created by Syed Abdullah

CS2103/T: Software Engineering

Locating Stuff in Handout

- **UML Sequence Diagram** – [L4P1] (pp 3 – 10)
- **UML Object Diagram** – [L3P1] (pg 6)
- **UML Use Case Diagram** – [L5P2] (pg 9)
- **UML Activity Diagram** – [L10P1] (pp 1 – 4)

Theorems/Principles/Whatever

- **Single Responsibility Principle** [L3P1, 8]
  "A class should have one, and only one, reason to change."
  Meaning: If a class has only one responsibility, it needs to change only when there is a change to that responsibility.
- **Interface Segregation Principle** [L4P1, 12]
  "No client should be forced to depend on methods it does not use."
  Meaning: Define parameters/variables according to what is to be used (i.e. if subclass methods not used, variable should be superclass)
- **Liskov Substitution Principle** [L5P1, 7]
  "If a program module is using a super class, then the reference to the super class can be replaced with a sub class without affecting the functionality of the program module."
  NOTE: Litmus test for subclasses to check behaviour conforms to interface/abstract/parent
- **Open-Closed Principle** [L6P1, 2]
  "A module should be open for extension but closed for modification"
  Example: Cmds in addressbook & Java Generics
- **Dependency Inversion Principle (DIP)** [L6P1, 3]
  "High-level modules should not depend on low-level modules. Both should depend on abstractions."
- **Law of Demeter** [L9P3, 1]
  - An object should have limited knowledge of another object.
  - An object should only interact with objects that are closely related to it.

Testing

Refer to L1P3

- **Scripted testing** (or proactive) – predetermined set of test cases used. Systematic
- **Exploratory testing** (reactive, error guessing, attack-based and bug hunting) – creating test cases based on results of previous test cases

Refer to L4P2

- **Test driver** – invokes the SUT with test inputs
- **Testability** – indication of ease of testing SUT

Refer to L8P2

- **Validation** – building the right system
- **Verification** – building the system right
- **Unit Testing** – test whether individual units (methods, classes, subsystems, ...) works correct **in isolation**
  - **Stubs/Mocks** – dummy component that receives outgoing messages from the SUT
  - **Dependency injection** – 'inject' objects to replace dependencies with other objects
- **Integration testing** – test whether different parts of the software works together
- **System testing** – test **whole system** against **system specifications** (only based on **specified external behaviour of system**)
  - **Performance testing** – responsiveness
  - **Load testing** – heavy load
  - **Security testing** – security
  - **Compatibility/interop testing** – system can work with other systems
  - **Usability testing** – how easy to use system
  - **Portability testing** – system works on different platforms
- **Acceptance Testing** – validation test to determine if system meets **requirements of customer**
- **Alpha Testing** – performed by users under conditions set by developers
- **Beta Testing** – performed on select subset of target users of system in natural work settings
- **GUI Testing**
- **Testing Coverage** – extent to which testing exercises code
  - **Function/method coverage** – no. of functions executed
  - **Statement coverage** – no of LoC executed
  - **Decision/Branch coverage** – based on **decision points**
  - **Condition coverage** – based on boolean **subexpressions**
  - **Path coverage** – based on possible paths through given part code executed
  - **Entry/Exit** – based on possible calls to and exits from operations

Refer to L8P3

- **Black-box** – test cases designed based on specified external behaviour
- **White-box** – test cases designed based on implementation (code)
- **Gray-box** – mix of black and white box
- **Effective & Efficient** – finds high % of existing bugs and with high rate of success (bugs found over test cases)
- **Equivalence partitioning** – dividing inputs into groups of possible inputs
- **Boundary Value Analysis** – values around the boundary of equivalence partition (e.g. one value from the boundary, one value just below the boundary, and one value just above the boundary)

Architecture

Refer to L7P1

- **Software architecture** – very high-level design
- **Architectural Styles**
  - **n-tier** – higher layers make use of lower layer services (lower layer independent)
  - **Client-Server** – obvious
  - **Transaction Processing** – workload of system is broken down to *transactions*, that are passed to dispatcher, which passes to executor

 License: Creative Commons BY-NC-SA 4.0 International License.

- **Service-oriented** – combine functionalities packaged as programmatically accessible service
- **Event-driven** – events are listened and acted

- **API** – contract between component and the user (of the API). Well-designed and documented.
- **Top down** vs **Bottom up** – determines if design starting from high or low level
- **Agile** vs **Full-design-up-front** – whether to design for the short or long term

Refer to L9P2

- **Abstraction occurrence pattern** – abstraction class holds common information and unique info kept in occurrence class
- **Singleton pattern** – restricts number of instantiated objects of class to just one (by making constructor private)
- **Facade pattern** – hides/encapsulates internal details using a class that external calls go through
- **Command pattern**
- **Model-View-Controller pattern** – reduce coupling from information-based systems
- **Observer pattern** – communication without coupling (using a class that listens)

Improve & Maintain Code Quality

Refer to L2P1

- **Understandability** – easy for others to work on: Follow coding standards, Name well, Be obvious, No misuse of syntax, Avoid error-prone practices, Minimize global vars, Avoid magic numbers, Throw out garbage (code), Minimize duplication, Comment minimally, but sufficiently – explain 'why' and 'what', Be simple, Code for humans, Single Level of Abstraction Per Method (SLAP)

Refer to L3P2

- **Exception** – deal with 'unusual' but not entirely unexpected situations (i.e. situations where program may not execute as intended)

Refer to L6P1

- **Coupling** – degree of dependence between components, classes, methods, etc. (E.g. Husband & Wife vs Store Attendant & Customer)
- **Cohesion** – measure of relatedness & focus of various responsibilities of a component.

