COMP1531 Notes

Sofware Engineering Fundamentals (University of New South Wales)

COMP1531 Lecture Notes

Week 1 Lecture:

Software Engineering: A discipline that enables customers to **achieve business goals** through developing software-based systems to **solve their business problems**.

- Great emphasis on the **methodology** for managing the development process.
    - Methodology is commonly referred to as Software Development Life-Cycle (SDLC).

Software Engineering is not Programming it involves:

- **Understanding** the business problem (understanding the interaction between the system-to-be, its)
- **Creative Formulation** of ideas to solve the problem based on this understanding
- **Designing** the "blueprint" or architecture of the solution

Software engineering acts as a bridge from customer needs (problem domain) to programming implementation (solution domain)

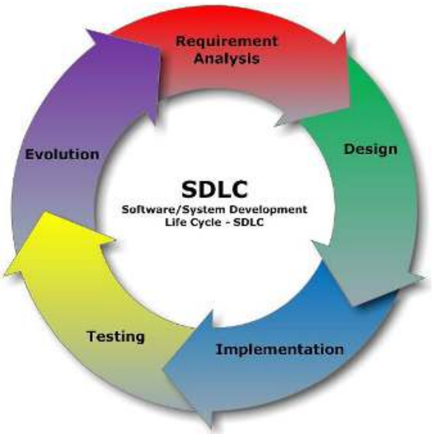
- The between clients and the programmer

Software engineers design solutions that that accurately target the customer's needs, that is, deliver value to the customer.

Why we need software engineering:

- Software Engineering provides us with a systematic and disciplined approach to software development.
- Software is intangible and software development requires imagination
- Software errors, poor design and inadequate testing have led to loss of time, money, human life. Software engineers need to prevent this by creating well developed software.

Software Development Lifecycle:



1. Requirements Analysis and Specification:
   Discover and learn about the problem domain and the “system-to-be”. Software engineers need to:
   - Analyse the problem, understand the problem definition what features/services does the system need to provide.
   - Determine both functional (inputs and outputs) and non-functional requirements (performance, security, quality maintainability, extensibility etc.)
   - Use-Case modelling, user-stories are popular techniques for analysis and documenting the customer requirements.

2. Planning and Design:
   A problem-solving activity that involves a “creative process of searching how to implement all of the customer’s requirements”. Here software engineers need to:
   - Plan out the system’s “internal structure or structural characteristics” that delivers the system’s “external behaviour” specified in the previous phase.
   - Produce software “blueprints” (design artifacts e.g. domain models)

   Often this phase overlaps with the Analysis phase.

3. Implementation:
   Encode the design in a programming language to deliver a software product.

4. Testing:
   Verify that the system works correctly and realises the goals.
   Testing process emcompasses:
   - Unit tests (individual components are tested)
   - Integration tests (the whole system is tested)
   - User acceptance tests (the system achieves the customers requirements)

5. Release and Maintenance:
   Deploying the system, fix defects and adding new functionality.
   The cycle then begins again when another feature/solution must be implemented.

Software Development Methodologies:

Can be categorised into two broad categories:

Waterfall Process:

- A linear process, where the various SDLC phases are carried out in a sequential manner.
- Plan-driven development model characterized by detail planning in terms of the:
    - Problem is identified, documented and designed.
    - Implementation task identified, scoped and scheduled
    - Development cycle followed by testing cycle
- Each phase must be fully completed, documented and signed off before moving onto the next phase.
- Simple to understand and manage due to project visibility.
- Suitable for risk-free projects with stable product statement, clear, well-known requirements with no ambiguities, technical requirements and ample resources or mission-critical applications.
- Disadvantages:
    - No working software (prototype) produced until late into the software life-cycle.
    - Rigid and not very flexible.
        - No support for fine-tuning of requirements through the cycle.
        - Good ideas need to be identified upfront; a great idea in the release cycle is a threat.
        - All requirements frozen at the end of the requirements phase (static requirements).
        - Once application as been implemented and in “testing” it is very difficult to retract or modify parts.
    - Heavy use of documentation
    - Incurs a large management overhead
    - Not suitable for projects where requirements are at a moderate risk of changing.

Iterative & Incremental:

- Processes which develop increments of functionality and repeat in a feedback loop.
- These methodologies approach solving the problem:
    - Break the big problem down into smaller pieces and prioritize them.
    - An “iteration” refers to a step in the life-cycle.
    - Each iteration results in an increment or progress through the overall project.
    - Seek customer feedback and change course based on an improved understanding at the end of each iteration.
- An incremental and iterative process:
    - Seeks to get a working instances as soon as possible.
    - Progressively deepen the understanding or “visualisation” of the target product.
- Examples:
    - Agile Methods (SRUM, XP)-methods that are more aggressive in terms of short iterations.
    - Unified Software Development Process: An iterative software development process where a system is progressively built and refined through multiple iterations, using feedback and adaptation.
        - Each iteration will include requirements, analysis, design, and implementation.
        - Iterations are timeboxed.
    - Rational Unified Process: A specific adaptation of Unified Process.

Lecture 2

Requirements: A condition or capability needed by a user to solve a problem or achieve an objective.

- Describes an aspect of what the proposed system should do or describe a constraint.
- Must help solve the client's problem
- Set of requirements as a whole represents a negotiated agreement among all stakeholders.

Functional Requirements	Non-Functional Requirements
Defines the functionality of the "system-to-be", the <u>set of services</u> provided by the system and is typically described as: ● What inputs the system should accept and under what conditions. ● The behaviour of the system. ● What outputs the system must produce and under what conditions.	Describes the <u>quality attribute</u> of the "system-to-be": ● The constraints of the functionality provided by the system. E.g. Reliability, security, maintainability, performance, efficiency. ● These requirements are quantifiable and must have a measurable way to assess if the requirement is met (metrics).

Requirements Engineering: A set of activities concerned with identifying and communicating the purpose of a software system and the context it will be used in.

It is a negotiation process where:

- Potential "users" of the system explore the requirements, agreeing what they want and what they need.
- Software engineers formulate a well-defined problem to solve, where a well-defined problem consists of:
  - A set of criteria/requirements according to which proposed solutions either definitely solve the problem or fail to solve it.
- The participants:
  - End users interested in the requested functionality
  - Customer (Business owner) interested in the cost and time-lines
  - Design team (software engineers, architects and developers) interested in how well the functionality is implemented.

Phases of Requirements Engineering:

1. Requirements Gathering (Elicitation):
   A process where customers, end-users articulate, discover and understand their requirements.
   - Customers specify: What is required, How will the intended system fit into the context of their business and How the system will be used on a day-to-day basis.
   - Developers understand the business context through meetings, market research, questionnaires and focus groups.
   - The problem statement is rarely precise.
   - Problems that may arise:
     - Limited access to stakeholders.
     - Conflicting Priorities.
     - Customers do not know what they want.
     - Missing Requirements
     - Customers change their mind
     - Too focussed on one requirement/detail
     - No clear definition of “Done”
     - Moving from abstract to concrete.

2. Requirement Analysis and Specification:
   - Starts with the customer statement of requirements or the vision statement.
   - Refine the reason about the requirements elicited
   - Scope the project, negotiate with the customer to determine the priorities - what is important, what is realistic.
   - Identify dependencies, conflicts and risks
   - Two Popular Techniques:
     - Use Case Modelling: Build a set of use-cases, to describe the tasks to be performed by the “system-to-be”.
       - Each use-case represents a dialog between user and system, helping the user achieve a goal.
       - In each dialog, user initiates an action, system responds with a reaction.
       - Build elaborate user-scenarios for each use-case that describe the interaction between user and system.
     - User Stories
   - Once analysis is completed, specifications are created where software engineers document the functional and non-functional (quality constraints of software-to-be) requirements using formal, structured notations or graphical representation to ensure clarity, consistency and completeness.
   - Finally, requirement validation: The process of confirming with the customer or user of the software that the specified requirement are valid, correct and complete.
     - Ensure developer’s understanding of the problem matches the customer’s expectations.

Test Driven Development (TDD) stipulates writing tests for the requirements during requirements analysis.
These tests are known as **User Acceptance Tests (UAT):**

- Capture the customer's assumptions about how the requirement will work and what could go wrong.
- Are defined by the customer or developer in collaboration with the customer

The customer can work with developer to write the actual test cases. A test case is particular choice of input values to be used in testing a program and expected output values.

Agile Requirements Analysis and Specification With User-Stories

- An approach used in agile software development to elicit requirements.
- A short, simple descriptions of a feature or requirement narrated from the perspective of the person who desires that capability.
    - Initial User Store (informal)
        - E.g. Student can purchase monthly parking passes online.
    - Initial User Story (formal): Uses a RGB (Role Goal Benefit) template.
        - As a <type of user[role]>, I want <some goal> so that <some reason [benefit]>
            - Role (who): Describes who will be benefited by the feature, must clearly identify the specific type of user.
            - Goal (what): Describes what the user wants from the perspective of the user.
            - Benefit (why): States why the user wants this feature. What benefit the user will get out of this feature.
        - E.g. As a student, I want to purchase a parking pass so that I can drive to school.
- A reminder to have a conservation with your customer (they shift the focus from writing about features to discussing them.
- Any stakeholder is able to write user-stories.
- Granularity of a User Story (starting from the biggest):
    - Themes: A collection of related epic user-stories
    - Epic User Stories: Covers large amount of functionality and generally too large for an agile team to complete in one iteration.
    - User Stories: Split an epic user story into multiple smaller atomic stories, so the story is small enough to be coded and tested in one iteration.
- Other details about user stories:
    - Each user story should be assigned a unique identifier.
    - User stories should also indicate estimated size.
    - User story should also indicate the priority.
    - User stories should be complemented with an "Acceptance Criteria" that defines when the requirement is "Done".
- Summary:
    - Design upfront a process for collaborative requirement gathering.
    - Identify and engage a product owner and knowledgeable subjects SMEs
    - Focus on breadth early, on depth later.
    - Break down epic stories to the right level, so the team has clarity on the requirement, "Just enough for the next step attitude".

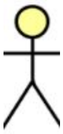
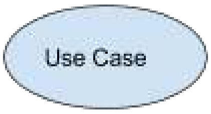
  - REFER TO LECTURE 3 NOTES ON WRITING GOOD USER STORY

Use Case Modelling:

Building a set of use-cases that describe the task to be performed by the "system-to-be" and the relation between these tasks and the outside world.

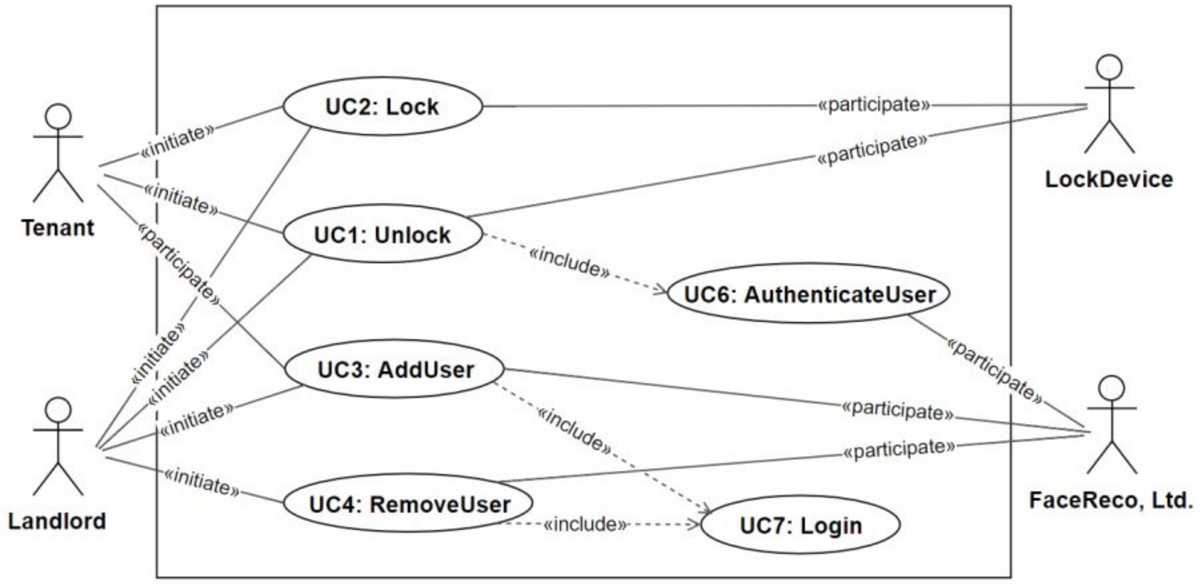
- Use Case Description: Represents a dialog between the user and the system, with the aim of helping the user achieve a goal.
- **Use cases signify what the system needs to accomplish, not how.**
- Use Cases:
  - Helps a user to achieve goals, each use-case name must include a "verb" capturing the goal achievement.
  - Each use-case description represents a dialog, where the user initiates actions and the system responds with reactions.
  - Each use-case specifies what information must pass the boundary of the system in the course of a dialog (without considering what happens inside the system)

Parts of a Use-Case Model

Representation	Description
 Actor	● Initiating Actor: Initiates the use case to achieve a goal. ● Participating Actor: Participates in the use case but does not initiate it. ○ Helps the system-to-be complete the use case.
 Use Case	● Located inside the system boundary ● Interact with other Use Cases or Actors ● Use Case is usually a verb. E.g. Unlock, Deposit, Input
<<initiate>>	● The representation of the dialog between an initiating actor and a Use Case. ● Connected with a solid line between the actor and Use Case.
<<participate>>	● The representation of the dialog between a Use Case and participating actor. ● Connected with a solid line between the Use Case the the actor.
<<include>>	● The representation of the dialog between two Use Cases. ● Connected with a dotted arrow **from** the Use Case that requires the secondary Use Case. ● This is not an option use case.
<<extend>>	● The representation of the dialog between between two Use Cases. ● Connected with a dotted arrow **from** the optional Use Case to the primary Use Case.

