CS2103T : Requirements

functional & non-functional requirement

- functional requirements specify what the system should do.
- NFRs specify the constraints under which system is developed & operated

What are the char of a well-defined requirement

- unambiguous, Testable, Clear, correct, understandable, feasible, independent, Atomic, Necessary Abstract, consistent, Non-redundant, complete

Prioritizing requirements

- requirements can priortized based on importance & urgency, can be discarded if Out of scope

How to gather requirements

- Brain storming → Generate ideas not validate
- Pdt survey → studying existing Pdt can reveal shrtcomings of existing soln that the new product can address
- Observation → Observe users in their natural work environment can uncover pdt requirement
- user surveys → can solicit large amt of responses & opinions from stakeholders
- Interviews → Interviewing stakeholders & domain experts can produce info abt requirement
- focus grp → but in grp setting
- Prototyping → can uncover requirements w/ regards to how user interacts w/ the system

How to specify requirements

- Prose, feature list, User stories, Glossary

User stories

can be omitted if obvious

- format → As a {user type/role} I can {fn} so that {benefit}
- User stories can be written at various levels with smaller user stories under it
- condition or implementation to be marked as done can also be added

Use case

- System: EZ-Link machine (those found at MRTs)
- Use case: UC2 top-up EZ-Link card
- Actor: EZ-Link card user
- Preconditions: All hardware in working order. → specifies the specific state you expect the system to be in b4 the use case start
- Guarantees: MSS → the card will be topped-up.
- MSS: → main success scenario → specifies what the uc promise to give @ the end of the operation
  1. User places the card on the reader. ↳ nothing goes wrong
  2. System displays card details and prompts for desired action.
  3. User selects top-up.
  4. System requests for top-up details (amount, payment option, receipt required?).
  5. User enters details.
  6. System processes cash payment (UC02) or NETS payment (UC03).
  7. System updates the card value.
  8. System indicates transaction as completed.
  9. If requested in step 5, system prints receipt.
  10. User removes the card.

  Use case ends.
- Extensions: → used to describe alternative flows
  *a. User removed card or other hardware error detected.
  *a1. System indicates the transaction has been aborted.
  Use case ends.

use case can include other use case

steps describes externally visible behaviour not internal details
↳ interaction w/ user & systems

CS2103T : Modeling : Class & object diagram

What is modeling in SE context?

- Modeling in the SE context is used to represent a software design, it captures only a select aspect (abstraction) hence providing a simpler view of a complex entity

How is modeling used

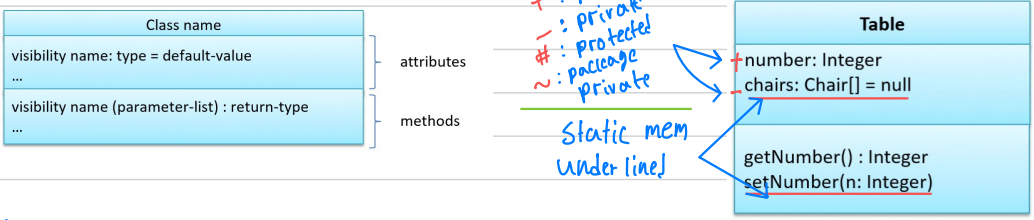
- 1) To analyse a cmplx entity 2) communicate visually 3) blueprint

How to model an OO soln

- recall that OO soln is basically a network of objects interacting w/ e.o → useful to model how such objects are 'networked' tgt → but such object structure can change over time ∴ Class structure used to illustrate rules that object structure has to follow

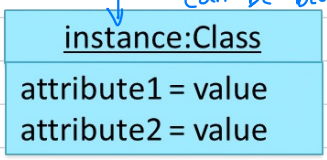
Class diagrams

basic structure:

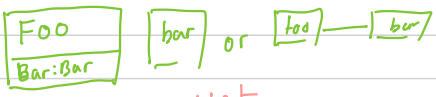


Object diagram

basic structure



can be blank to rep an unnamed object

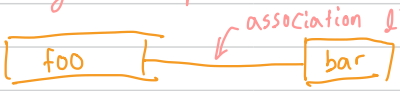


not

associations can also be represented as attributes but not both @ the same time

How do we rep the net work?

