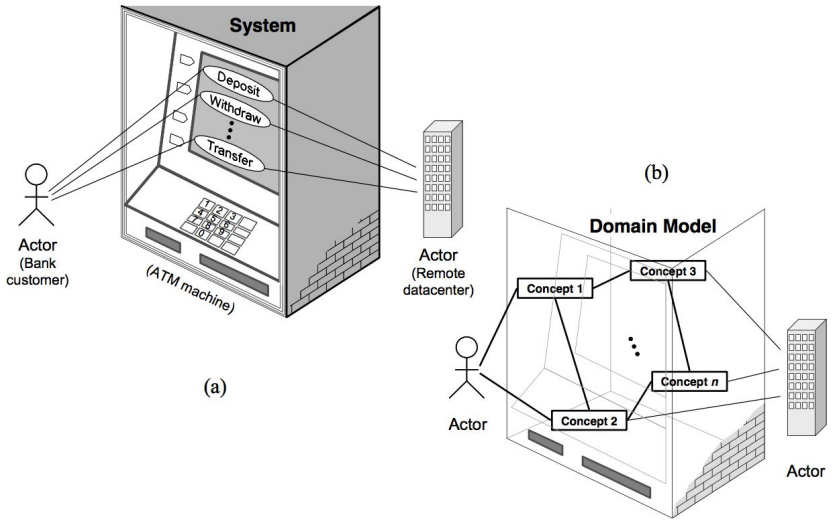
Topic 5: Domain Modelling using Object-Oriented Design Techniques

Domain Model (conceptual model)

The domain model

1. Created to provide a visual representation of the problem domain, by decomposing the domain into key concepts or objects in the real-world and identifying relationships between these objects
2. Goal is to understand how the system-to-be will work
3. Determines how elements of the system-to-be interact (internal behaviour) to produce the external behaviour
4. Is developed and continuously refined

Use-case (a) vs. Domain Model (b)



Domain Modelling Techniques

1. Studying the problem statement
2. Use sources to identify the conceptual classes and their key attributes/responsibilities and determine the collaboration between these classes
    a. Noun and Verb Phrase Identification
        i. Noun: class
        ii. Verb: attribute/method
    b. CRC Cards
        i. Class - represents a collection of similar objects
        ii. Responsibility - something that the class knows or does
        iii. Collaborator - another class must be interacted with to fulfil its responsibilities

Student	
Enrols in a Seminar *Knows* Name *Knows* Address *Knows* Phone Number	Seminar

    c. UML Class Diagram
        i. Evolves from the CRC domain models

Object Oriented Design

Objects

- Real world entities that can be tangible (car, phone) or intangible (account, time)
- Each object has its own attributes and behaviours

Attribute

- Characteristics and properties of the object

Behaviour

- What the object can do (methods or functions)

Object State

- Each object encapsulates some state - the currently assigned values for its attributes

Object Interface and Communication

- The methods on the object through which objects can communicate with each other
- If A calls B
    - A sends a message to B
    - A is the client and B is the server

Class

- A blueprint to describe many objects that share the same semantics, properties and behaviour
- Defines the attributes and methods of the object
- An object is instantiated from a class (an instance of a class)
- An object has state but a class does not

Abstraction

Abstraction says “create one generic class that has the common properties and behaviour of the objects”

- Is at the heart of OO design and key to defining a class
- The object of abstraction is to:
    - Focus on the common essential qualities of the idea
    - Focus on the current application context
    - Write <u>one</u> class utilising these ideas
    - If some specific object of this class has a special property or behaviour, consider inheritance

Encapsulation

- Means hiding the object state so that it can only be observed or modified through the object’s methods
- An object’s attributes represent its individual characteristics or properties, so access to the object’s data must be restricted
- An object’s state should only be observed or modified through the object’s methods

Benefits

- Restricted access means you only have to worry about a single class
- Encapsulation abstracts away the implementation, reduces dependencies between different parts of the application and ensures that a change will not cause a rippling effect on the system