	● Optional Use Case or Conditional Use Case

Example) Use Case Diagram of a Lock System for an Apartment



Traceability Matrix:

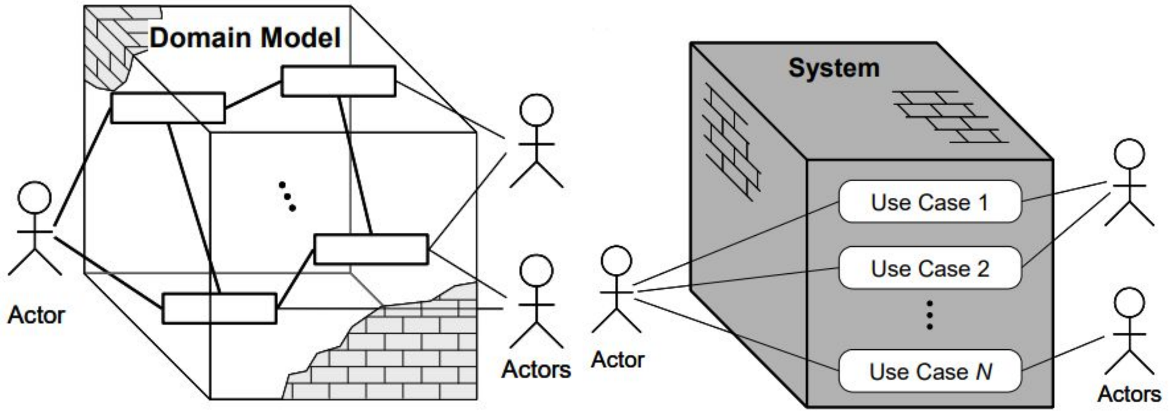
- Traceability: Refers to the property of a software artefact (use-case, a class etc) of being traceable to the original requirement that motivated its existence.
- Traceability matrices are continued through the domain model, design diagrams, etc…
- In the context of Use-Cases, the matrix serves to:
  - To check that all requirements are covered by the use cases.
  - To check that none of the use cases is introduced without a reason.
  - To prioritize the work on Use-Cases.

Domain Models (Conceptual Model or Domain Object Model)
Provides a visual representation of the problem domain, through decomposing the domain into key concepts or objects in the real-world and identifying the relationships between these objects.

- Domain modelling determines "internal behaviour" - " how elements of system-to-be interact to produce the external behaviour.

Domain Modelling vs Requirement Analysis

Domain Modelling	Requirement Analysis
Internal Behaviour	External Behaviour
In domain analysis, we consider the system as a "transparent box"	In use case analysis, we consider the system as a "black box"

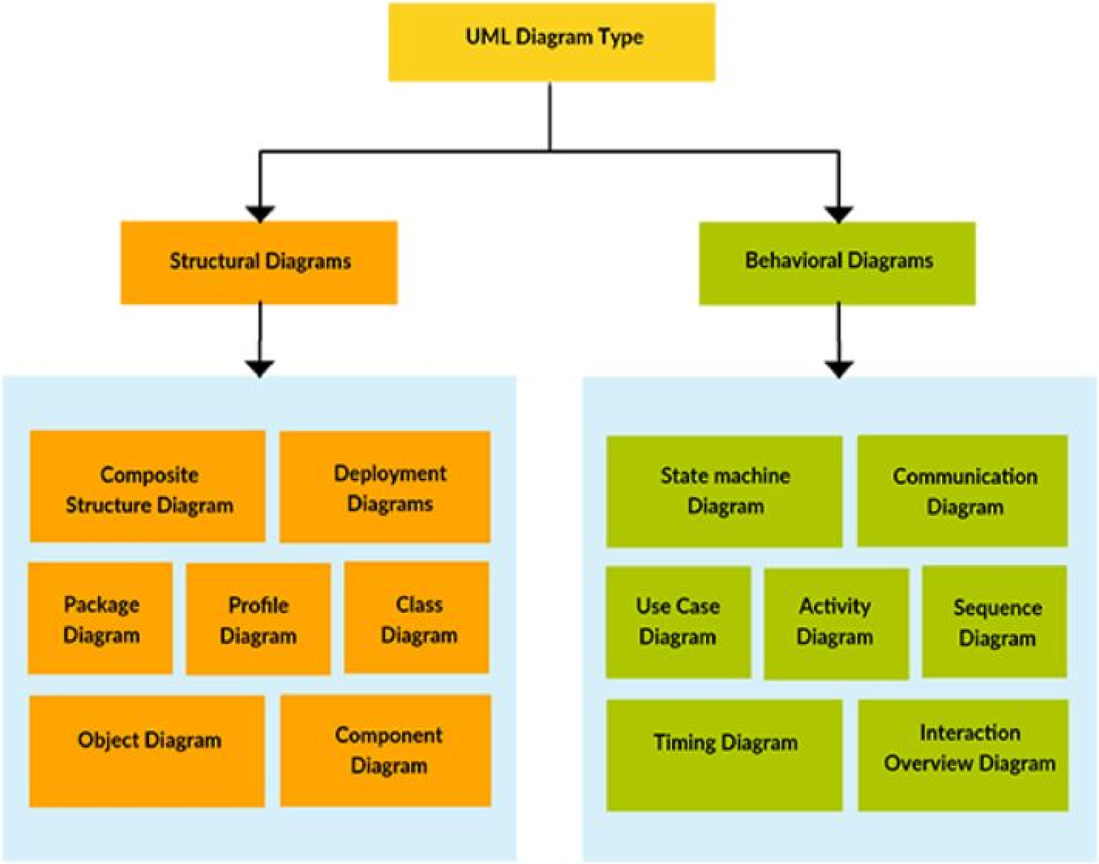


Benefits of Domain Modelling:

- Triggers high-level discussions about what is central to the problem (the core domain) and relationships between sub-parts (sub-domains).
- Ensures that the system-to-be reflects a deep, shared understanding of the problem domain as the objects in the domain model will represent domain concepts.
- Importantly, the common language resulting from the domain model, fosters **unambiguous shared understanding** of the problem domain and requirements among business visionaries, domain experts and developers.

Unified Modelling Language (UML)

- Programming languages not abstract enough for OO design
- An open source, graphical language to model software solutions, application structures, system behaviour and business processes.
- Several Uses:
  - As a design that communicates aspects of your system
  - As a software blueprint
  - Sometimes, used for auto code-generation
- UML Diagram Categories:
  - Structure Diagrams:
    - Show the static structure of the system and its parts and how these parts relate to each other.
    - They are said to be static as the elements are depicted irrespective of time (E.g. Class Diagram)
  - Behaviour Diagrams:
    - Show the dynamic behaviour of the objects in the system.
    - A subset of these diagrams are referred to as interaction diagrams that emphasis interaction between objects.



Creating Domain Models

One widely adopted technique is based on the **Object-Oriented (OO) Design Paradigm**.

Objects:

- Objects are real-world entities and could be either tangible and visible (E.g. A phone) or something intangible (E.g. Time)
- Every object has:
    - Attributes: Properties of the object
    - Behaviour: What the object can do (or methods)
- Each object encapsulates some state (the **currently assigned values** for its attributes); gives the object its identity (as state of one object is independent of another)

Object Collaboration:

- Objects interact and communicate by sending **messages** to each other.
- Objects play a client and server role and could be located in the same memory space or on different computers.
- If “object A” wants to invoke a specific behaviour on “object B”, it sends a message to B requesting the behaviour.
    - The message is typically made of three parts:
        i. The object to whom the message is addressed
        ii. The method you want to invoke on the object
        iii. Any additional information needed.

Objects’ Interface:

- An object’s interface is the set of the object’s methods that can be invoked on the object.
- The interface is the fundamental means of communication between objects.

Classes:

- Classes: Serve as a “blueprint” defining the attributes and methods (behaviour) of this logical group of objects.
    - Classes group objects that share some common properties and behaviour.

Object and Classes

- An object is **instantiated** from a class and the object is said to be an **instance** of the class.
    - An object instance is a specific realization of the class.
- Two object instances from the from the same class share the same attributes and methods, but have their own object identity and are independent of each other.
    - An object has state but class does not
    - Two object instances from the same class share the same attributes and methods, but have their own object identity and are independent of each of each other.

Key Principles of OO Design:

Abstraction:

- The process of designing classes so they are reduced to their necessary attributes and methods.
- Abstraction allows us to isolate parts of the problem and consider its solution outside of the main solution.
    - An **Abstract**:
        - Focus on the common essential qualities of the object.
        - Focus on the current application context.

Encapsulation:

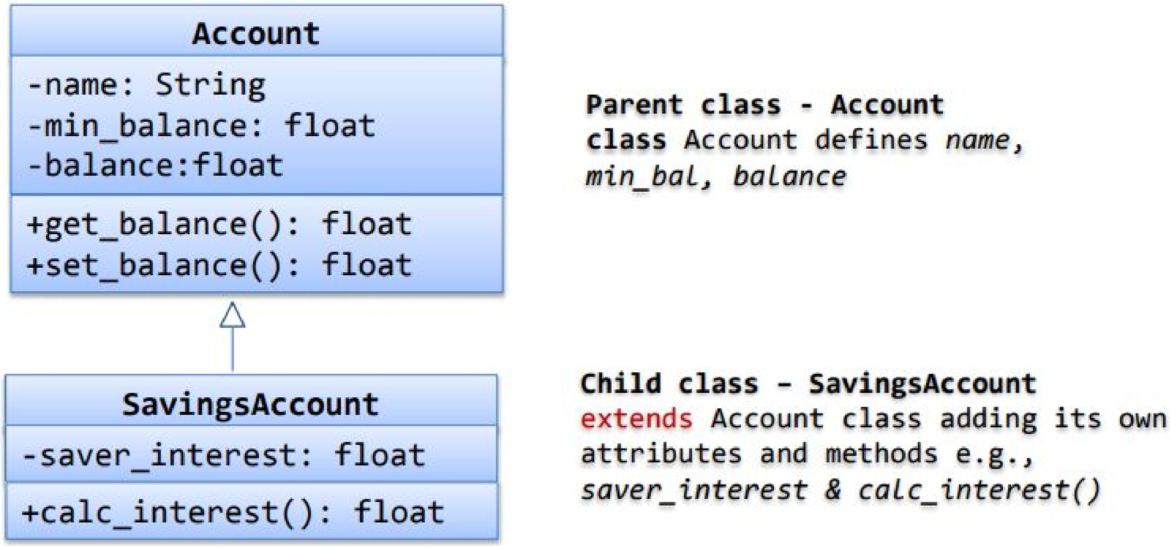
- The process of hiding an object's data and processes from its environment.
- Only the object can alter its own data. <u>Encapsulation means that objects control their own private attributes using their methods</u>.
- This enables the creation of robust and reusable classes of objects.
- Importance of Encapsulation:
    - Ensures that an object's state is in a consistent state.
    - Increases Usability
        - Restricting access to the interface makes the object's role is more defined.
        - Clear contract between invoker (caller of the object) and the provider (the object), invoker is adhering to the method signature, guarantees consistent behaviour.
    - Abstracts the implementation by reducing the dependencies so that a change to a class does not cause a rippling effect on the system.

Relationships

1. Inheritance [“is-a” or “is-a-kind-of” relationship]:

- The ability of object to take on the characteristics of their parent class or classes.
- Supports modularity and robust codes
- Models a relationship between classes:
    - Parent/Base Class: A class which is more general.
    - Sub Class: A class which is a more specialised version of the base class.
    - Models a “is-a” relationship. (E.g. “A dog is-a type of pet”)
- Implementing Inheritance:
    - Create a new class (sub-class), that inherits common properties and behaviour from a base class (parent-class or super-class).
        - This child class inherits/is-derived from the parent class.
    - Sub-class can extend the parent class by defining additional properties and behaviour specific to the inherited group of objects.
    - Sub-class can override methods on the parent class with their own specialised behaviour.
        - Overriding: Methods (or properties) which replace an inherited method (or property).

Example of Inheritance)

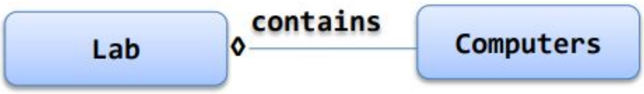


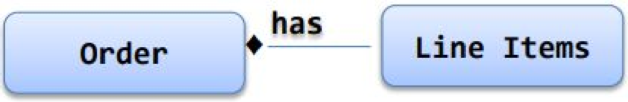
Parent class - Account
class Account defines *name*, *min_bal*, *balance*

Child class - SavingsAccount
extends Account class adding its own attributes and methods e.g., *saver_interest & calc_interest()*

*Note the use of a hollow arrow pointing FROM child class TO parent class.

