**1.1 Software Development Life Cycle**



* 1. Requirements Analysis
  + Analyse and understand problem domain
  + Determine function and non-functional components
  + Generate user stories/ use case

2. Design

* + Producing software architecture/blue-prints
  + System diagrams and schematics
  + Modelling of data flows

3. Development

4. Testing

* + Use unit or behaviour tests to test your software

5. Deployment

* + “Ship it”

6. Maintenance

* + Monitor
* DevOps
  + Modern philosophy of blending the development and operations teams
  + Merge “system IT” and “developers”

**2.1 Requirements**

* Definition: A condition or capability needed by a user to solve a problem or achieve an objective
* Other definition:
  + Agreement of work to be completed by all stakeholders
  + Descriptions of constraints of a proposed system
* Functional v Non-Functional
  + Functional requirements specify a specific capability/service that the system should provide (eg. Messenger should be able to send message)
  + Non-Functional requirements place a constraint on how the system can achieve that. Typically this is a performance characteristic.
* What matters?
  + Investigate stakeholder needs
  + Expand, refine, and connect specific ideas
  + Understand the iterative and ongoing nature
* User stories
  + It is a method of requirements engineering used to inform the development process and what features to build
  + As a <type of user>, I want <some goal> so that <some reason>
  + Keep customers at the centre
  + Keep it problem focused, not solution focused

**2.2 Agile**

* Principles behind the agile manifesto

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software
2. Welcome changing requirements, even late in the development. Agile processes harness change for the customer’s competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project
5. Build projects around motivated individuals.
6. …

* Summary
  + Iterative and incremental
  + Quick turnover
  + Light on documentation
* Agile practices
  + Standups
    - Answer 3 key questions
      * What did I do?
      * What problems did I face?
      * What am I going to do?
  + Asynchronous Stand-ups
    - * No need to find a suitable time for everyone
      * May work better for big team
      * Disadvantage:
        + “Blockers” take longer to be addressed
        + Easy to forget to give an update
        + Less persona updates from others can be missed
  + Sprints/ Iterations
    - Time is fixed, scope is flexible
    - Plan only for the next sprint
    - Typically have a release at the end of each sprint
  + Task board
  + Pair programming
  + Test Driven Development (TDD)
    - Writing tests before the implementation
    - Write only enough code to make the next test pass

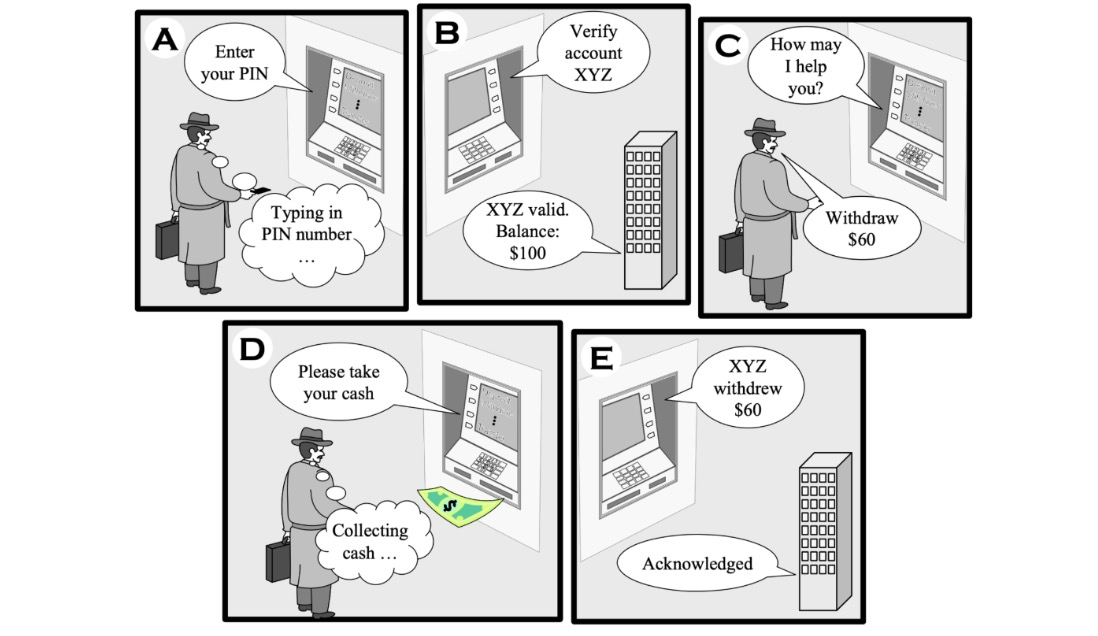
**3.2 User Acceptance**

* Good tests
  + INVEST
    - Independent
    - Negotiable: avoid too much details
    - Valuable: must hold some value to the client
    - Estimable
    - Small: user story should be small
    - Testable
* User Acceptance Criteria
  + Break down a user story into criteria that must be met for the user, or customer, to accept
  + Written in natural language
  + Can be refined before implementation
  + It should not be too broad
  + They should be written before development