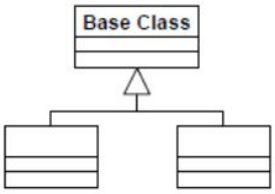
Class is broken up into three parts:

1. Public interface - the services provided by the object
2. Contracts - the terms and conditions of use
3. Implementation - details of how the object conducts its business

Inheritance

Inheritance says “create a new class (subclass) that inherits common properties and behaviour from a base class (superclass)

- Defines a “is-a” type of relationship



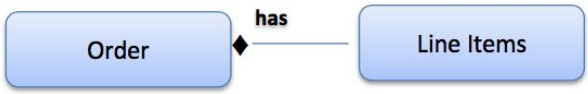
Composition and Aggregation

A special type of relationship where one class “contains” another class

- Defines a “has-a” relationship
- E.g. a course offering has students

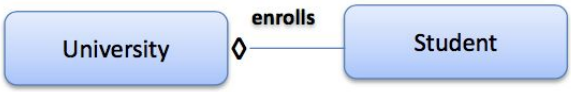
Composition

- The contained item is an integral part of the containing item



Aggregation

- The contained item is an element of a collection but it can also exist on its own



SOLID Principles

SOLID Goals

Make software easy to:

- Understand, maintain, extend, reuse and test
- Avoid software rot

Software Rot

Software rot is bad code arising from:

- Lack of knowledge
- Rushing
- Change of system/context/requirements etc.

Recognising software rot:

- Rigidity - tendency of software to be difficult to change even in simple ways
- Fragility - tendency of program to break in many places when a single change is made
- Immobility - Effort and risk involved in separating parts of a system from the original system are high

Solving software rot:

- Refactoring (changing the code without changing the external functionality)
    - Requires time that may not be available
    - May impact other parts of the system

Avoiding software rot:

- Agile design, OO design and following the SOLID principles

SOLID Principles

SRP - Single Responsibility Principle

A class should have <u>one and only one</u> reason to change

- Every class should have only one responsibility
- Responsibility is defined as a reason for change
- If a class has more than one responsibility, the responsibilities become coupled

SRP does not imply “do only one thing”:

- One function can invoke several other functions, but it should not be responsible for how these functions are implemented

Advantages:

1. Readability - easier to focus/identify one responsibility
2. Reusability - the code can be reused in different contexts
3. Testability - each responsibility can be tested in isolation

OCP - Open Closed Principle

Software entities (classes, modules, functions) should be <u>open for extension but closed for modification</u>

- Open for extension
    - As the requirements change, the behaviour of the class can be extended with new behaviours
- Closed for modification
    - Extending the behaviour of the module does not result in changes to the source

LSP - Liskov Substitution Principle

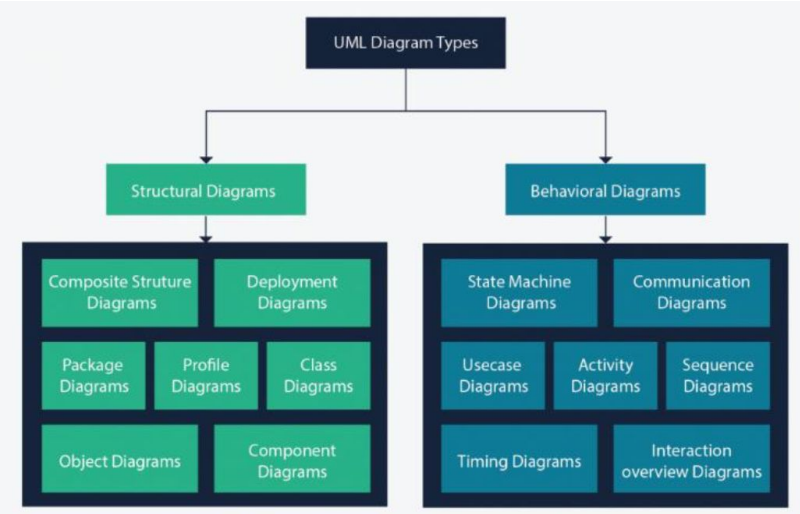
ISP - Interface Segregation Principle

DIP - Dependency Injection Principle

UML (Unified Modelling Language)