2. Association [“has-a” or “contains” relationship]

- A special type of relationship between two classes, that shows that the two classes are:
  - Linked to each other
  - Or combined into some kind of “has-a” relationship, where one class “contains” another class.
    - A course-offering (object A) has {contains} students (object B)
  - Modelled in UML as a line between two classes.
- Can be further refined as:
  - Aggregation Relationship: The contained item is an element of a collection but it can also exist on its own.
    - E.g. A student at a University: Can exist purely as a student but can also be contained within a course-offering.
    - 

    - Note the use of the hollow diamond. Pointing FROM the contained object TO the container object.
  - Composition Relationship: The contained item is an integral part of the containing item.
    - E.g. An engine in a car.
    - 

    - Note the use of the filled diamond. Pointing FROM the contained object TO the container object.
    - The box with the diamond is the container.
    - The contained item cannot exist without the container and container cannot exist without the container.

Noun/Verb Analysis

- Overview
  - Analyzing textual description of the domain to identify noun phrases. These will become the “Candidate Conceptual Classes”
  - The verbs will (potentially) become the methods associated with the classes.

CRC Models:

- A collection of CRC cards, that specifies the OO design of the system.
- CRC Cards:
    - CRC stands for:
        - **C**lass: An object-oriented class name, represents a collection of similar objects.
        - **R**esponsibility: What does the class know? What does this class do?
        - **C**ollaboration: Relationship to other classes (What classes does this class use?)
- How to create a CRC Model:
    - Read the description
    - Identify core classes (look for nouns)
    - Create a card per class (start with just class names)
    - Add responsibilities (look for verbs)
    - Add collaborations
    - Add more classes as you discover them
    - Refine by identifying inheritance.

Evolving CRC domain models into UML class diagrams where:

- Concepts are represented as classes
- Collaborations between the classes established as relationships.
    - Depending on the kind of relationship, we can use the difference notations.

CLASS DIAGRAMS (Finally)

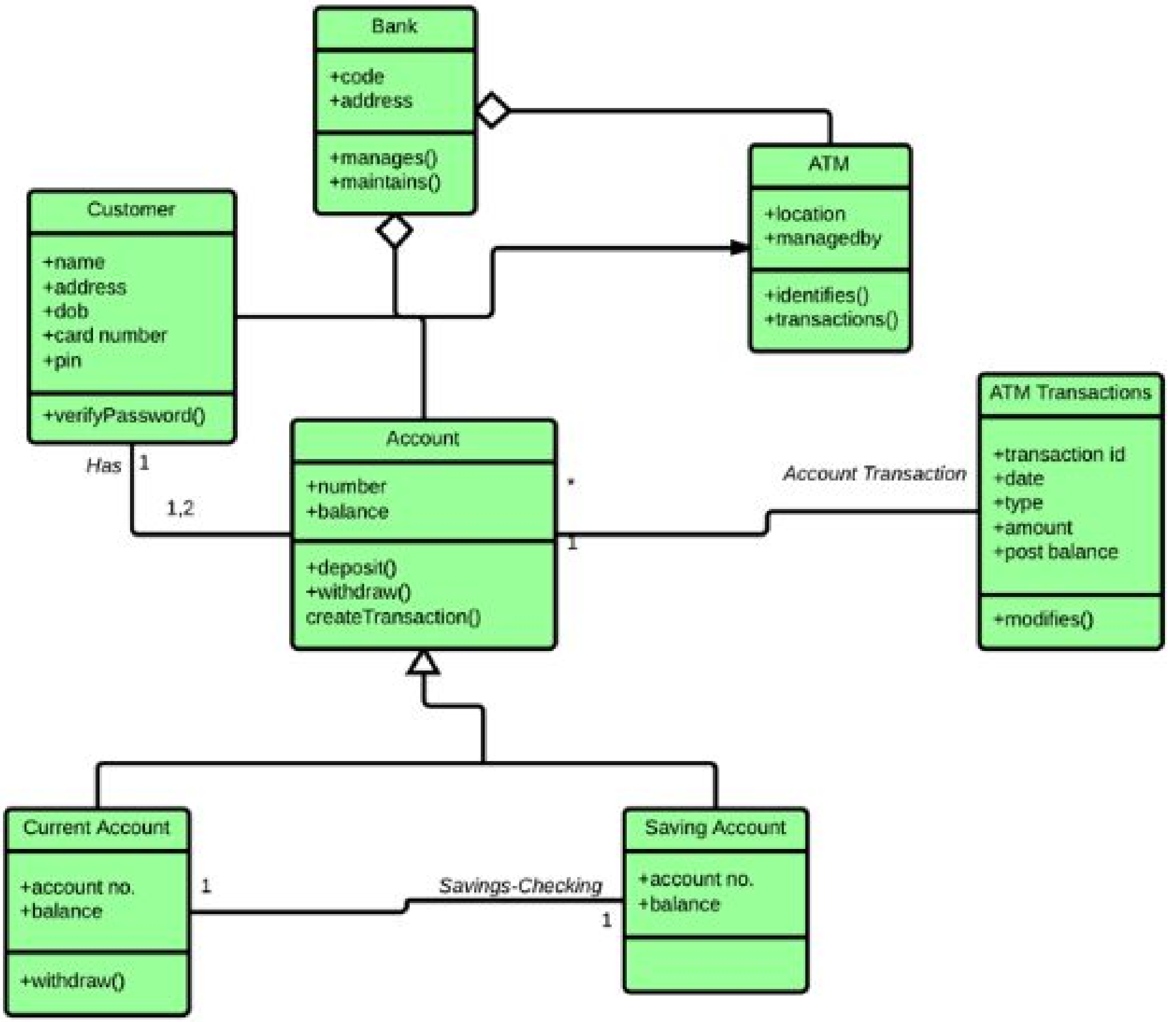
Basics:

- Upper Section: Contains the name of the class. This section is always required, whether you are talking about the classifier or an object.
- Middle Section: Contains the attributes of the class.
  - Use this section to describe the qualities of the class.
  - This is only required when describing a specific instance of a class.
- Bottom Section: Includes class operations (methods).
  - Displayed in list format, each operation takes up its own line.
  - The operation describe how a class interacts with data.

Member Access Modifiers:
All classes have different access levels depending on the access modifier (visibility). Note the corresponding symbols.

- Public (+)
- Private (-)
- Protected (#)
- Package (~)
- Derived (/) - From inheritance

Example of an ATM System class diagram)


Bank
+code
+address
+manages()
+maintains()
ATM
+location
+managedby
+identifies()
+transactions()
Customer
+name
+address
+dob
+card number
+pin
+verifyPassword()
Has
1
1,2
Account
+number
+balance
+deposit()
+withdraw()
createTransaction()
*
1
Account Transaction
ATM Transactions
+transaction id
+date
+type
+amount
+post balance
+modifies()
Current Account
+account no.
+balance
+withdraw()
1
Savings-Checking
1
Saving Account
+account no.
+balance

Lecture Week 5

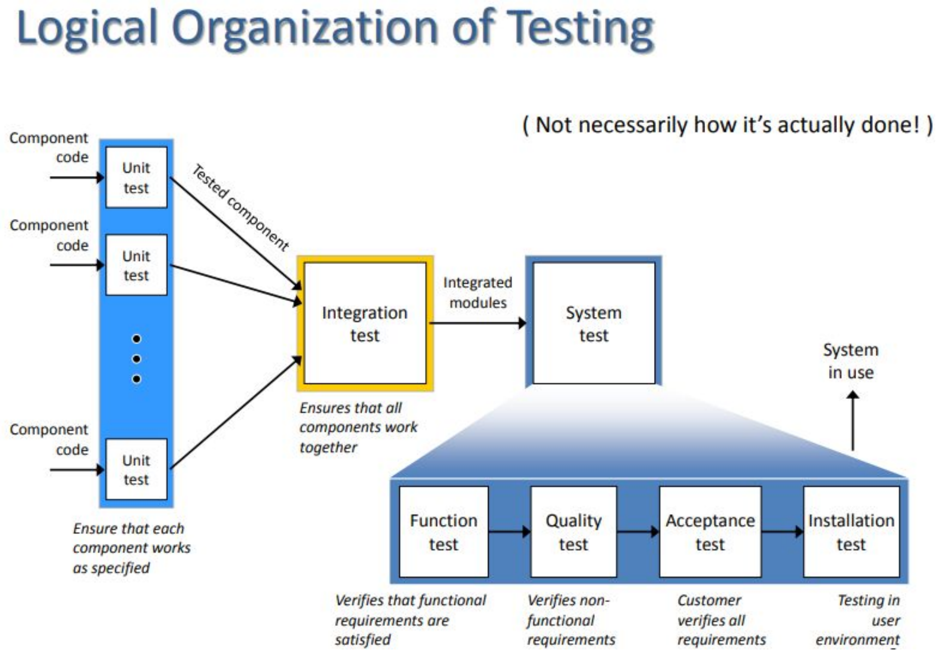
Software Testing:
Fault/Defect/Bug: A wrong or incorrect hardware or software element of a system that can cause the system to fail.

Key Tradeoff: Testing as many potential cases as possible while keeping the economic cost limited.

Goal: Find faults as cheaply and quickly as possible.

- Ideally, we would design a single “right” test case to expose each fault and run it.
- In practice, we have to run many “unsuccessful” test cases that do not expose any faults.

Logical Organisation of Testing



Logical Organization of Testing

(Not necessarily how it’s actually done!)
Component code
Unit test
Component code
Unit test
Component code
Unit test
Ensure that each component works as specified
Tested component
Integration test
Ensures that all components work together
Integrated modules
System test
System in use
Function test
Verifies that functional requirements are satisfied
Quality test
Verifies non-functional requirements
Acceptance test
Customer verifies all requirements
Installation test
Testing in user environment

Testing Terminology:

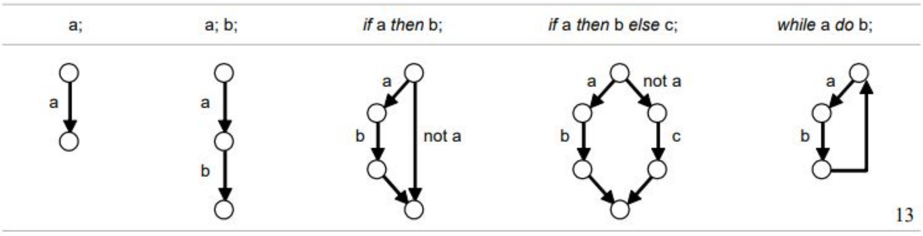
- Black Box Testing:
  - A testing approach commonly adopted by customers, such as User Acceptance Testing (UAT)
  - Testing a running program with a set of inputs without looking at the implementation.
- White Box Testing:
  - Testing with test data with knowledge of implementation (system, architecture, algorithms used, program code)
- Regression Testing:
  - Verify software that was previously developed and tested still performs after the program has been changed or its interfaces with other software.

Test Coverage:
Measures the degree to which the specification or code of a software program has been exercised by the tests.

Code Coverage:
Measures the degree to which the source code of a program has been tested

- Code Coverage Criteria:
    - Equivalence testing
    - Boundary testing
    - Control-flow testing
    - State-based testing
- Equivalence Testing: Is a black-box testing method that divides the space of all possible inputs into equivalence groups such that the program “behaves the same” on each group.
    - Steps:
        - 1. Partition the values of input parameters into equivalence groups.
        - 2. Choosing the test input values
- Boundary Testing: A special case of equivalence testing that focuses on the boundary values of input parameters.
    - Based on the assumption that developers overlook special cases at the boundary of equivalence classes.
    - Selecting Values:
        - Zero, min/max values, empty sets, empty strings and null.
        - Edge values or values used to make “decisions” in the code.

Code Coverage: Control-flow Testing

- Statement Coverage: Each statement executed at least once by some test case.
- Edge Coverage: Every edge (branch) of the control flow is traversed at least once by some test case.
    - Constructing a Control Graph:
    - 

- Condition Coverage: Every condition takes TRUE and FALSE outcomes at least once in some test case.
- Path Coverage: Finds the number of distinct paths throw the program to be traversed at least once.

Test Driven Development
Every step in the development process must start with a plan of how to verify that the result meets a goal.
Developer should not create a software artifact (a UML diagram, or source code) unless they they know how it will be tested
An important principle in XP, Scrum.

Exception Handling

Exception:

- An error that happens during the execution of a program, causing a program to terminate abruptly.
- Could be caused by providing wrong input to the data, run out of memory, file or network resources not available.
- DIFFERENT TO PROGRAM BUGS

Exception Handling:

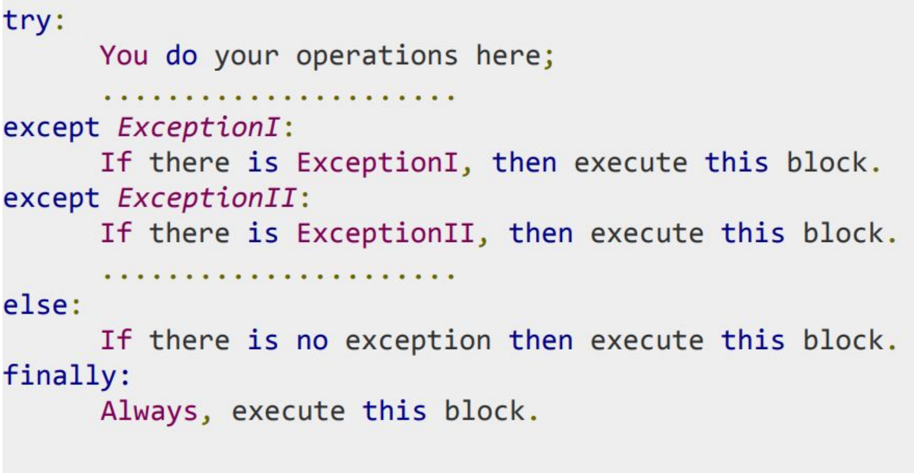
- Enables handling such situations gracefully and avoid intermittent failures.
- Exception handling is critical for creating robust and stable applications.