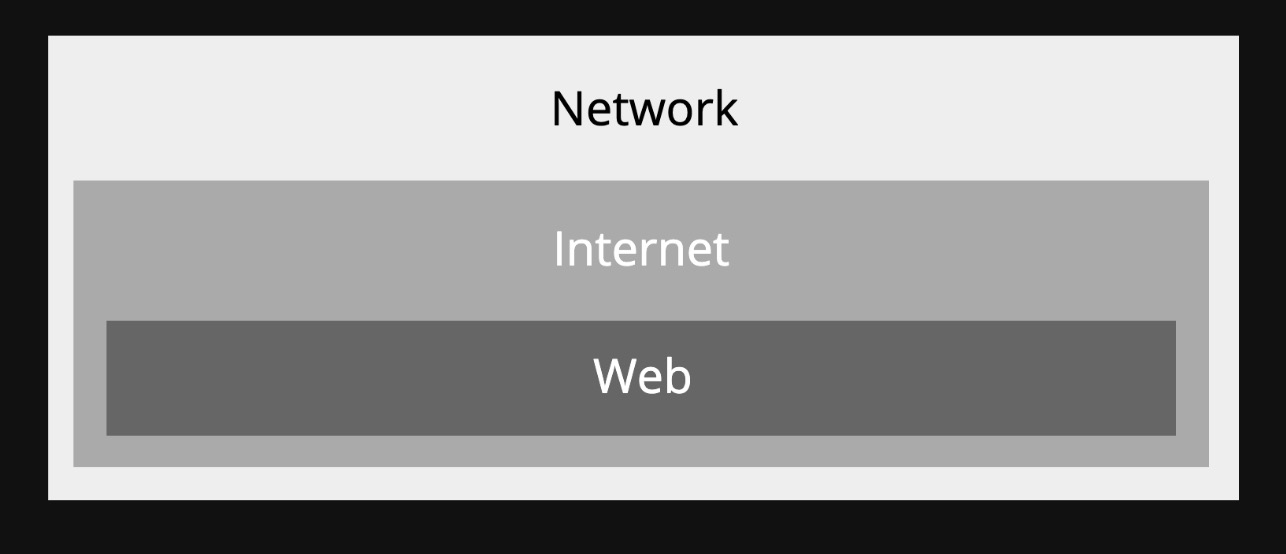
**3.3 Verification & Validation**

* Unit testing
  + Testing of individual software components
  + White-box
  + Tested by software engineers
* Integration testing
  + Testing interactions between integrated components or system.
  + White-box or black- box
  + By software engineers or independent testers
* System testing
  + Testing an integrated system to verify that it meets specified requirements
  + Normally independent testers
* Acceptance Testing
  + Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.
  + Black box
  + User or Customer
* Coverage
  + python3-coverage run --source=. -m pytest
  + python3-coverage report
  + python3-coverage html
* Branch coverage checking

**3.4 Use Cases**

* Represent a dialogue between the user and the systemk with the aim of helping the user achieve a business foal
* User initiates action and the system responds with reactions
* They consider the system a black box
* 
* Representations
  + Informal list of steps
  + Use-case diagram
  + **Cockburn** style

**4.1 HTTP, Flask**

****

* THE NETWORK
  + **Network**: A group of interconnected computers that can communicate (eg. LAN)
  + **Internet**: A global infrastructure for networking computers around the entire world together
  + **World Wide Web**: A system of documents and resources linked together, accessible via URLs.
* HTTP (Hypertext transfer protocol)
  + Protocol for sending and receiving HTML documents
* Talking to Flask
  + 3 Ways to do it:

1. cURL (client URL)

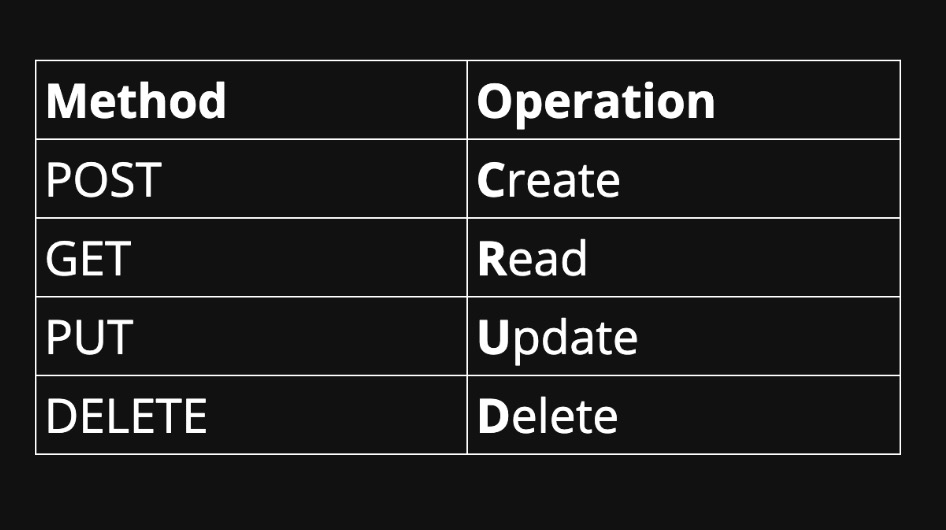
* Use it to get html details

1. API client

* Postman etc.