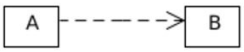
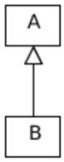
An open source, graphical language to model software solutions, application structures, system behaviour and business processes. Used for:

- Communicating aspects of the system
- A software blueprint



Class Diagram

Using CRC technique to identify classes and their attributes, responsibilities and collaborators

Relationship	Depiction	Interpretation
Dependency		A depends on B This is a very loose relationship and so I rarely use it, but it's good to recognize and be able to read it.
Association		An A sends messages to a B Associations imply a direct communication path. In programming terms, it means instances of A can call methods of instances of B, for example, if a B is passed to a method of an A.
Aggregation		An A is made up of B This is a part-to-whole relationship, where A is the whole and B is the part. In code, this essentially implies A has fields of type B.
Composition		An A is made up of B with lifetime dependency That is, A aggregates B, and if the A is destroyed, its B are destroyed as well.
Generalization		A generalizes B Equivalently, B is a subclass of A. In Java, this is `extends`.
Realization		B realizes (the interface defined in) A As the parenthetical name implies, this is used to show that a class realizes an interface. In Java, this is `implements`, and so it would be common for A to have the «interface» stereotype.

Each class can be broken down by it's class name, attributes and methods:

- Visibility can be public (+) or private (-)

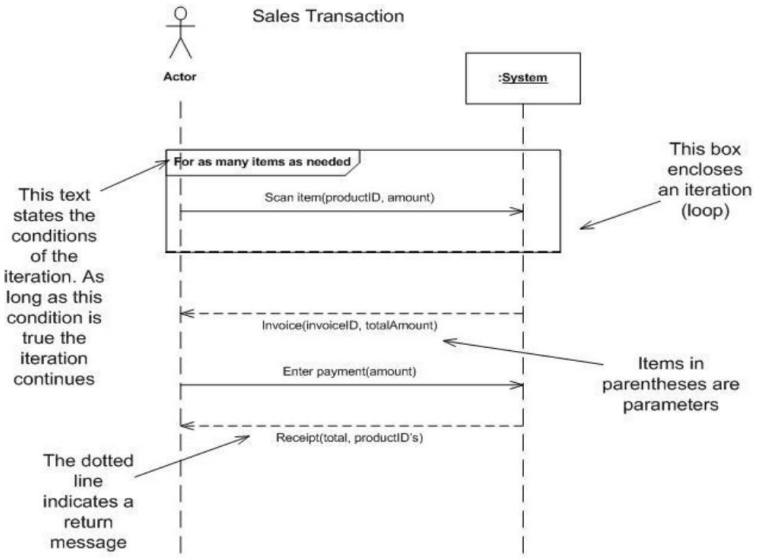
Classname
- attribute_name: data_type - ...
+ method_name(parameters): return_type - ...

Sequence Diagram

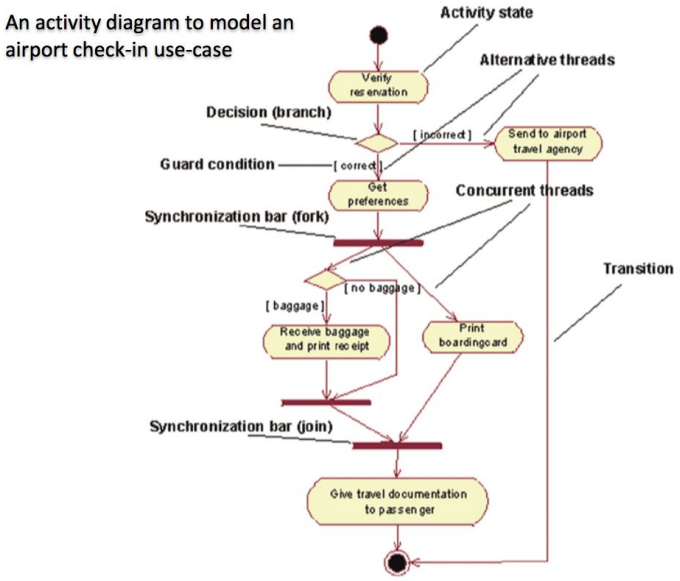
A UML behavioural diagram, providing a visual summary of a use-case scenario