Exception Handling in Python:

In Python, when an error occurs:

- An exception is raised through creating a Python object Exception
- The normal flow of the program is disrupted
- This exception must be handled, else program terminates
- Use Python’s try/except clause to handle exceptions

Syntax:



```
try:
      You do your operations here;
      ......................
except ExceptionI:
      If there is ExceptionI, then execute this block.
except ExceptionII:
      If there is ExceptionII, then execute this block.
      ......................
else:
      If there is no exception then execute this block.
finally:
      Always, execute this block.
```

Assert

- Great for testing generally
- Golden Rule: Do not use assertion for data validation!

Week 6 Lecture

Incremental and Iterative Project Life-Cycle Approaches

1. Break the big problem down into smaller pieces and prioritize them.
2. An “iteration” refers to a step in the life-cycle
3. Each “iteration” results in an “increment” or progress through the overall project.
4. Seek the customer feedback and change course based on improved understanding at the end of each iteration.

An incremental and iterative process

- Seeks to get to a working instance as soon as possible.
- Progressively deepen the understanding or “visualisation” of the target product.

Rational Unified Process (RUP)

An iterative software development process.

Consists of 4 major phases:

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

Extreme Programming (XP)

A prominent agile software engineering methodology that:

- Focuses on providing the highest value for the customer in the fastest possible way.
- Acknowledges changes to requirements as a natural and inescapable aspect of software development.
- Places higher value on adaptability (to changing requirements) over predictability (defining all requirements at the beginning of the project)- being able to adapt is a more realistic and cost-effective approach.
- Aims to lower the cost of change by introducing a set of basic principles and practices to bring more flexibility to changes.

XP Core Principles:	Description:
High Quality	● Pair-Programming: Code written by pairs of programmers working together intensely at the same workstation, where one member of the pair “codes” and the other reviews. ● Continuous Integration: Programmers check their code in and integrate several times per day. ● Sustainable pace: Moderate, Steady Pace ● Open Workspace: Open Environment ● Refactoring: Series of tiny transformations to improve the structure of the system. ● Test-Driven Development: Unit Testing and User Acceptance Testing.
Simple Design	● Focus on the stories in the current iteration and keeps the design simple and expressive. ● Migrate the design of the project from iteration to iteration to be the best design for the set of stories currently implemented. ● Spike solutions, prototypes, CRC cards are popular techniques during design. ● Three mantras: ○ Consider the simplest design for the current batch of user stories. ○ Resist the temptation to add the infrastructure before it is needed. ○ XP developers don’t tolerate duplication of code, wherever they find it, they eliminate it.
Continuous Feedback	● Developers receive constant feedback by working in pairs. constant testing and continuous integration. ● XP team receives daily feedback on progress and obstacles through daily stand-up meetings. ● Customers get feedback on progress with user acceptance scores and demonstrations at the end of each iteration ● XP developers deliver value to customer through producing working software progressively at a steady pace and receive customer feedback and changes that are accepted.

Extreme Programming turns the traditional software development process sideways, flipping the axis of the previous chart, where we visualise the activities, keeping the process itself a constant.

XP Planning Game: Consists of 4 Major Steps:

1. Initial Exploration
   a. Developers and customer have conversations about the system-to-be and identify significant feature.
   b. Each feature broken into one or more user stories.
   c. Developers estimate the user story in user story points based on team's velocity.
      i. Stories that are too large or small are difficult to estimate. An epic story should be split into pieces that aren't too big.
      ii. Velocity: Sum of the estimates of the completed stories. A measure of productivity.
      iii. Developers have a more accurate idea of average velocity after 3 or 4 weeks, which is used to provide better estimates for ongoing iterations.

2. Release Plan
   a. Negotiate a release date
      i. Customers specify which user stories are needed and the order for the planned date (business decisions)
      ii. Customers can't choose more user stories than will fit according to the current project velocity.
      iii. Selection is crude, as initial velocity is inaccurate. RP can be adjusted as velocity becomes more accurate.
   b. Use the project velocity to plan:
      i. By time: Compute number of user stores that can be implemented before a given date (multiply number of iterations be the project velocity)
      ii. By Scope: How long a set of stories will take to finish divide the total weeks of estimated user stories by the project velocity.

3. Iteration Planning
   a. Use the release plan to create iteration plans.
   b. Developers and customers choose an iteration size: typically 1 or 2 weeks.
   c. Customers prioritise user stories from the release plan in the first iteration, but must fit the current velocity.
   d. Iteration ends on the specific date, even if all the stories aren't done.
   e. Defining "done"- A story is not done until all its acceptance tests pass.

4. Task Planning
   a. Developers and customers arrange an iteration planning meeting at the beginning of each iteration.
      i. Customers choose user stories for the iteration from the release plan but must fit the current project velocity.
   b. The velocity in task days (iteration planning) overrides the velocity in story weeks (release planning) as it is more accurate.

Project Velocity:

- A measure of the team's productivity. Estimated from the number of user-story points that the team can complete in a particular iteration.
- Allows an estimation of Project Duration = Total Work Size/ Project Velocity.

Project Tracking:

- The recording the results of each iteration and use those results to predict what will happen in the next iteration.
    - Tracking the total amount of work done during each iteration is the key to keep the project running at a sustainable, steady pace.
    - Use a velocity chart or burn-down chart to track the project velocity which shows how many story points were completed (passed the UAT)
        - Velocity Chart: Track the project velocity, which shows how many story points were completed.
        - Burn Down Chart: Shows the week-by-week progress. The slope of the chart is a reasonable predictor of the end-date. The height of the bars may not always decrease week after week as new stories maybe added.

XP Scenarios:

- Useful for problem domains where requirements change, when customers do not have a firm idea of what they want.
- XP was set up to address project risk.
    - Used to mitigate risk and increase likelihood of success.
- Group size of 2-12
- An extended development team comprising managers, developers and customers all collaborating closely.
- XP also places great emphasis on testability and stresses creating automated unit and acceptance tests
- XP projects deliver greater productivity.

Agile Development Disadvantages:

- Close collaboration may not be ideal for development outsourcing, clients and developers separated geographically, or business clients who simply don't have the manpower, resources.
- Emphasis on modularity, incremental development and adaptability may not suite clients desiring contracts with firm (strong) estimates and rigid schedules.
- Reliance on small self-organised teams makes it difficult to adapt to large software projects and neglects to consider the need for leadership while team members get used to working together.
- Lack of comprehensive documentation can make it difficult to maintain or enhance the software.
- Agile development - Requires highly experienced software engineers who know how to both work independently and interface effectively with business users.

Deciding upon Methodology:

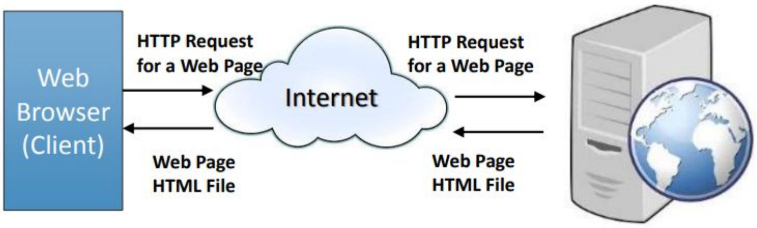
- Generally, SaaS (Software as a Service) and Web 2.0 applications that require moderate adaptability are likely to be suited to agile system.
- Mission-critical applications such as military, medical that require high degree of predictability are more suited to waterfall.
    - Software as a function rather than a service.

Week 6 HTML and Flask

HTTP and World Wide Web:

World Wide Web (WWW): An information sharing model that is built on top of the Internet (infrastructure), where information is:

- Structured as documents “web pages” written in HTML, connected using hypertext links forming a huge “web” of connected information.
- How a web page is assembled:
    - A client requests a webpage by specifying the URL or clicking on a hyper link.
    - Browser sends a HTTP request to the web page server named in the URL and requests for the specific document.
    - The web server locates the requested file and sends a HTTP response.
        - If not found then an error message 404 is returned
        - If it is found the server returns the requested file to the browser
    - Browser parses the HTML document and assembles the page.
- Web Architecture Diagram:



HTTP Session State

- HTTP is a stateless protocol. Server and client are only aware of each other during a current request.
- Neither client no server can retain information between different requests across web pages.
- Then, how is it possible to customise the content of a website for a user:
    - Cookies
    - Sessions
    - Hidden Variables (when the current page is a form)

Server-Side Processing (Eg, PHP, Perl, FLASK, Django, etc)

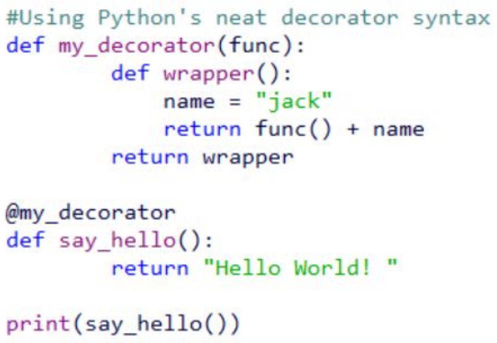
- Receives dynamic web page request
- Server processes user input, renders the dynamic web page to be returned to the client for display on browser.

Client-Side Processing (Eg, Javascript)

- Processing needs to be “executed” by the browser to:
  - Complete the request for the dynamic page. (Valid User-Input)
  - Create the dynamic web page
- More responsive UI and lowers the bandwidth cost.

Function Decorators:

- Function decorators are simply wrappers to existing functions
- They are useful for extending the behaviour of functions without having to actually modify them.
- A decorator function basically takes a function as an argument, generates a new function that augments the work of the original function and returns the newly generated function.
  - Example:

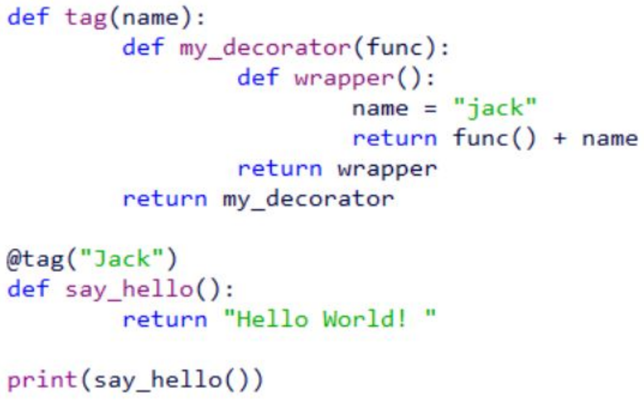


```
#Using Python's neat decorator syntax
def my_decorator(func):
        def wrapper():
            name = "jack"
            return func() + name
        return wrapper

@my_decorator
def say_hello():
        return "Hello World! "

print(say_hello())
```

- Alternatively: Instead of hard-coding the variable name, it could actually be passed in as an augment to the decorator function.
  - For example the function my_decorator can be wrapped inside another function tag, which could take in the name argument.



```
def tag(name):
        def my_decorator(func):
                def wrapper():
                        name = "jack"
                        return func() + name
                return wrapper
        return my_decorator

@tag("Jack")
def say_hello():
        return "Hello World! "

print(say_hello())
```

Flask:

- A micro web framework written in Python, developed by Armin Ronacher
  - Aims to provide a simple. solid core but designed as an extensible framework
- For developers, you can choose which packages to include or write your own customer extensions.
- Two main dependencies:
  - Werkzeug which supports routing (request + response), utilizing functions such as debugging and WSGI (Web Server Gateway interface - standard interface between web server and web applications)
  - Jinja2, a powerful templating language to render dynamic web pages

Jinja2:

- Essentially *.html files with static response text along with placeholder variables and programming logic for the dynamic parts.
- Use delimiters such as {% … %} for embedding programming logic
- {{...}} for outputting the results of an expression or variable
- Process of replacing the variables with actual values and returning a final response is called rendering.

Week 7 Lecture

Reasons for Software Rot:

- We write bad code:
  - We do not know how to write better code.
  - Requirements change in ways that original design did not anticipate
  - Changes requires refactoring, which is an investment of time and resources
    - Businesses may not like this as it is costly. May prefer quick and dirty solutions.
    - Changes may be made by different developers who do not understand the original philosophy.

Design Smells:

When software rots it smells.

- A design smell:
  - Is a symptom of poor design
  - Often caused by violation of key design principles
  - has structures in software that suggests refactoring.
    - Refactoring: The process of restructuring (changing the internal structure of software) software to make it easier to understand and cheaper to modify without changing its external, observable behaviour.

Design Smells

Design Smell	Description
Rigidity	Tendency of the software being too difficult to change even in simple ways. A single change causes a cascade of changes to other dependent modules.
Fragility	Tendency of the software to break in many places when a single change is made.
Immobility	Design is hard to reuse. Design has parts that could be useful to other systems, but the effort needed and risk in disentangling the system is too high.
Viscosity	Software Viscosity: Changes are easier to implement through “hacks” over “design preserving methods” Environment viscosity: Development environment is slow and

	in-efficient.
Opacity	Contains constructs that are not currently useful. Developers ahead of requirements
Needless Repetition	Design contains repeated structures that could potentially be unified under a single abstraction Bugs found in repeated units have to be fixed in every repetition.