1. Web Browser

* Restful API & “CRUD”
  + A RESTful API is an application program interface (API) that uses HTTP requests to GET, PUT, POST and DELETE data

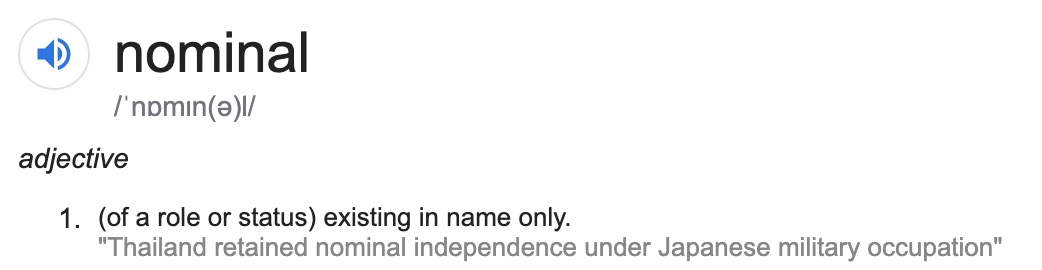


**4.2 Auth, State**

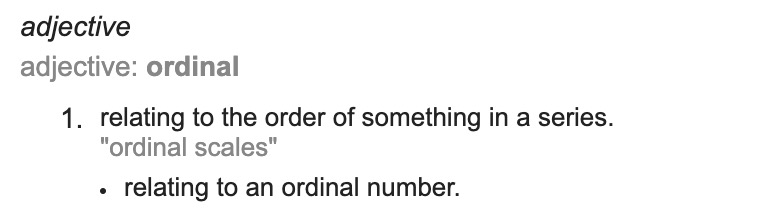
* Authentication V Authorisation:
  + Authentication: the process of verifying the identity of a user
  + Authorisation: process of verifying an identity’s access privileges
    - Token: a packet of data used to authorise the user
    - Issue with passing around a raw user\_id: Authentication can be faked

**5.2 Measure & Estimation**

* Measurement is a process by which numbers or symbols are assigned to properties of objects
* Software Measurement
  + Cost or effort to produce the software
  + Code coverage
  + Code quality
  + Software complexity
* Measurement Scales
  + Nominal



* + - Group subjects into different categories,
    - Number has no meaningful order
    - Eg. weather can be cloudy/sunny, gender: male = 1; female = 2
    - Two key properties
      * Mutually exclusive: measured attribute falls into one and only one category
      * Jointly exhaustive: categories cover all possible values of the attribute
  + Ordinal



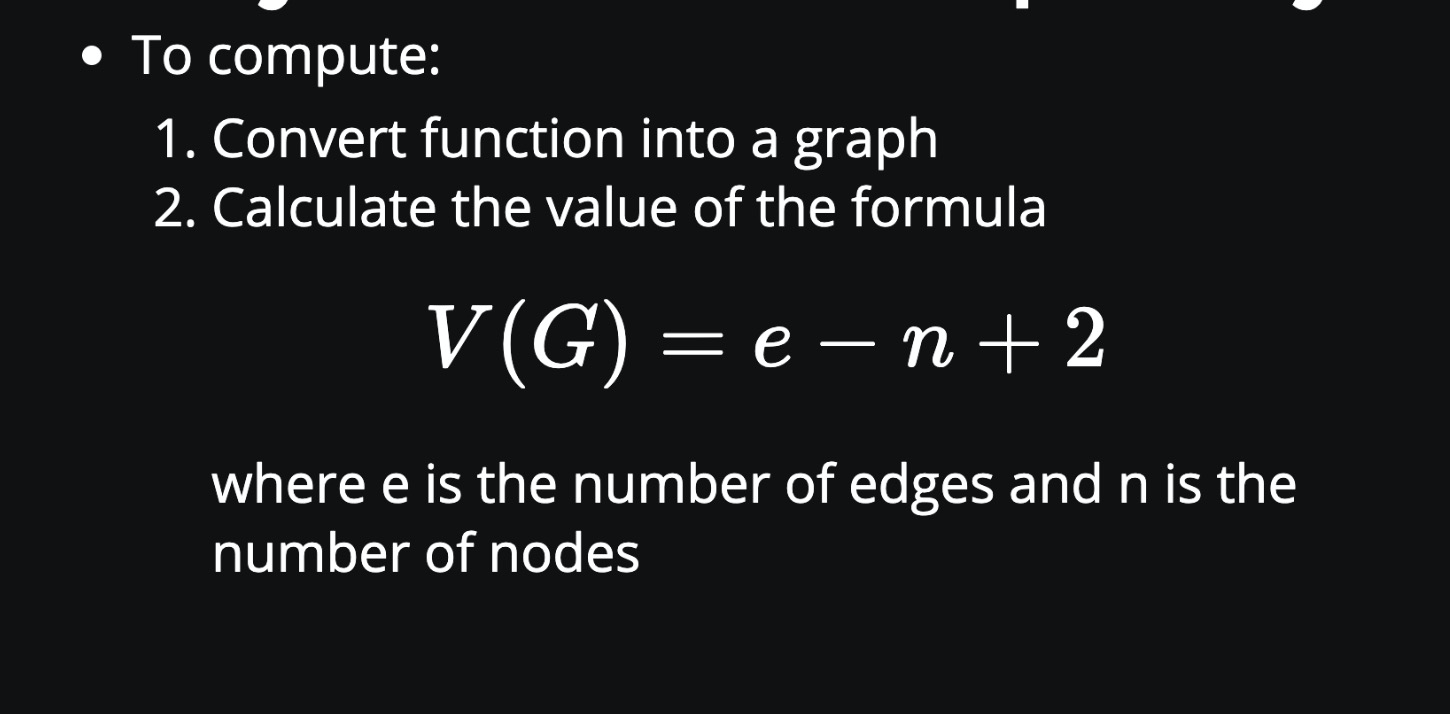
* + - Subjects can be compared in order
    - Number has meaningful meaning
    - Eg. star ratings, place when finish a race
    - Key properties:
      * **Asymmetry**: if A > B is true then B > A is false
      * **Transitivity**: if A > B and B > C then A > C
  + Interval
    - Exact differences between measurement points
    - Eg. 30 C is 10 hotter than 20 C
    - Only addition and subtraction can be applied
    - No true zero point
  + Ratio
    - Has an absolute or non-arbitrary zero point (absence of that attribute)
    - Eg. Length in cm
    - Can have addition, subtraction, multiplication and division to them
    - Have relative ratios:
      * 20cm is twice as long as 10cm
  + Absolute
    - Only one way to measure a property
    - Eg. How many chairs are in this room?
* Scales we have encountered:
  + Code coverage: **Ratio** (% code covered **compared to** % code written)
  + Pylint: **Ordinal**