- Should be done for the main success scenario of the use case, and frequent or complex alternative scenarios
- Specifies:
  - External actors
  - Methods invoked by these actors
  - Return values (if any) associated
  - Indication of any loops

a) Sequence Diagram



b) Activity Diagram



Activity Diagram

Useful to visualise the workflow of a business use-case

<u>Activity vs Sequence</u>

- Complementary techniques
  - Activity diagram → workflow and partitioning of class responsibilities
  - Sequence diagram → business entity handling and how objects interact

Software Architecture

Software architecture:

- A set of high-level decisions that determine the structure of the solution
- The “structure” of the system, which comprises software elements, the properties of these elements, and the relationships among them
- Requires partitioning the system into logical subsystems or parts
    - Tackles complexity by ‘dividing and conquering’
    - Avoid reinventing the wheel
    - Support flexibility and future evolution by decoupling unrelated parts
- Key system properties:
    - Security, reliability, performance
    - Reusability, extensibility, maintainability
    - Coupling, cohesiveness
    - Usability, compatibility
    - Cost

Architectural Decisions

Concerned with the non-functional requirements and decomposition of functional requirements

1. How to decompose the system into parts?
2. How the parts relate to one another?
3. How to document the system’s software architecture?

Architectural Styles

Defined by:

1. Components
    a. Processing elements that do the work (e.g. classes, databases, tools, etc)
2. Connectors
    a. Enable communication among different components (e.g. function call, event broadcasts etc.)
3. Constraints
    a. Define how the components can be combined to form the system (where data may flow in and out of the components/connectors)
    b. Topological definition of the arrangement of the components and connectors

Central Repository (database)

Appropriate for a complex body of knowledge, that needs to be accessed and manipulated in several ways (e.g. applications: graphical editors, database applications, AI knowledge bases)

- Components
    - A permanent data structure that represents the state of the system
    - Data accessors - a collection of independent computational elements that operate on the central data
- Connectors
    - Read/write mechanism

Specialisations:

- Blackboard Architecture - an accessor component changes data on the repo, all other components notified
- Passive Data Repository - components access repo at will

Advantages:

1. Efficient way to share large amounts of data
2. Centralised management of the repository
    a. Concurrency access and data integrity, security, backup

Weaknesses:

1. Components must agree upon a repository data model
2. Distribution of data can be problematic
3. Connectors implement complex infrastructure

Pipe-and-Filter

A design that is suitable for processing data streams, and transform into output data streams (e.g. UNIX shell commands, compilers)

- Components - Filters transform input into output
- Connectors - Pipe data streams