Characteristics of Good Design:
Good software aims for building a system with loose coupling and high cohesion among its components.

- Coupling: The degree of interdependence between components or classes.
  - High coupling occurs when:
    - Component A depends on the internal workings of component B and is affected by the internal changes to component B.
  - Leads to a complex system, with difficulties maintaining.
  - Aim for loosely coupled classes.
  - Zero-Coupled classes are not usable. There must be a balance.

- Cohesion: The degree to which all elements of a component or class or module work together as a functional unit.
  - Highly Cohesive Modules are:
    - Much easier to maintain and less frequently changed and have higher probability of reusability.

SOLID Principles
Used to achieve High Cohesion and Low Coupling -> Avoid design smells.

- **S**RP - Single Responsibility Principle
- **O**CP - Open Closed Principle
- **L**SP - Liskov Substitution Principle
- **I**SP - Interface Segregation Principle
- **D**IP - Dependency Inversion Principle

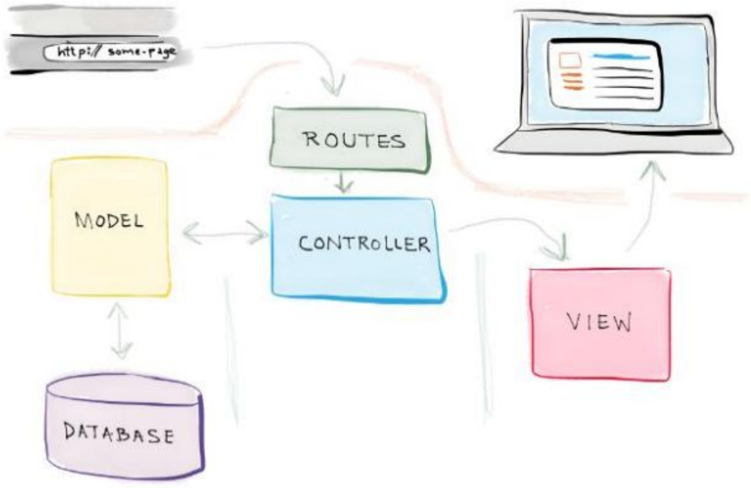
SRP: “A class should have one reason to change”

- Cohesion and SRP: The forces that cause the module to change.
- Reasons behind SRP:
  - Changes in requirements means change in responsibilities.
  - A cohesive responsibility represents a single axis of change -> a class should only have one responsibility.
    - A class with several responsibilities creates:
      - Unnecessary couplings between those responsibilities.
      - Change to one responsibility may impair the class’s ability to meet the others.
      - Leads to fragile designs that break in unexpected ways when changed.
- Advantages of SRP:
  - Helps to achieve “highly cohesive” classes that have:
    - Readability: Easier to focus one responsibility and you can identify the responsibility.
    - Reusability: The code can be reused in different contexts.
    - Testability: Each responsibility can be tested in isolation. When a class encapsulates several responsibilities, several test-cases are required.

Open Closed Principle: “Software Entities (classes, modules, functions) should be open for extension but closed for modification.”

- Modules that satisfy OCP have two primary attributes:
  - Open for extension: As requirements change, the module can be extended with new behaviours to adapt to the changes.
  - Closed for modification: Extending the behaviour of the module must not require change the original source, or binary code of the module.
- OCP reduces rigidity:
  - A change does not cause a cascade of related in dependent modules.
- How to write software that is Open for extension but closed for modification:
  - Using abstraction and dynamic binding.
  - Abstractions are implemented as abstract base classes, that aren’t fixed yet, represent an unbound group of possible behaviours.
- Conformance to OCP:
  - Difficult and requires experience. There is also a degree of risk as a result of some educated guessing.
  - Expensive: Increase the complexity of software design.
    - Apply OCP only when it is needed for the first time.
  - Yields and Benefits: Flexibility, Reusability and Maintainability.

Separation of Concerns Using MVC (Model-View-Controller):
A software architectural pattern that decouples data access, application logic and user interface into three distinct components.



View:
This is the presentation layer and provides the interaction that the user sees. (E.g. a Web Page).

- View component takes inputs from the user and sends actions to the controller for manipulating data.
- View is responsible for displaying the results obtained by the controller from the model component in a way that user wants them to see or a predetermined format.
- It is the responsibility of the controller to choose a view to display date to the user.

Model:
This is the processing layer.

- Holds all the data, state
- Responds to instructions to change change of state (from the controller).
- Responds to requests for information about its state (usually from the view).
- Sends notifications of state changes to “observer”(view)

Controller:
The glue between user and processing (Model) and formatting (View) logic.

- Accepts the user request or inputs to the application, parses them and decides which type of Model or View should be invoked.

Benefits of Separation

- Abstraction of Application Architecture into three high-level components (Model, View, and Controller). Model has no knowledge of the View that is provided to the to the user.
- Support multiple views of the same data on different platforms at the same time.
- Enhances testability.

Weaknesses of Separation:

- Complexity
- Cost - Active models could overload the view with update requests.

Lecture Week 8

Databases: Represents a logically coherent collection of related data.

Database Management System (DBMS): A software application that allows users to:

- Create and maintain a database (DDL)
- Define queries to retrieve data from the database
- Perform transactions that cause some data to be written or deleted from the database (DML)
- Provides concurrency, integrity, security to the database.
- A database and DBMS are collectively referred to as a database system

Data Models: Describes how the data is structured in the database.

There are several types:

- Relational Model:
  - A data structure where data is stored as a set of records known as tables.
  - Each table consists of rows of information (also called a tuple)
  - Each row contains fields known as columns
- Document Model:
  - Data is stored is hierarchical fashion. Eg XML
- Object-Oriented Model:
  - A data structure where data is stored as a collection of objects
- Object-Relational:
  - A hybrid model that combines the relational and the object-oriented database models.

Database Schema: Adheres to a data model and provides a logical view of the database, defining how the data is organised and the relationships between them and is typically set up at the beginning.

- A database schema instance is the state of the database at a particular instance of time.

Relational Database System
A Relational Database Management System (RDBMS) is a DBSM that:

- Based on the relational data model, stores data as TUPLES or Records in Tables

Database Design Steps:

- Requirements analysis : Identify data operations
- Data Modelling: High level and abstract engineering
- Database Schema Design: Detailed, relational, model/tables
- Database Implementation: Database implementation (create instance of schema)
- Build Operations/Interface: Build operations/interface (SQL, stored procedures, GUI)
- Performance Tuning: Performance tuning (physical re-design)
- Schema Evolution: Logical schema re-design

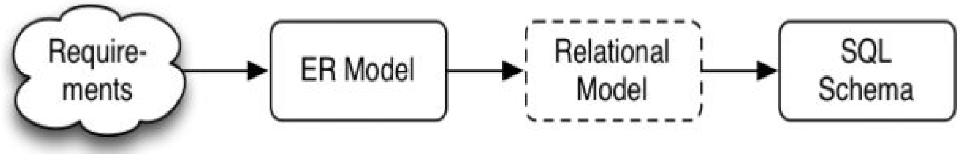
Data Modelling:

Consists of Building:

- Logical Models: Abstract Models, e.g. ER Model or OO Model
- Physical Models: Record-Based models, e.g. Relational Model, classes which deal with the physical layout of data in storage.

Strategies for designing a database:

- Design using abstract model (Conceptual Modelling)
- Map to physical model (Implementation-level modelling)



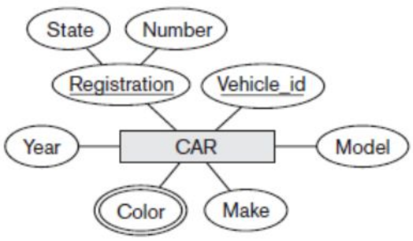
Aims of Data Modelling:

- Describe what data is contained in the database.
- Describe Relationships between data items
- Describe constraints on data
- Data modelling is a design process

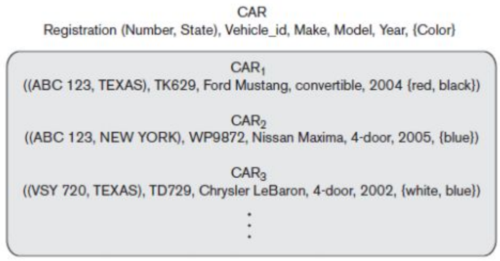
ER Diagrams

The world is viewed as a collection of inter-related entities.
ER modelling uses three major modelling constructs:

- Entity: A thing or object of interest in the real-world and is distinguishable from other objects.
- Attribute: A data item or property of interest describing the entity, the can be:
    - Simple: (Cannot be broken into smaller sub-parts) e.g. Age
    - Composite: (Have a hierarchy of attributes): Address which can be broken into Street address, city, state, etc.
    - Single-Valued (have only one value for each entity)
    - Multi-valued (have a set of values for each entity)
- Entity-Set: Can be either:
    - A set of entities with the same set of attributes
    - An abstract set of entities with the same set of attributes



An ER diagram showing an entity-set CAR with two key attributes (registration and vehicle_id), three single-valued attributes (year, model, make) and a multi-valued attribute (color)



CAR
Registration (Number, State), Vehicle_id, Make, Model, Year, {Color}

CAR_1
((ABC 123, TEXAS), TK629, Ford Mustang, convertible, 2004 {red, black})

CAR_2
((ABC 123, NEW YORK), WP9872, Nissan Maxima, 4-door, 2005, {blue})

CAR_3
((VSY 720, TEXAS), TD729, Chrysler LeBaron, 4-door, 2002, {white, blue})

⋮

An entity set CAR with three entities

17

- Discerning Entities:
    - Define a key (super-key): Any set of attributes
        - Whose set of values are distinct over entity set.
        - Natural (name + address + birthday) or artificial (SSN)
    - Candidate Key = any super-key, but not (name) or (address)
    - Primary Key = A candidate key designated by DB Designer that uniquely identifies an entity.
    - Examples) Customer = (custNo, name, address, taxFileNo)
        - Definite Super Keys: Any set of attributes involving custNo or taxFileNo
        - Possible Super Keys: e.g. (name, address) ⟵ Name AND Address
        - Unlikely Super Keys: e.g. (name), (address) ⟵ Name OR Address

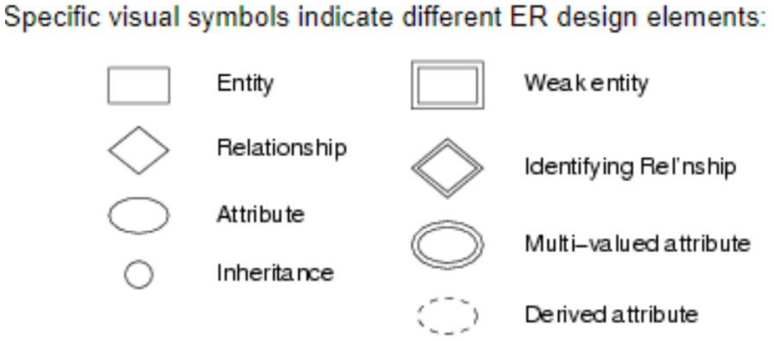
- Relationship: Relates two or more entities.
  - E.g. Joe Smith (a STUDENT entity) ENROLLED IN (relationship) COMP1531 (a course entity)
- Relationship Set: Set of similar relationships, associating entities belonging to one entity-set to another:
  - Degree = The number of entities involved in the relationship.
    - ER model, >= 2
  - Cardinality = #Associated entities on each side of the relationship.
    - E.g: Many to one is N:1. One to many is 1:M, One to one 1:1

Difference between ER Models and OO Models:
ER doesn't consider operations

Finally to....
Entity Relationship Diagrams: A graphical tool for data modelling.
Consists of:

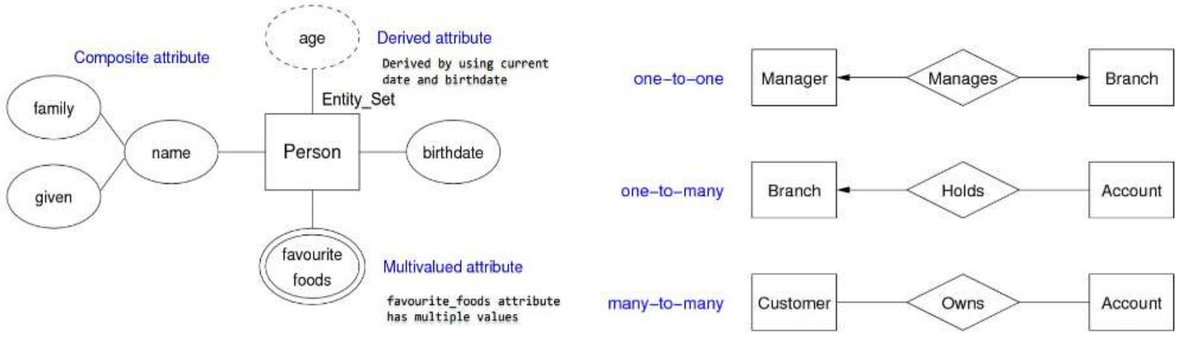
- A collection of entity set definitions
- A collection of relationship set definitions
- Attributes associated with entity and relationship sets
- Connections between entity and relationship sets

Symbols:



Specific visual symbols indicate different ER design elements:

Examples of Attribute Notation and Cardinality in Relationship Sets:



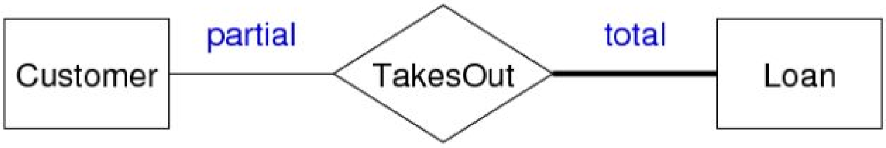
Level of Participation Constraints:
Let:

- R be the set of Relationships
- A be the set of Entities

Total Participation: FOR ALL a in A participates in >=1 relationship in R
Partial Participation: FOR some a in A participates in relationships in R

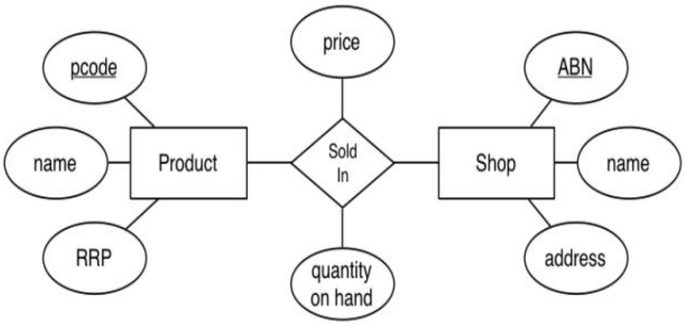
Examples:

- Every bank loan is associated with at least one customer.
- Not every customer in a bank has a loan.