**Note: Ordinal vs Interval**. Interval has fixed difference. Eg. 50 deg and 49 is 1 deg difference and 20 deg and 19 deg is also the same. Ordinal is relative difference. Eg. I finish 1st in the race and John finishes 2nd, although the position is 1 difference but the time may not be. Ethan who finishes 3rd it not 2\* difference compared to me.

* Code coverage
  + Ratio scale
  + Determined by how much of the codebase is executed
  + Tends to correlate with test coverage

**5.3 Software Complexity**

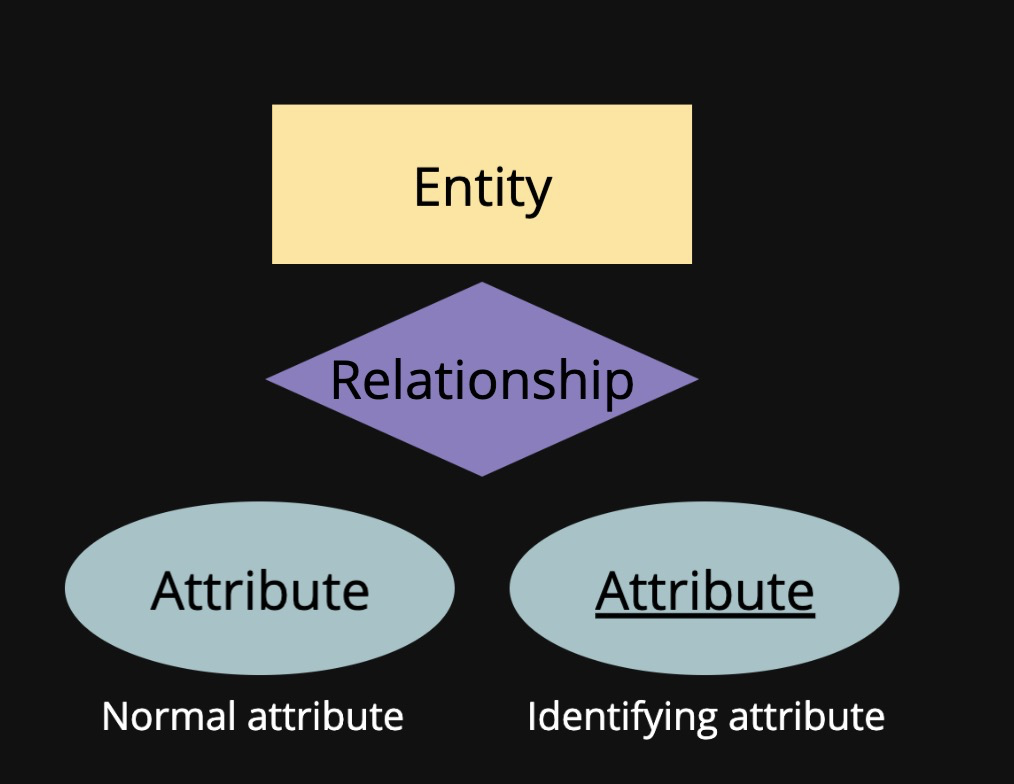
* No Silver Bullet:
  + Describe software complexity by dividing it into two categories: **essential** and **accidental**
  + Essential: Complexity that is inherent to the problem. For example, if the client requires the program to do 30 different things, then those 30 things are essential
  + Accidental: Complexity that is not inherent to the problem. For example,k generating ro parsing data in specific formats.
* Measuring complexity
  + Coupling
    - A measure of how closely connected different software components are.
    - Eg. frontend are usually loosely coupled from the backend
  + Cohesion
    - The degree to which elements of a module belong together
    - Either low or high
  + Cyclomatic complexity
    - An **interval measure** of branching complexity of functions
    - Computed by counting the number of linearly-independent paths through a function.