Advantages

1. Easy to understand, decouples different steps, supports reuse, flexible and easily maintained

Disadvantages

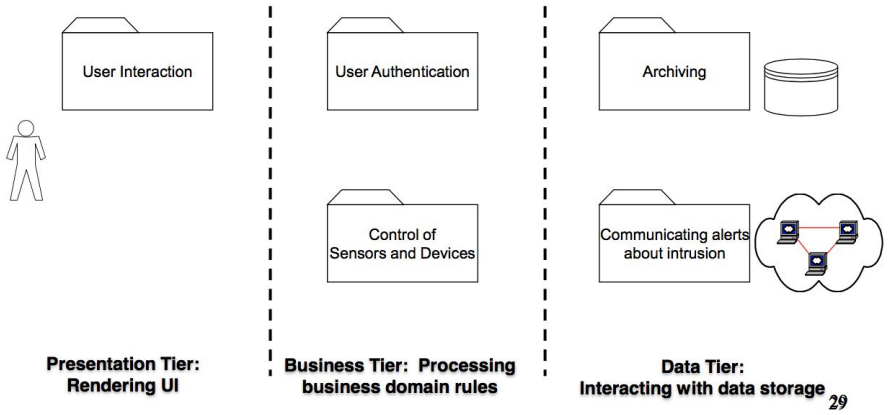
1. If an intermediate filter crashes, challenges arise

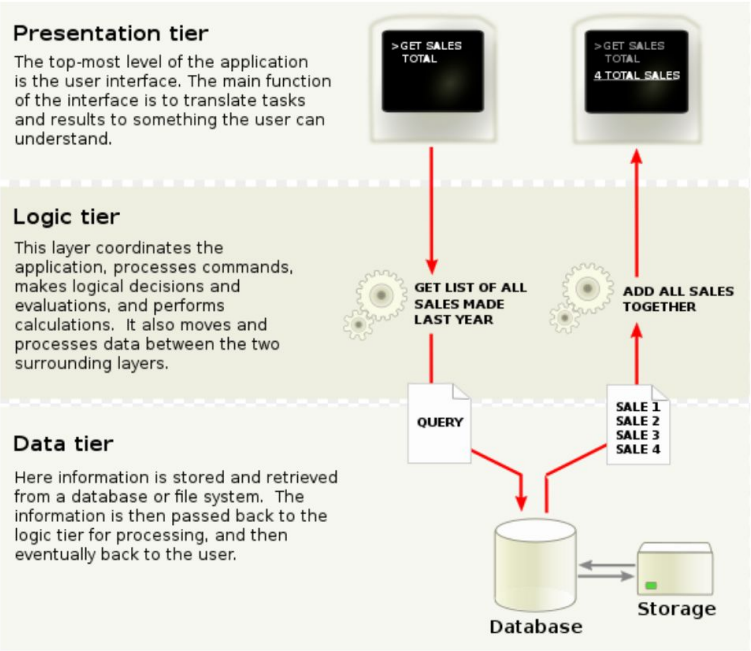
Client/Server

A design suitable for sharing data between a client and a service provider geographically separated (e.g. file server, database server, email server)

- Components
  - A server that provides specific services (database or file server)
  - A client component that requests these services
- Connectors
  - Based on a request-response model

3-Tier Architecture





Advantages:

1. Easier maintenance and reuse, increased security, scalability and performance

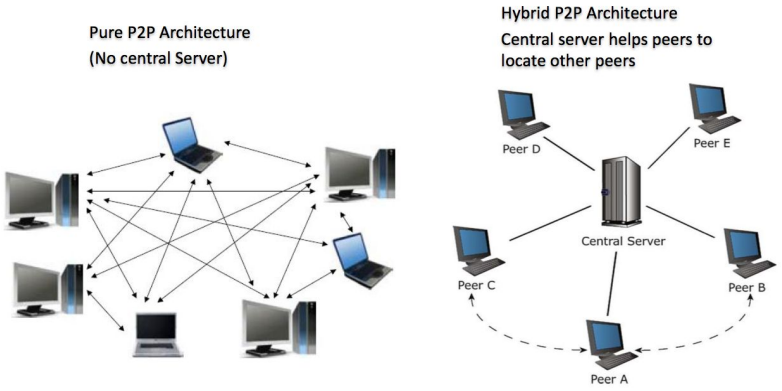
Disadvantages:

1. Complex and expensive infrastructure

Peer-to-Peer

A design suitable for resolving network congestion and single point of failure (e.g. Bit Torrent)
Architecture:

- Each peer component can function as both a server and a client
- Information distributed among all peers



Advantages:

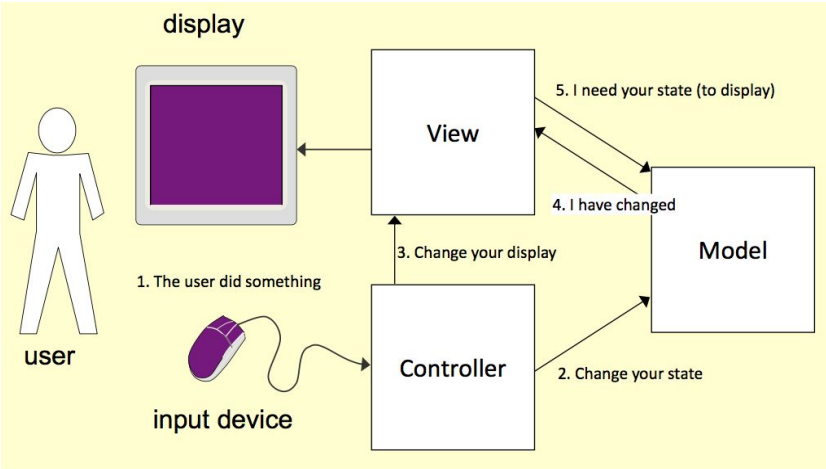
1. Efficient as all clients provide resources
2. Capacity of the network increases with number of clients
3. Robust as it is immune to a single point of failure

Weaknesses:

1. Architectural complexity

Model-View-Controller

Decouples data access, application logic and user interface into three distinct components



Model

- Holds all the data, state and application logic
- Responds to instructions to change state from controller
- Responds to requests for information about its state from the view

View

- Gets data directly from the Model and manages display of information

Controller

- Takes user input and informs the view or model to change as appropriate

Advantages:

1. Accommodates change
2. Supports multiple views of the same data on different platforms at the same time
3. Enhances testability

Weaknesses:

1. Complexity
2. Cost of frequent update

Databases

Key Terms

Database

A database is a logically coherent collection of related data (facts that can be recorded and have implicit meaning). Allows data to be stored, manipulated and shared

DBMS

A database management system (DBMS) is an application that allows users to:

- Create and maintain a database (DDL)
- Define queries that retrieves data
- Perform transactions to write or delete data from the database (DML)

RDBMS

Relational Database Management System

- Stores data as tuples or records in tables
- Allows the user to create relationships between tables

Data Model

- Describes how the data is structured in the database
- There are several types of data models:
    - Relational model
        - Data is stored as a set of records known as tables
            - Row = tuple
            - Column = field
    - Document model
        - Data is stored in a hierarchical fashion
    - Object-oriented model
        - Data is stored as a collection of objects
    - Object-relational model
        - Combined the relational and OO database models

Database Schema

- Adheres to the data model and provides a logical view of the database, defining how the data is organised and the relationships between them

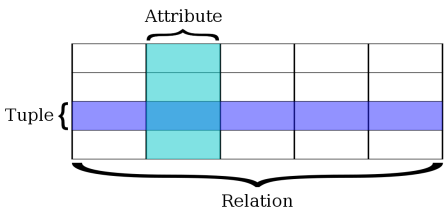
Database Instance

- The state of the database at a particular instance in time

Relational Data Model

Describes the world as a collection of interconnected relations (or tables)