Relationship Type with attributes:

Example:

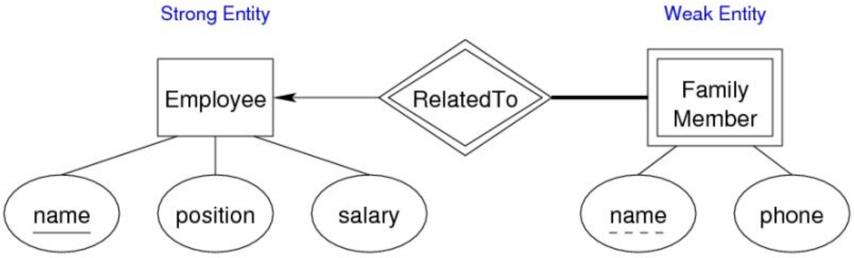


(price and quantity are related to products in a particular shop)

Weak Entity Set:

- Has no key of its own;
- Exists only because of association with strong entities

Example:



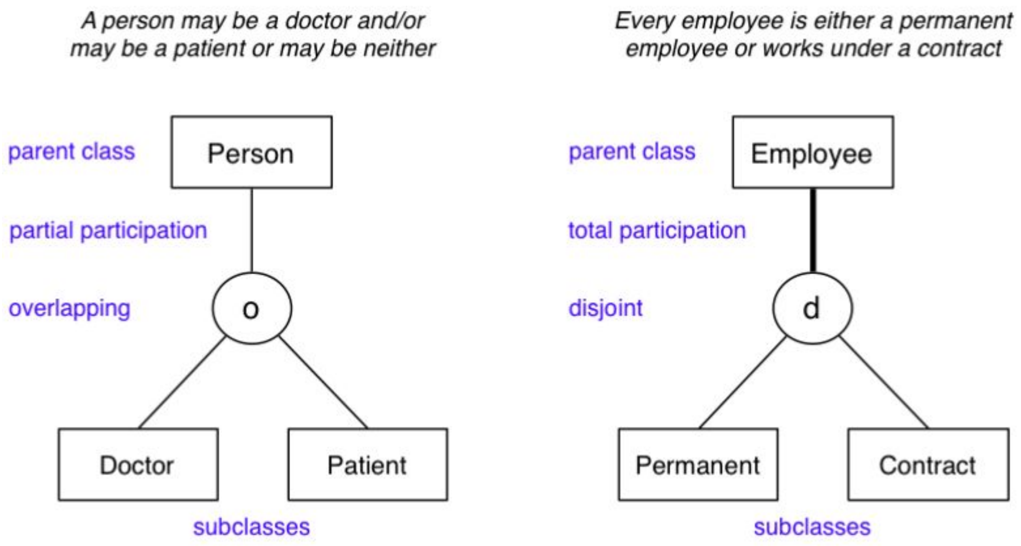
Subclasses and Inheritance

A subclass of an entity set A is a set of entities:

- with all attributes of A, plus (usually) its own attribute
- involved in all of A’s relationships, plus its own

Properties of subclasses:

- Overlapping or disjoint: Can an entity be in multiple subclasses
- Total or Partial: Does every entity have to also be in a subclass



A person may be a doctor and/or may be a patient or may be neither

Every employee is either a permanent employee or works under a contract

Week 9 Lecture

Relational Data Model

Relational Data Model: Describes the world as a collection of inter-connected relations (or tables)

Goal of Relational Model:

- A simple, general data modelling formalism
- Maps easily to file structures (easy to implement)

Two styles of terminology: (Mathematical or Data-Oriented)

Mathematical	Relation	Tuple	Attribute
Data-Oriented	Table	Record (row)	Field (Column)

The relational model has one structuring mechanism

- a relation corresponds to a mathematical "relation"
- a relation can also be viewed as a "table"

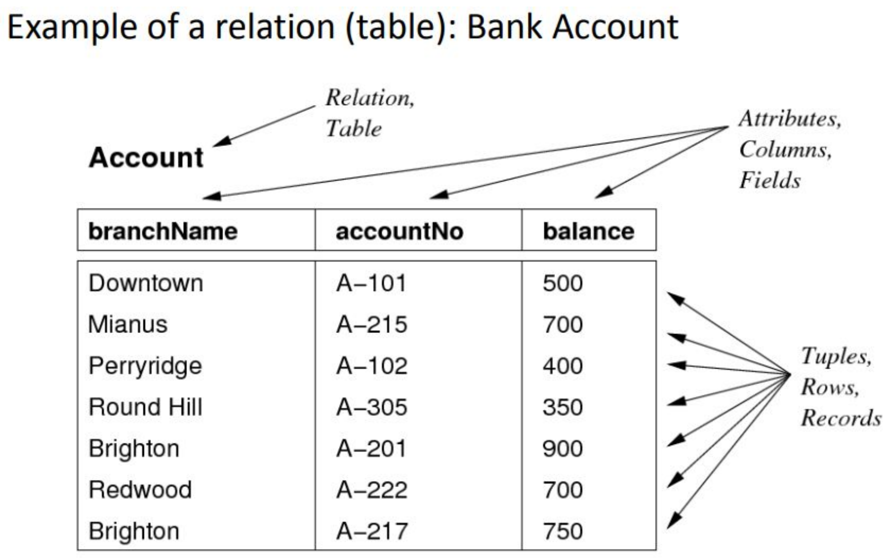
Each relation(table) [Denoted R,S,T…..] has:

- name, unique within a given database
- set of attributes (or column attributes)

Each attribute [Denoted A,B,...] has:

- name, unique within a given relation
- an associated domain (the set of allowed values)

DB definitions also make use of constraints.



Example of a relation (table): Bank Account

Account

branchName	accountNo	balance
Downtown	A–101	500
Mianus	A–215	700
Perryridge	A–102	400
Round Hill	A–305	350
Brighton	A–201	900
Redwood	A–222	700
Brighton	A–217	750

Hierarchy of Relational Model

Tuples(rows)

A set of values(attribute or column values)

Attribute values:

- Are atomic (no composite or multi-valued attributes)
- Belong to a domain; a domain has name, data type and format
- A distinguished value NULL belongs to all domains.
- NULL has several interpretations: none, don’t know, irrelevant

Relation(Table)

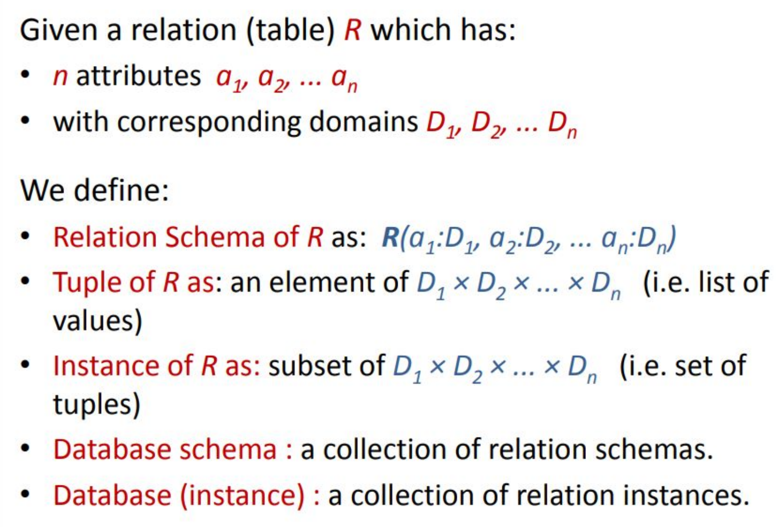
A set of tuples

- Since a relation is a set, there is no ordering on rows.
- Normally, we define a standard ordering on components of a tuple.
- Each relation generally has a primary key (subset of attributes, unique over relation)

Database

A set of relations(tables)

Mathematically:



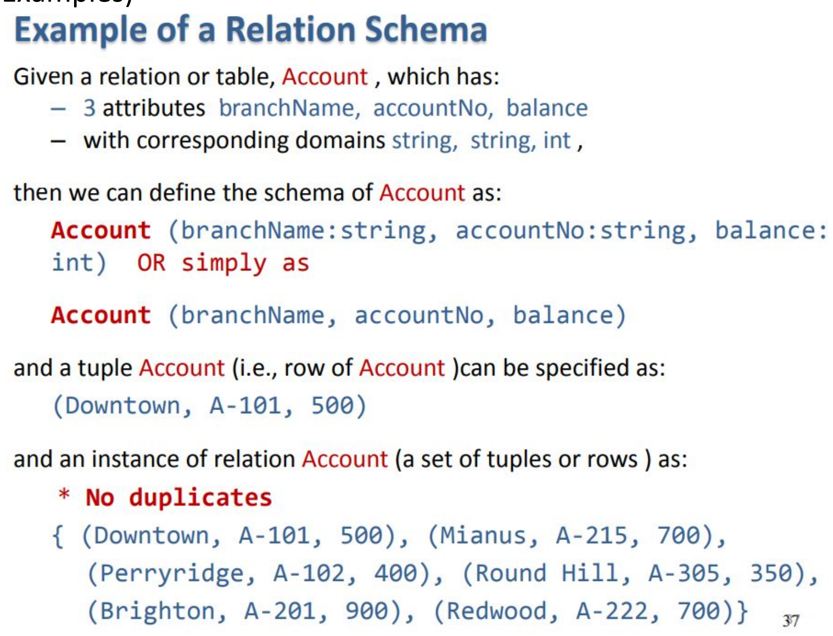
Given a relation (table) R which has:

- n attributes $a_1, a_2, \ldots a_n$
- with corresponding domains $D_1, D_2, \ldots D_n$

We define:

- Relation Schema of R as: $\boldsymbol{R}(a_1{:}D_1, a_2{:}D_2, \ldots a_n{:}D_n)$
- Tuple of R as: an element of $D_1 \times D_2 \times \ldots \times D_n$ (i.e. list of values)
- Instance of R as: subset of $D_1 \times D_2 \times \ldots \times D_n$ (i.e. set of tuples)
- Database schema : a collection of relation schemas.
- Database (instance) : a collection of relation instances.

Examples)



Example of a Relation Schema

Given a relation or table, Account , which has:

- 3 attributes branchName, accountNo, balance
- with corresponding domains string, string, int ,

then we can define the schema of Account as:

```
Account (branchName:string, accountNo:string, balance:
int)  OR simply as

Account (branchName, accountNo, balance)
```

and a tuple Account (i.e., row of Account)can be specified as:

```
(Downtown, A-101, 500)
```

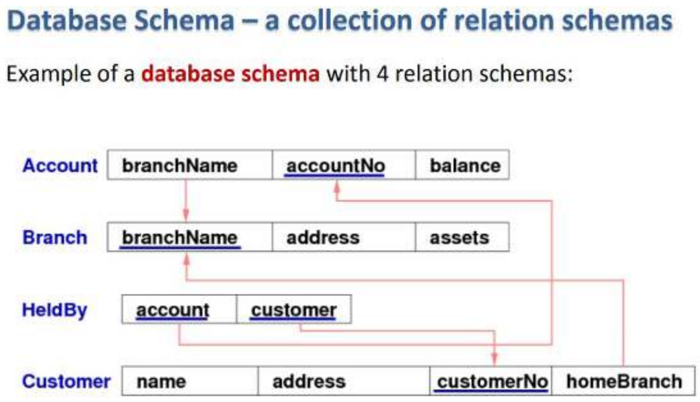
and an instance of relation Account (a set of tuples or rows) as:

```
 * No duplicates
{ (Downtown, A-101, 500), (Mianus, A-215, 700),
  (Perryridge, A-102, 400), (Round Hill, A-305, 350),
  (Brighton, A-201, 900), (Redwood, A-222, 700)}
```

Degree

The degree (or arity) of a relation: The number of attributes n of its relation schema.