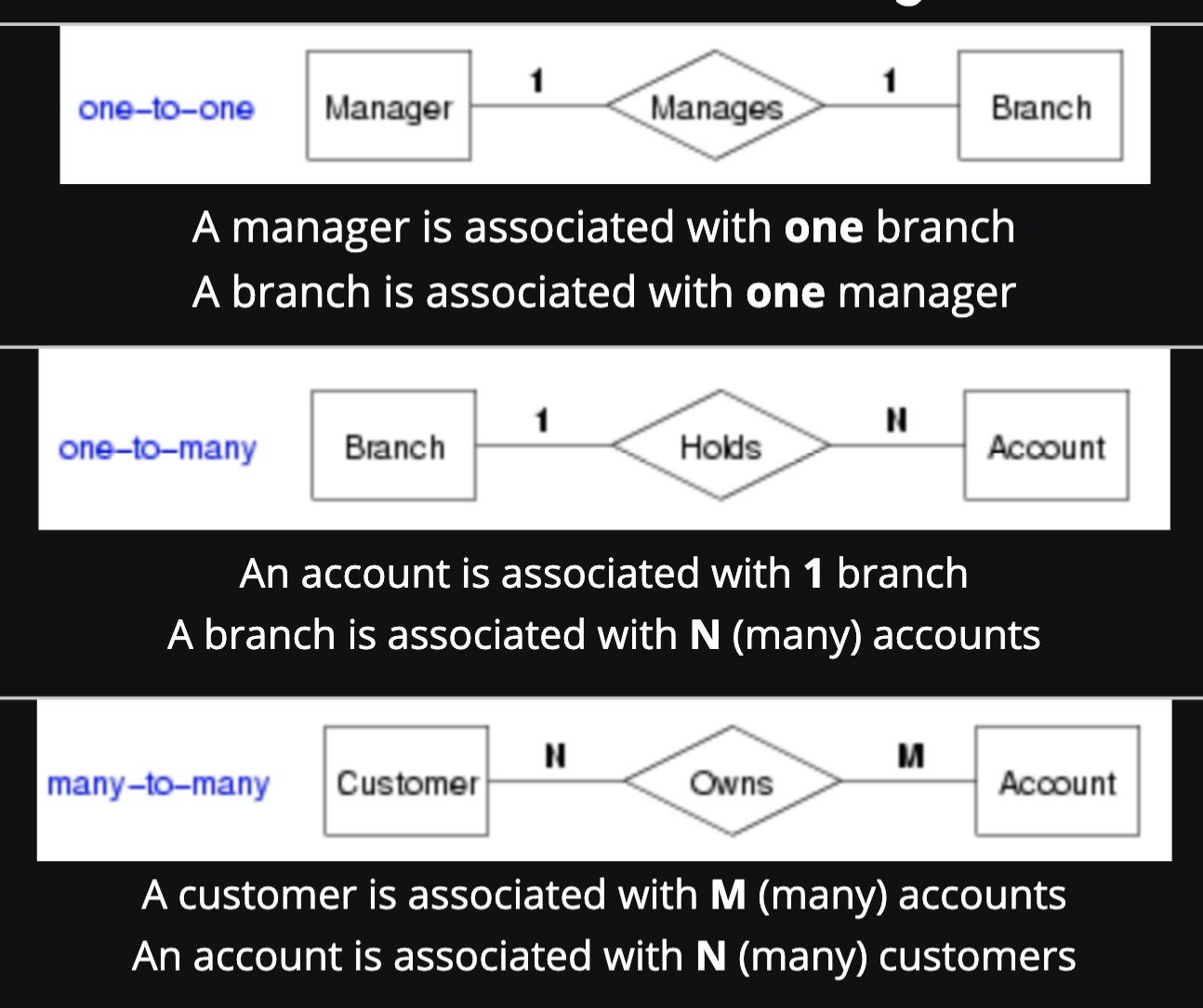
* Usage: a simple understandable measure of function complexity.
* Drawback:
  + assumes non-branching statements have no complexity
  + Keeping cyclomatic complexity low encourages splitting function s up, regardless of whether that really make the code more understandable.

**6.1 Data**

* Data storage
  + Persistence
    - When program state outlives the process that created it. This is achieved by storing the state as data in computer data storage.
* Data transfer
* Data modelling
  + ER Diagrams



* ER Diagram Multiplicity (info about relationship)
  + =Cardinality
    - Denoted with ‘1’ or ‘n/m’
  + Participation
    - A requirement to participate
    - Denoted with a bold line
  + Multiplicity = cardinality + participation