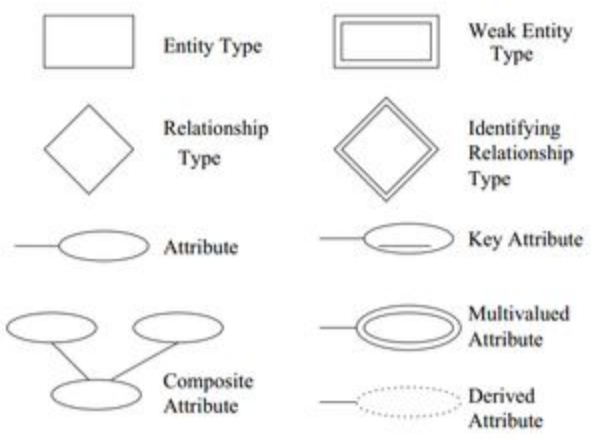
1. Each relation (table) has:
    a. A (unique) name
    b. A set of attributes (or column headings)
    c. A (unique) key - a subset of attributes
2. Each attribute has:
    a. A (unique) name
    b. An associated domain (set of allowed values)
3. Attribute values are:
    a. Atomic (no composite or multi-valued attributes)
    b. Belong to a domain (which has a name, data type and format)



Given a relation R which has n attributes (a1, a2, ... an) with corresponding domains (d1, d2, … dn)

- Relation schema of R is R(a1:d1, a2:d2, … an:dn)
- Tuple of R is is a list of values for a particular row
- Instance of R is a set of values for a range of tuples

Entity Relationship Diagram



SQL Syntax

Query syntax is:

- SELECT attributes
- FROM relations
- WHERE condition

Data Types:

- Int
- Float
- Char(n)
- Varchar(n)
- Date

SQL provides a DDL - Data Definition Language for creating relations:

```
CREATE TABLE TableName (
attrName1 domain1 constraints1 ,
attrName2 domain2 constraints2 , ...
PRIMARY KEY (attri,attrj,...)
FOREIGN KEY (attrx,attry,...)
REFERENCES OtherTable (attrm,attrn,...)
);
```

SEE COMP3311 NOTES FOR MORE INFO ON DATABASES

ER to Relational Data Model Mapping

One technique for DB design is to first design a conceptual schema using a high-level data model, and then map it to a conceptual schema in the DBMS data model for the chosen DBMS

For ER → RD Model there are 7 steps:

1. Entities
    a. For each regular (not weak) entity type E, create a relation R with:
        i. Attributes (all simple attributes, and *, of E)
        ii. Key (choose one of the keys of E as the primary key for the relation
    b. For each specialised entity type E, with parent entity type P, create a relation R with:
        i. Attributes (the attributes of the key P, plus the simple attributes of E)
        ii. Key (the key of P)
2. Weak Entities
    a. For each weak entity type W, with owner entity type E, create a relation R with:
        i. Attributes (all simple attributes of W, and *, include as a FK the prime attributes of the relation derived from E
        ii. Key (the foreign key plus the partial key of W)
3. 1:1 Relationships
    a. For each 1:1 relationship type B, let E and F be the participating entity types, let S and T be the corresponding relations
        i. Choose one of S and T (prefer one that participates totally), say S
        ii. Add the attributes of the primary key of T to S as a FK
        iii. Add the simple attributes (*) of B as attributes of S
4. 1:N Relationships
    a. For each 1:N relationship type B, let E (1) and F (N) be the participating entity types, let S and T be the corresponding relations
        i. Add the attributes of the primary key of S to T as a foreign key
        ii. Add to T any simple attributes (*) of the relationship
5. N:M Relationships
    a. For each N:M relationship type B, create a new relation R. Let E and F be the participating entity types, let S and T be the corresponding relations
        i. Attributes (the key of S and the key of T as FK, plus the simple attributes of B)
        ii. Key (the key of S and the key of T)
6. Multivalued attribute A
    a. For each multivalued attribute A, create a new relation R. Let A be an attribute of E
        i. Attributes (1: simple attribute), 2: composite attribute)
            1. A together with the key of E as a FK
            2. The simple components of A together with the key of E as a FK
        ii. Key (all attributes)
7. N-ary Relationship Type
    a. For each n-ary relationship type (n>2) create a new relation with:
        i. Attribute (the key of S and and the key of T as FK, plus the simple attributes of B)
        ii. Key (the key of S and the key of T, except that if one of the participating entity types has participation ratio 1, it's key can be used as a key for the new relation

*simple components of composite attributes