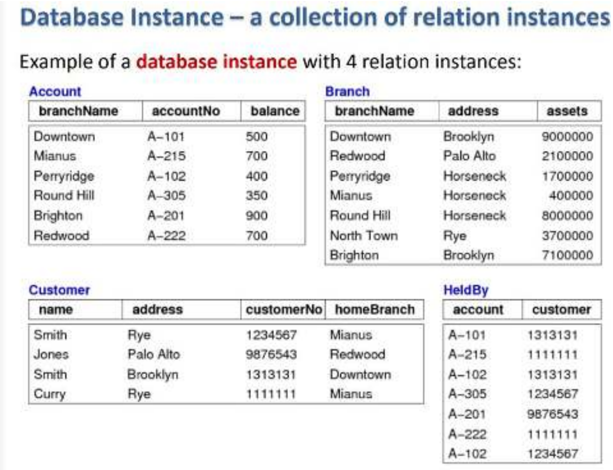
- E.g. The schema of “Account” is 3.



Database Schema – a collection of relation schemas

Example of a database schema with 4 relation schemas:

Account	branchName	accountNo	balance	
Branch	branchName	address	assets	
HeldBy	account	customer		
Customer	name	address	customerNo	homeBranch



Database Instance – a collection of relation instances

Example of a database instance with 4 relation instances:

Account

branchName	accountNo	balance
Downtown	A–101	500
Mianus	A–215	700
Perryridge	A–102	400
Round Hill	A–305	350
Brighton	A–201	900
Redwood	A–222	700

Branch

branchName	address	assets
Downtown	Brooklyn	9000000
Redwood	Palo Alto	2100000
Perryridge	Horseneck	1700000
Mianus	Horseneck	400000
Round Hill	Horseneck	8000000
North Town	Rye	3700000
Brighton	Brooklyn	7100000

Customer

name	address	customerNo	homeBranch
Smith	Rye	1234567	Mianus
Jones	Palo Alto	9876543	Redwood
Smith	Brooklyn	1313131	Downtown
Curry	Rye	1111111	Mianus

HeldBy

account	customer
A–101	1313131
A–215	1111111
A–102	1313131
A–305	1234567
A–201	9876543
A–222	1111111
A–102	1234567

Relational Modelling Constraints

Used to represent real-world problems.

We need to describe:

- What values are/are not allowed
- What combinations of values are/are not allowed.

Constraints are logical statements that:

- Key constraint
- Domain Constraint
- Referential Integrity

Key Constraints:

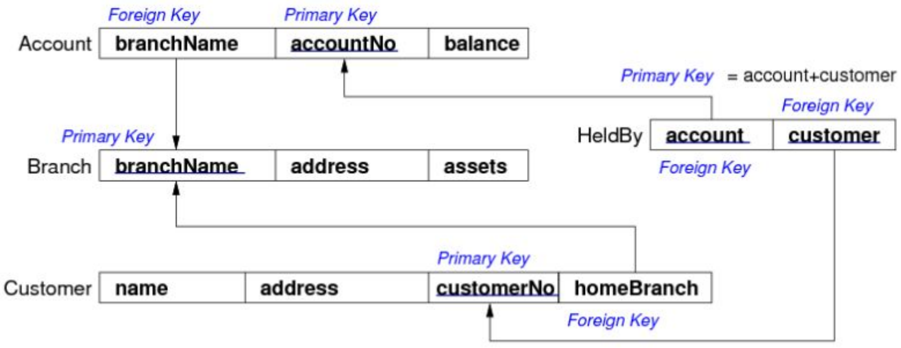
- A key attribute is one whose value can be used to uniquely identify each tuple (row) in the relation.
- A relation can have more than one key, so each key is a candidate key.
- **Entity Integrity Constraint:**
    - One of the candidate keys is designated as the primary key of the relation.
    - An entity integrity constraint states that no primary key value can be NULL.

Domain Constraints

- Specify that within each tuple, the value of each attribute must be an atomic value from the corresponding domain.

Referential Integrity Constraint

- Describe references between relations (tables)
- Are related to the notion of a foreign key (FK), where the primary key of the parent table is linked to a foreign key in the child table.
- Example)
    - The Account relation (child) needs to take note of the branch where each account is held.
    - The notion that the branchName (in Account) MUST REFER to a valid branchName (in Branch) is a referential integrity constraint

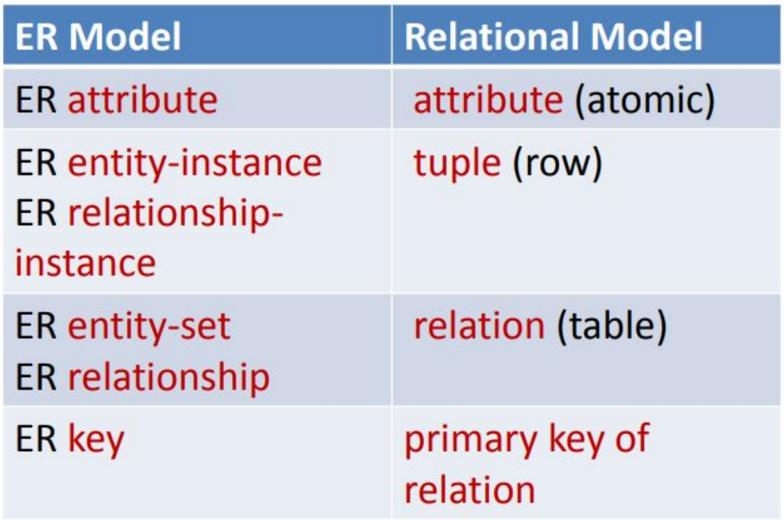


DBMS Terminology

DBMS-Level	Database names must be unique
Database-Level	Schema names must be unique
Schema-Level	Table names must be unique
Table-Level	Attribute names must be unique

Mapping ER Model to Relational

Correspondences:



ER Model	Relational Model
ER attribute	attribute (atomic)
ER entity-instance ER relationship-instance	tuple (row)
ER entity-set ER relationship	relation (table)
ER key	primary key of relation

Differences:

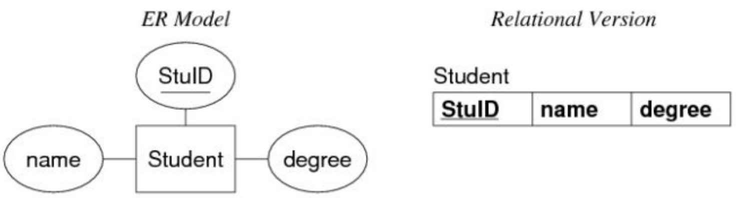
- The relational model uses relations to model entities and relationships.
- The relational model has no composite or multi-valued attributes.
- The relational model has no OOP notions (subclasses or inheritance)

Mapping Strong Entities

Easy as attributes in both are atomic.

ER Model	Relational Model
Entity Set E with atomic attributes {A,B,C...}	Relation(table) R with attributes(columns) {A, B,C...}

Example:



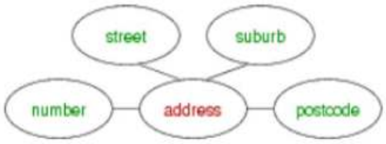
(Note: the key is preserved in the mapping)

Mapping Composite Attributes

Harder as Relational Modeling does not support composite attributes.

Consider Composite attributes as:

- Structuring attributes (non-leaf attributes)
- Data attributes (leaf attributes, containing atomic values)



Approach	ER Model	Relational Model
Approach 1: Remove Structuring Attributes, such that:	Addr{number, street, suburb, pcode}	Maps to: (AddrNumber, AddrStreet, AddrSuburb, AddrPcode)
Approach 2: Concatenate atomic attribute values into a string, such that:	name{First name, Last Name}	Maps to: (First Name + Last Name)

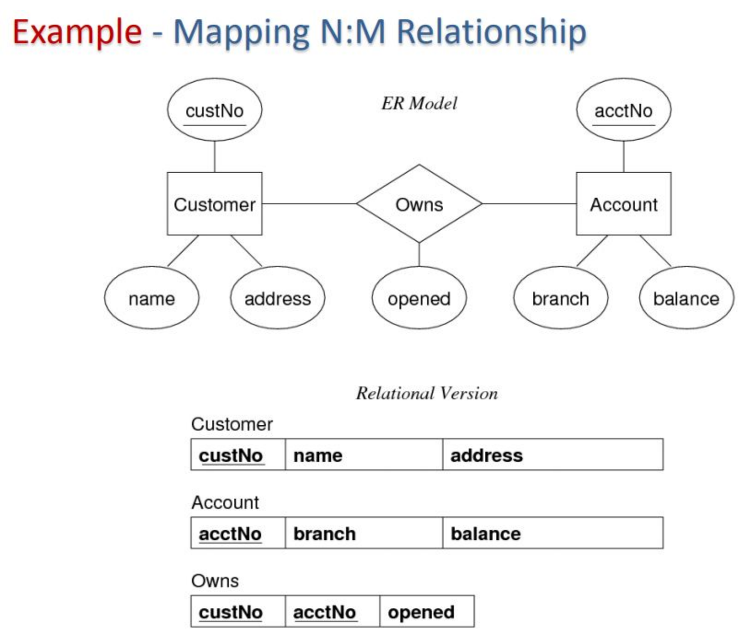
Mapping Relationships

ER relationship -> relational table(relation)

- Identifying one entity as “parent” and other entity as “child”.
- General Rule: “Primary Key of Parent is added to child as Foreign Key”
- Any attributes of the relationship are added to CHILD relation

Mapping Relationships

- Entities can be mapped using normal attribute mapping.
- Relationships can be mapped by making the key for the relationship in the ER model the union of the key attributes for S and T.



Example - Mapping N:M Relationship

Relational Version

Customer

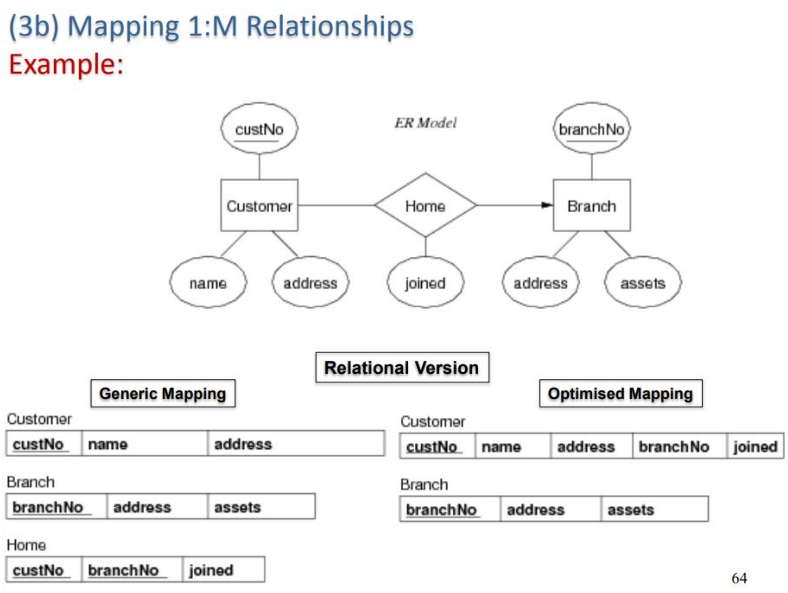
custNo	name	address

Account

acctNo	branch	balance

Owns

custNo	acctNo	opened



(3b) Mapping 1:M Relationships

Example:

Relational Version

Generic Mapping

Customer

custNo	name	address

Branch

branchNo	address	assets

Home

custNo	branchNo	joined

Optimised Mapping

Customer

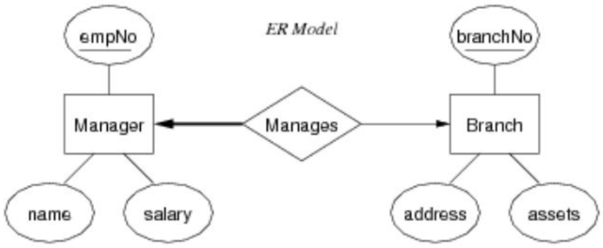
custNo	name	address	branchNo	joined

Branch

branchNo	address	assets

(3c) Mapping 1:1 Relationships

Example:

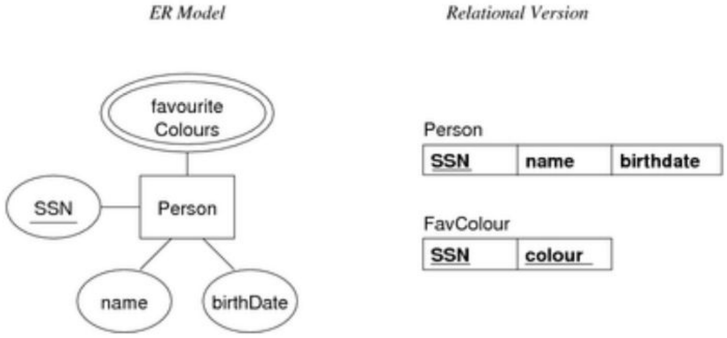


- Handled similarly to 1:N relationships
- For a 1:1 relationship between entity sets *E* and *F* (*S* and *T*):
  - choose one of *S* and *T* (e.g. *S*) (*Note : Choose the entity set that participates totally, if only one of them does)*
  - add the attributes of *T*'s primary key to *S* as foreign key
  - add the relationship attributes as attributes of *S*

(4) Mapping multi-valued attributes

- treat like an N:M relationship between entities and values
- create a new relation where each tuple contains:
  - the primary key attributes from the entity
  - one value for the multi-valued attribute from the corresponding entity

Example:



Person

SSN	name	birthdate

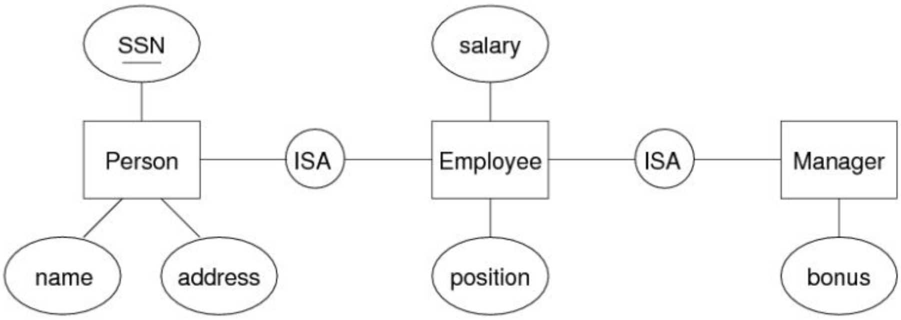
FavColour

SSN	colour

Mapping Subclasses

Consider the ER Diagram:

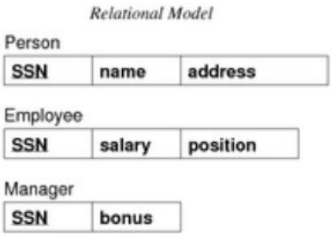
Entity–Relationship Model



Three Different Approaches to mapping the inheritance:

ER Style:

- Each entity becomes a separate table containing attributes of subclass + FK to superclass table.