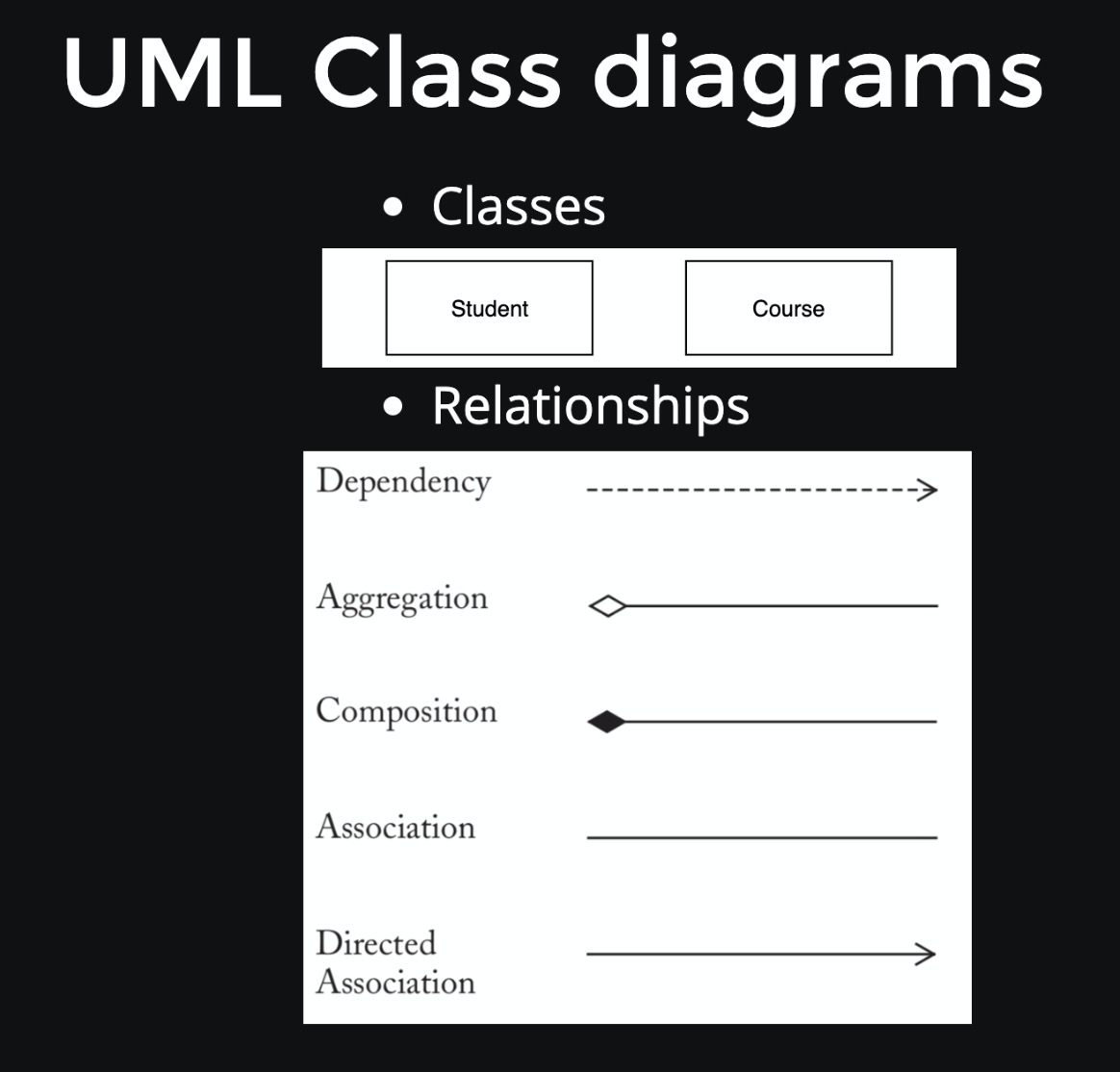
**7.1 Software Engineering Design Principles**

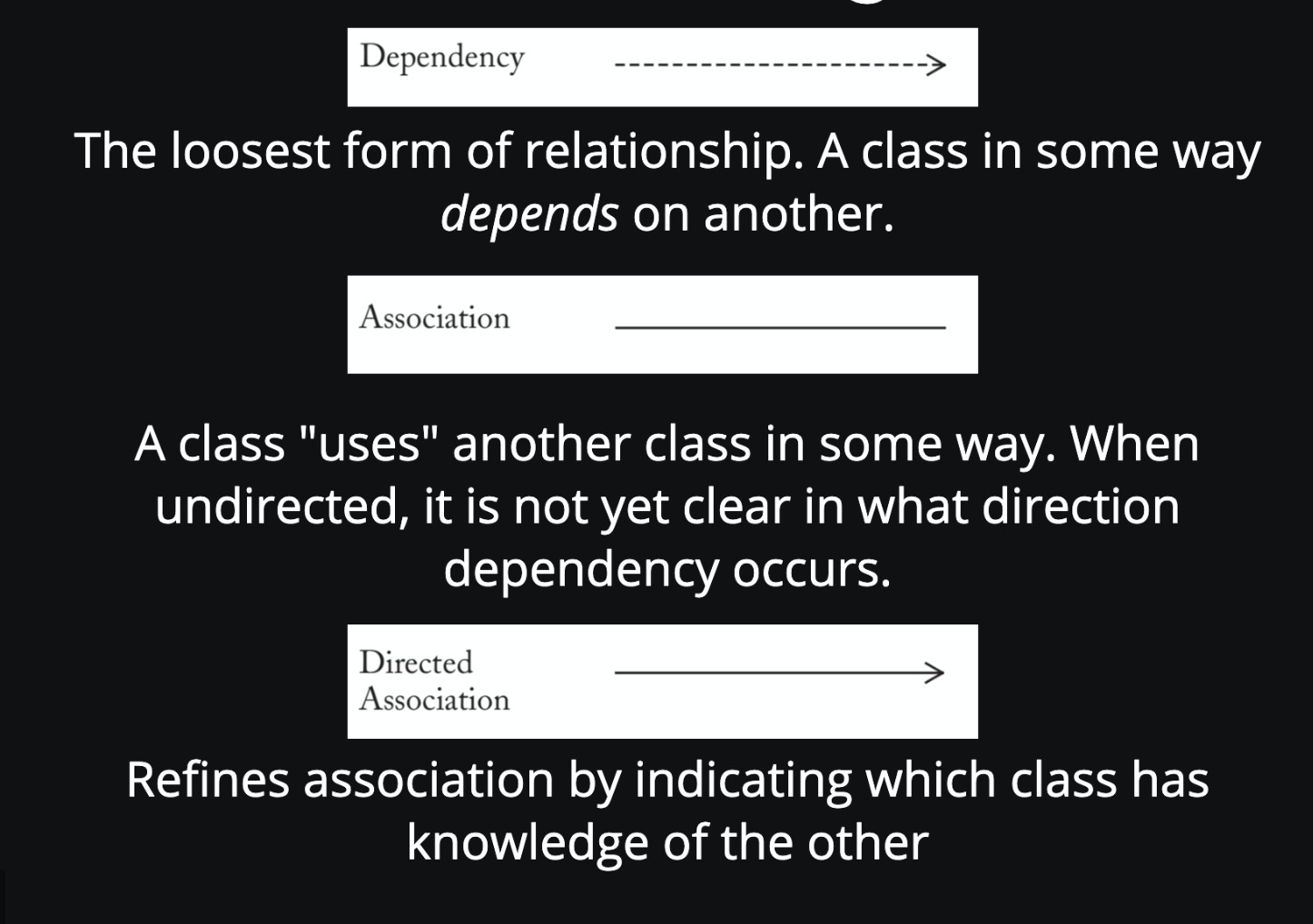
* Design Smells
  + Rigidity: difficult to change
  + Fragility: tend to break when making simple change
  + Immobility: previous work is hard to reuse
  + Viscosity: changes feel slow to implement
  + Opacity: difficult to understand
  + Needless complexity
  + Needless repetition
  + Coupling
* Design principles
  + Purpose:
    - Extensible
    - Reusable
    - Maintainable
    - Understandable
    - Testable
  + Achieve through abstraction
  + DRY - Don't repeat yourself
  + KISS - Keep it Simple, Stupid
* Encapsulation: maintaining type abstraction by restring direct access to internal representation of types (types include classes)
* Top-down thinking
  + Discourage YAGNI(You aren't gonna need it)