Refer to L7P2

- **Assertions** – confirm assumptions about the program state (if assertion is false, code has a bug)
- **Logging** – troubleshoot problems that cannot be prevented by exceptions and assertions
- **Defensive programming** – pro-actively eliminate any room for things to go wrong
  - **1-to-1 association** – ensure that both objects can be created
  - **Referential integrity** – ensure consistency in references
  - **Design-by-contract** – honour operation if caller meets required preconditions

Refer to L9P3

- **Separation of Concerns** – separation of features, so that each module is mutually exclusive (can be applied in any layer)

Requirements & Project Management

Refer to L7P3

- **Big-bang integration** – components are integrated at the same time (not recommended)
- **Incremental integration** – Top-down, Bottom-up or Sandwich
- **Build automation** – automating the process
- **Dependency management** – manage third party libraries, keep up to date
- **Continuous Integration** – integration, building and testing happens automatically after code change

Refer to L5P2

- **Requirements gathering** – Brainstorming, User Surveys, Observation, Interviews, Focus Groups, Prototyping
- **Textual descriptions** – quick overview of domain/system that is understandable to both users and developers
- **Feature list** – grouped by criteria such as priority (e.g. must-have, nice-to-have), order of delivery, object or process related.
- **User stories** – As a ⟨use type/role⟩ I can ⟨function⟩ so that ⟨benefit⟩
- **Use case** – describes an interaction between the user and the system. Specify the following
  - Software System
  - Use Case
  - Actors (excluding system)
  - Preconditions
  - Guarantees
  - Main Success Scenario (MSS)
  - Extensions

Object Oriented Programming

Refer to L3P1

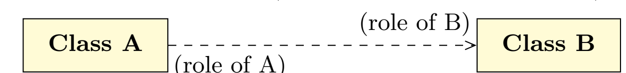
Class name
Attributes (visibility) name : type = default-value
Methods visibility name (params) : return-type

- **Associations** – two objects collaborate with one another

Can be represented as:

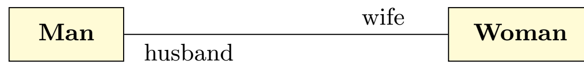
- Attributes – either containing a single object or an array
- Classes – covered below

- **Dependency** – two objects have temporary contracts (e.g. using constant)

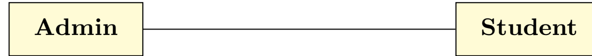


License: Creative Commons BY-NC-SA 4.0 International License.

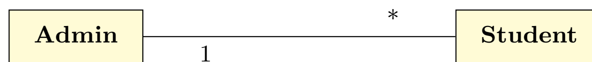
- **Role labels** – indicates the role played by the classes in the relationship



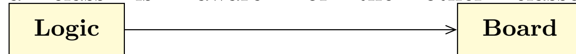
- **Association label** – describes the association and direction



- **Multiplicity** – How many objects of A is associated with ONE object of B
  - 0..1 – optional, can be linked to 0 or 1 objects
  - 1 – compulsory, must be linked to one object at all times.
  - * – can be linked to 0 or more objects.
  - n..m – the number of linked objects must be n to m inclusive



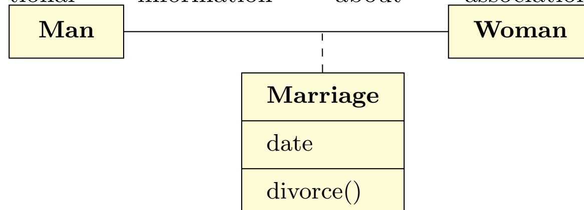
- **Navigability** – arrowhead indicating whether a class is "aware" of the other classes



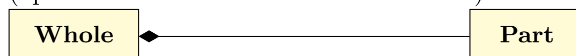
- **Object Diagrams** – like **Class Diagrams** but underlined names (of form `instance:class`)
- **Class-level members** (static) – attached to a class, not instance. **Denoted by underlining**.

<ins>**Refer to L4P1**</ins>

- **Association classes** – stores additional information about association



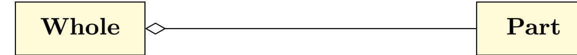
- **Composition** – "whole-part" relationship ("parts" existence is due to the "whole")



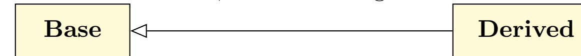
No cyclical links (in UML, assume none)

- **Aggregation** – "container-contained" ("contained" can exist without "container")

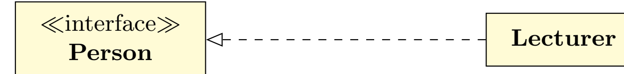
<ins>**Not recommended!**</ins>



- **Inheritance** – class that derive from a base class, inheriting its methods



- **Interfaces** – behaviour specification, no implementation



<ins>**Refer to L5P1**</ins>

- **Polymorphism** – provision of a single interface to entities (classes) of different types. Each object may behave differently for the same method!
  - **Operation overriding** – defined by programmer in code
  - **Dynamic binding** – resolution at runtime (e.g. using a superclass variable that holds a subclass)
- **Abstract Class** – behaviour specification + partial implementation

 License: Creative Commons BY-NC-SA 4.0 International License.