Relational Model

Person

SSN	name	address

Employee

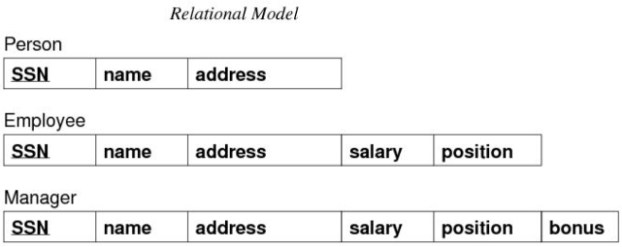
SSN	salary	position

Manager

SSN	bonus

Object-Oriented

- Each entity becomes a separate table, inheriting all attributes from all superclasses.



Relational Model

Person

SSN	name	address

Employee

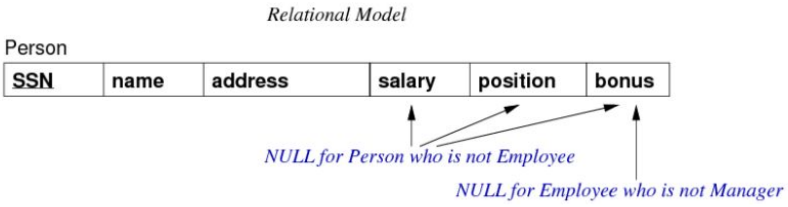
SSN	name	address	salary	position

Manager

SSN	name	address	salary	position	bonus

Single table with NULLS

- Whole class hierarchy becomes one table containing all attributes of all subclasses (null, if unused)



Relational Model

Person

SSN	name	address	salary	position	bonus

NULL for Person who is not Employee

NULL for Employee who is not Manager

Object Relational Mapper (ORMs)

A high-level abstraction framework that maps a relational database system to objects.
Automates all the CRUD(create/retrieve/update/delete) operations
ORM is agnostic to which relational database is used.

Advantages/Usefulness:

- Shields the developer from having to write complex SQL statements and focus on the application logic using their choice of programming language.
- Harmonisation of data types between the OO language and the SQL database.
- Automates transfer of data stored in relational database tables into objects.

Week 10 Lectures

Software Architecture

Simply it is:

- The macroscopic organisation of the system built.
- Partition the system in logical sub-systems or parts, then provide a high level view of the system in terms of these parts and how they relate to form the whole system.

Architecture focuses on non-functional requirements (“cross cutting concerns”) and decomposition of functional requirements, whilst design is focused on implementing functional requirements.

Why do we Decompose (break down) systems:

- Understanding and communication
- Tackle complexity by “divide-and-conquer”
- To prepare for extension of the system (separation of concerns)
- Focus on creative parts and avoid “reinventing the wheel” (find new and used modules).

Software Architectural Style:
A family of systems in terms of a pattern of structural organisation
A style is defined by:

- Components: A collection of computational units or elements that “do the work”
    - E.g. Classes, Databases, tools, processes
- Connectors: Enable communication between different components of the system and uses a specific protocol.
    - E.g. Function call, remote procedure call, event broadcast
- Constraints: Define how the components can be combined to form the system
    - Define where data may flow in and out of the components/connectors
    - Topological constraints that define the arrangement of the components and connectors

Software Architecture Styles:

Client-Server:

- Basic architectural style for distributed computing:
- Two distinct component types:
  - Server: Provides specific services. (E.g. Database or File Server)
  - Client: Initiates connections to request services provided by a server.
- Connecter is based on a request-response model.
  - E.g. File Server, Database Server, Email Server

What problems does this solve?
How to share data between a client and a service provider distributed geographically across different locations.

Client-Server Pro's and Con's:

Pros:	Cons:
Makes effective use of network and distribution of data is straightforward.	Single point of failure- when server is down, client requests cannot be met
Roles and responsibility of system distributed among several independent machines.	Network congestion, when large number of client requests are made simultaneously to one server
Easy to add new servers or upgrade existing server.s	Complex and expensive infrastructure
Deployment of modules to different servers enhancing security, scalability and performance.	

Peer-to-Peer (P2P)

- Each peer component can function as both a server and a client.
- Information distributed among all peers and any two components can set up a communication channel to exchange information as needed.

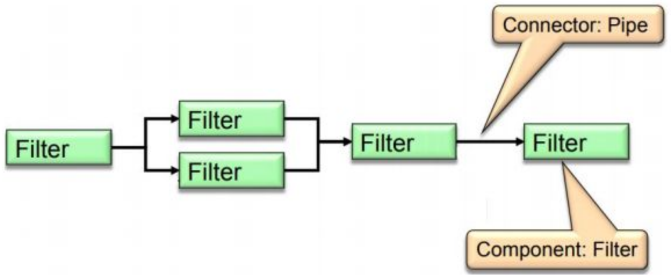
What problems does P2P Solve?
Network congestion and single point of failure that arise from a client server model.

P2P Pro's and Con's:

Pros:	Cons:
Efficiency: More efficient as all clients provide resources.	Architectural complexity
Scalability: Unlike client-server, capacity of the network increases with number of clients.	Distributed resources are not always available.
Robustness: Data replicated over peers Immune to single point of failure	More demanding of peers.

Pipe and Filter:

- Component: Filter, this reads input streams. Each filter encapsulates a data processing step to transform data and produce out streams.
- Connector: Pipe, data transformed by a filter is sent to other filters for further processing using a pipe.
- Features:
  - Data is processed incrementally as it arrives.
  - Output can begin before input is fully consumed
  - Filters must be independent:
    - Filters do not share state
    - Filters do not know upstream or downstream filters



Pipe-and-Filter Pro’s and Con’s:

Pros:	Cons:
Easy to understand the overall input/output behaviour of a system. It is a simple composition of filters.	Highly dependent on order of filters.
Flexible: Filters can be replaced and modified. Filters can be combined.	Not appropriate for interactive applications. Data stream only flows in one directions
Support concurrent processing of data streams.	

Repository Style:

- Components: Two distinct types:
  - Central data repo: It is central, reliable and permanent.
  - Data accessors: A collection of independent components that operate on central data; components do not interact directly but only through the central repository
- Connectors:
  - Read/Write mechanism
  - Components access repo as and when they want

Repo Pro's and Con's:

Pros:	Cons:
Efficient way to share large amounts of data.	All independent components must agree upon a repository data model.
Centralised management of the repository. ● Concurrency ● Security ● Backup	Distribution of data can be a problem
Components need not be concerned with how data is produced.	Connectors implement a complex infrastructure.

Deciding on Styles:

- Architectural decisions often involve compromise
- The best design for a system completely depends on:
  - What defines "goodness" of a design
  - System requirements
  - Business priorities
  - Other factors such as: Available resources, target customers, technology trends, compatibility, potential future requirements.

Software Architectural Views

A projection of a model showing a subset of its details.

Master Model: Has all the details.

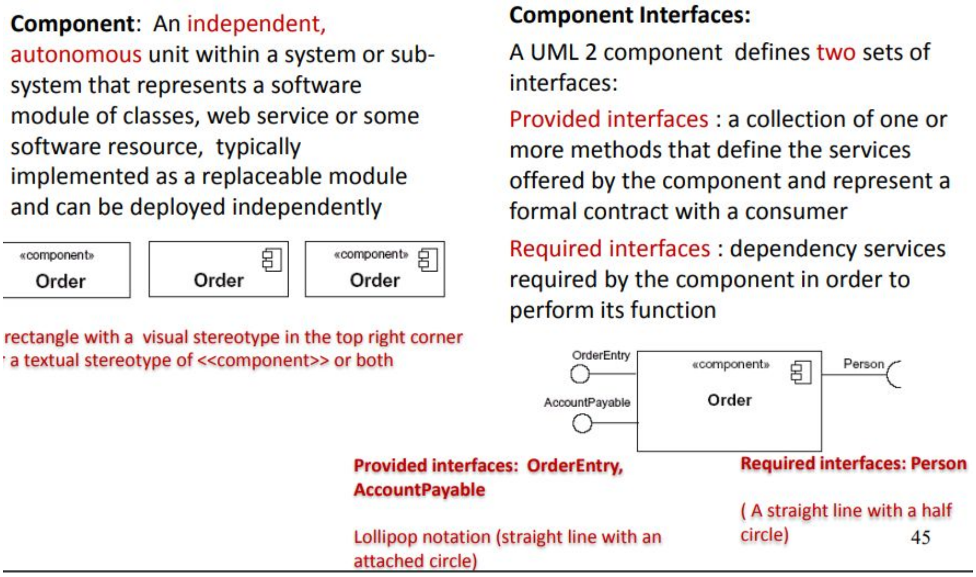
- Views are projects of this model, the subset of information.

Model View:
Decomposes the functionality into a coherent set of software (code) units.

- Logical break-down into sub-system often shown using package diagrams.
- Interaction among components at run-time shown using sequence diagrams, activity diagrams.
- Data shared between low level components and collaborations typically expressed as UML class diagrams.

Architecture View (Component and Connector View):

- Describes a runtime structure of the system, including: Components, data stores, connectors.
- Box and line diagram (informal), UML component diagram (formal).



Component: An independent, autonomous unit within a system or sub-system that represents a software module of classes, web service or some software resource, typically implemented as a replaceable module and can be deployed independently

rectangle with a visual stereotype in the top right corner
a textual stereotype of <<component>> or both

Component Interfaces:

A UML 2 component defines two sets of interfaces:

Provided interfaces : a collection of one or more methods that define the services offered by the component and represent a formal contract with a consumer

Required interfaces : dependency services required by the component in order to perform its function

Provided interfaces: OrderEntry, AccountPayable

Lollipop notation (straight line with an attached circle)

Required interfaces: Person

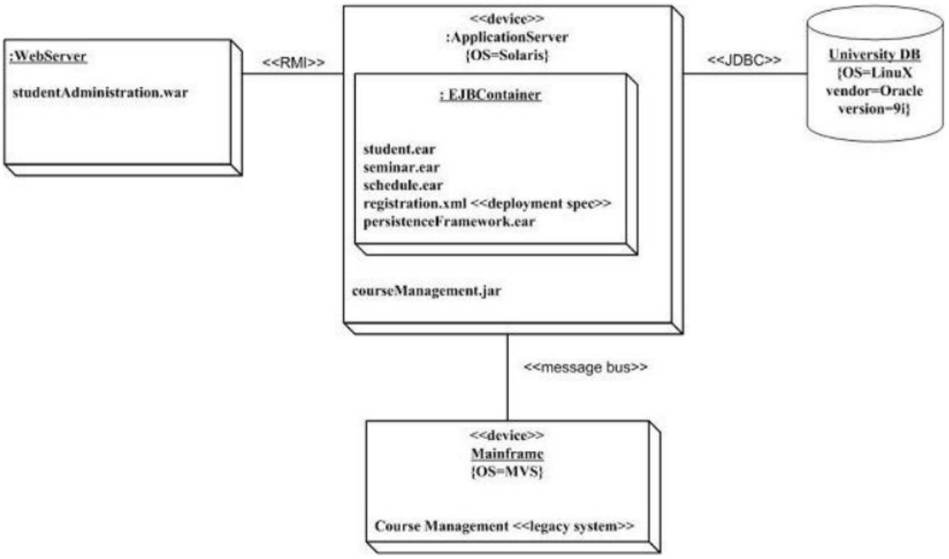
(A straight line with a half circle)

45

Allocation View:

- Describes how the software units map to the environment (hardware resources, file-systems and people).
- Exposes properties like which process runs on which processors
- Typically expressed as a UML deployment diagram.
  - A static view of the run-time configuration of the processing nodes and the components that run on these nodes.
  - Show the hardware for your system
  - Ideal for applications deployed to several machines.

Example of a UML 2 deployment diagram


:WebServer
studentAdministration.war
<<RMI>>
<<device>>
:ApplicationServer
{OS=Solaris}
: EJBContainer
student.ear
seminar.ear
schedule.ear
registration.xml <<deployment spec>>
persistenceFramework.ear
courseManagement.jar
<<JDBC>>
University DB
{OS=LinuX
vendor=Oracle
version=9i}
<<message bus>>
<<device>>
Mainframe
{OS=MVS}
Course Management <<legacy system>>