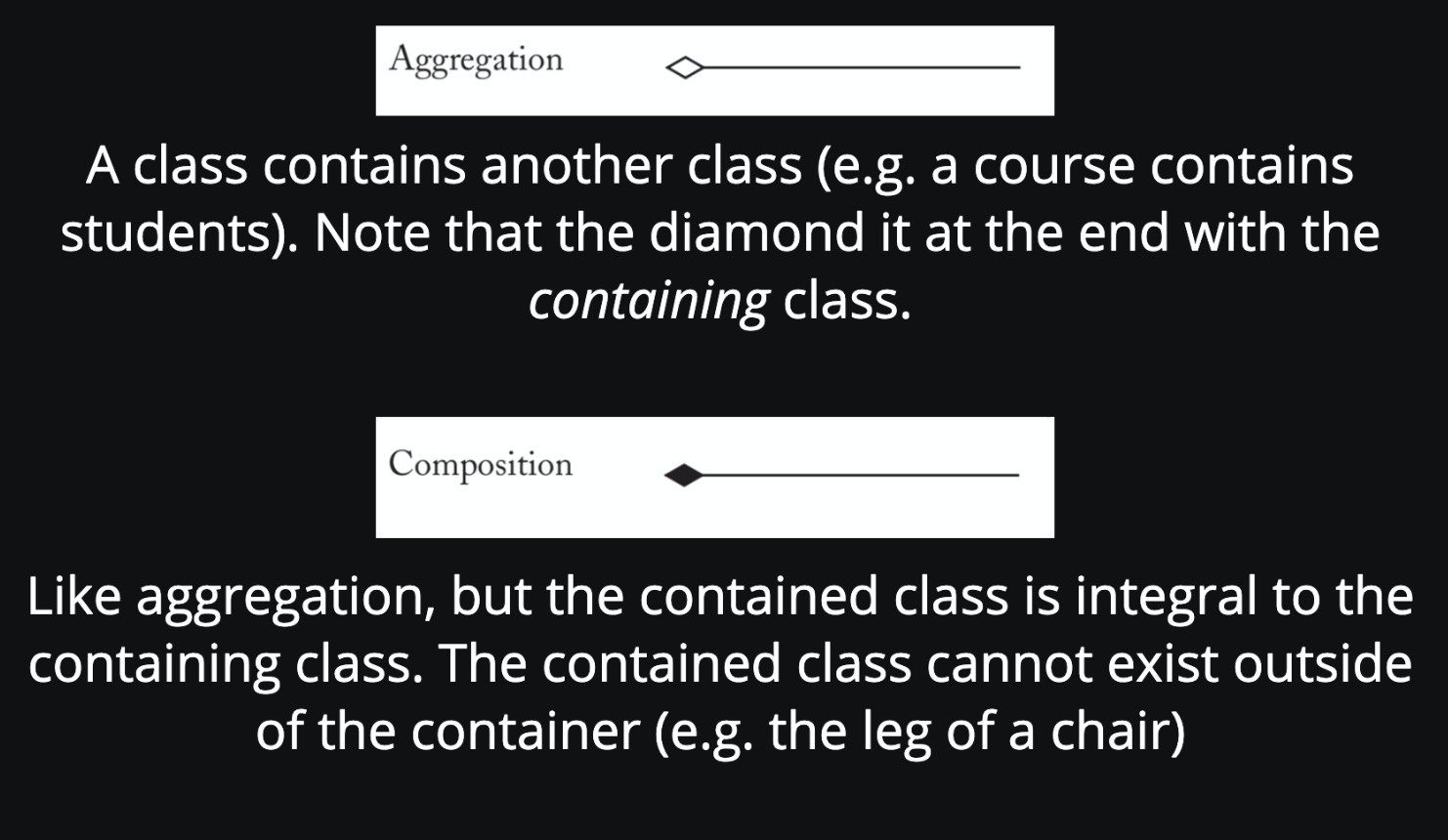
**7.3 Concept Modelling**

Why are models used?

* To convey the fundamental principles and basic functionality of systems.
* Domain Modelling
  + A conceptual model of a domain that incorporates both **data** and **behaviour**
  + Domain: A sphere of knowledge particular to the problem being solved
  + Ubiquitous language: use language that represent real things in the domain expert’s mental model

****

****

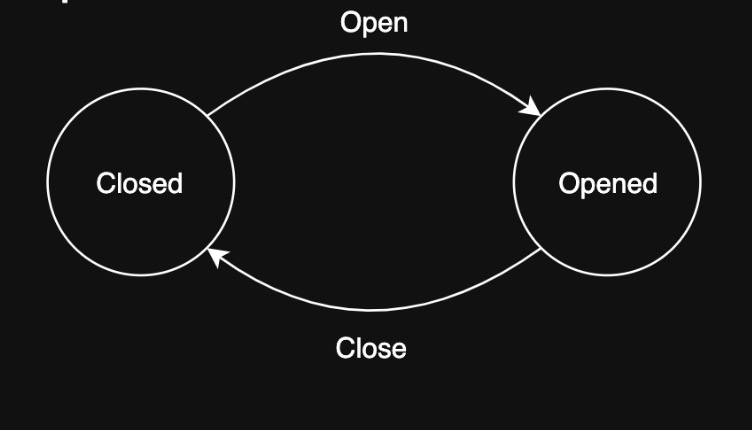


**8.1 Software Engineering Principles (2)**

* Single Responsibility Principle
* Every module/function/class in a program should have **responsibility** for just a **single** piece of that program’s functionality.
* So it is more readable and modular

**8.3 - Further Modelling**

* System Modelling
  + Structural - Emphasise the static structure of the system
    - UML class diagrams
    - ER diagrams
    - Etc
  + Behavioural - Emphasise the dynamic behaviour
    - State diagrams
    - Etc
  + State Machines
    - Useful for modelling system that have clearly defined states.
    - Eg. ULs with different screens, Network protocols, conversational interfaces



**8.4 - Property-based testing**

A method of testing where tests are defined as general properties

* Test input is generated automatically by supplying a strategy for generating that input
* In the event of a test failure, the framework will shrink the generated input to find the smallest value that still fails the test
* Hypothesis
  + Tests are decorated with @given to supply strategies for generating test input
* What properties to test?

**9.1 Development part 1**

* Continuous Integration
  + Practice of automating the integration of code changes from multiple contributors into a single software project
  + Process

1. Write tests

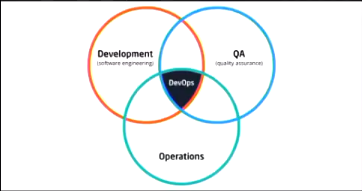
* + - Tests for each story
    - Broad tests: unit, integration, acceptance, UI tests

2. Us code coverage checkers

3. Merge and integrate code as often as possible

4. Ensure the build always work

* Continuous Delivery
  + Allows accepted code changes to be deployed to customers quickly and sustainability. This involves the **automation of the release process**, with a **manual trigger** step by humans.
  + Different deployments
    - Dev:
      * Released often, available to developers to see their changes in deployment
      * Used for developers internally
    - Test:
      * As close to release as possible, ideally identical to prod
      * Similar to PTR
    - Prod:
      * Released to customers, ideally as quickly as possible
      * Live server
  + Are sometimes called “shipping”
  + Benefits:
    - Sustainable
    - Faster
* Flighting
  + Moving builds out to particular slices of users, beyond the simplicity of dev, test and prod
  + Eg. Nightly builds
* Continuous Deployment
  + Extension of continuous delivery whereby changes **attempt to flight toward production automatically** and the only thing stopping them is a failed tests
* DevOps
  + A role dedicated to gluing the 3 pillars of deploying quality assured software
  + Operation & Development & Quality Assurance



**9.2 - Deployment, Maintenance**

Maintenance: After deployment, the use of analytics and monitoring tools to ensure that as the platform is used and remains in a healthy state.

* Purpose:
  + Preserving user experience
    - Monitoring errors, warnings, and other issues that affect performance or uptime
    - Enhancing user experience: Using analytical tools to monitor users or understanding their interactions. Often leads to customer interviews and user stories

**10.1 Design**

* Software design
  + Understandability
  + Encapsulation
  + Maintainability
  + Extensibility
  + Reusability
  + Testability
  + Reliability
  + Performance
  + Usability
  + Safety
  + Security
* Libraries
  + Most code reuse is through libraries.
  + Downside:
    - Author remove the library
    - Author breaking the library with an update
    - Author being malicious
    - Infrastructure that provides the library not being available
    - Depends on other library
* Pinning
  + Good for applications
  + Doesn’t work for libraries because we want them to be reusable
* Versioning
  + Semantic versioning
    - MAJOR.MINOR.PATCH

10.3 Safety

* Safety vs security
  + Safety = protection from accidental misuse
  + Security = protection from deliberate misuse
* Static VS Dynamic
  + Static: static prop can be inferred without executing the code
  + Dynamic: dynamic prop are checked during execution (eg. python)
* Memory safety
  + Protecting from bugs relating to memory access
  + Trade off between performance and safety
* Handling runtime errors
  + EAFP = easier to ask for forgiveness than permission
    - Pros:
      * Can simplify core logic
      * Multiple different sorts of errors can be handled with one except block
    - Cons:
      * Make code non-structured
      * Harder to reason what code will be executed
  + LBYL = Look Before you Leap
    - Encourages you to check that something can be done before you do it
    - Pros:
      * Doesn’t require exceptions
      * Code is structured and therefore easier to reason about
    - Cons:
      * Core logic can be obscured by error checks
  + Removing errors statically

Exam

* 30% multiple choice
* 20% short answer
* 50% programming
  + Similar to lab exercises