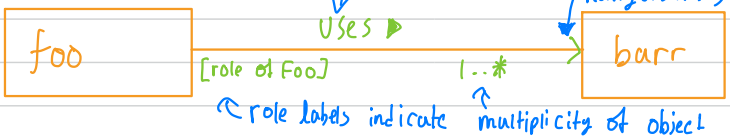
- associations are the main connections among the classes in a class diagrams / objects in an object diagram → they are represented as line between 2 diagrams



How do we give info about an ass

Labels describe meaning of association

navigability; can use in obj diag also



role labels indicate

multiplicity of object

How to add more inf model

We can add more info to a model using a note *note the small fold @ the btm right

Ho do we show diff types of r/s such as: Inheritance, composition, etc?

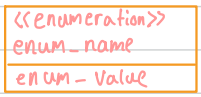
Composition (Whole-part) :

aggregation (container-contained) :

Inheritance :

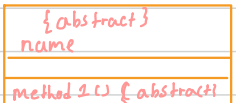
Dependencies : - - - →

Enumerations :

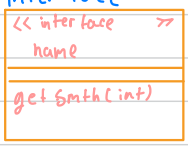


Inheritance (interface) : - - - ▷

abstract class

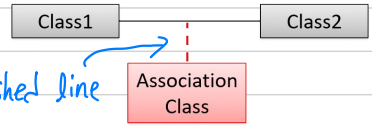


interface



* methods in interface must also appear in implementing class

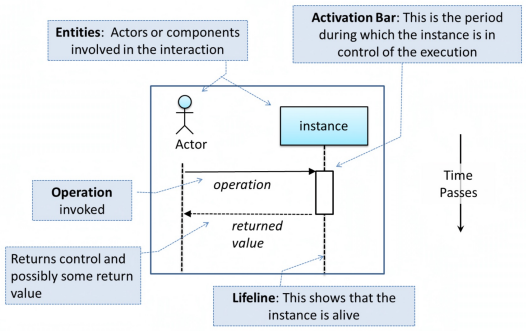
→ some times data is related to association rather than either class ∴ existance of association class



dashed line

CS2103T : Modeling : Sequence diagram

basic notation



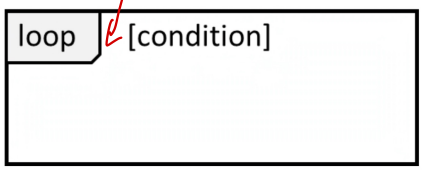
* some common mistakes
  1) activation bar too long
  2) broken activation bar

* minimal notation
  activation bar & return arrows may be omitted

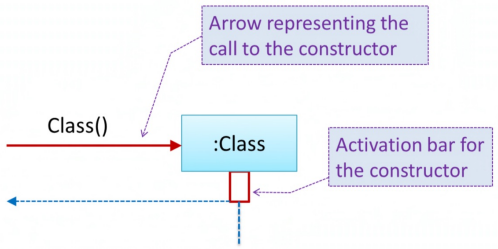
other notations

Loops

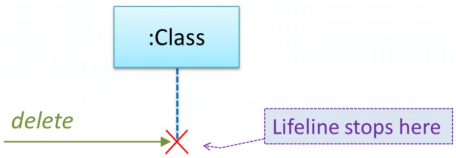
* be aware of the slant



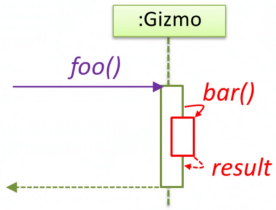
Object creation



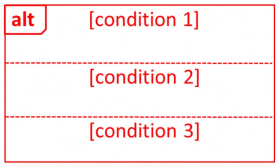
object deletion



Self invocation



alt frames

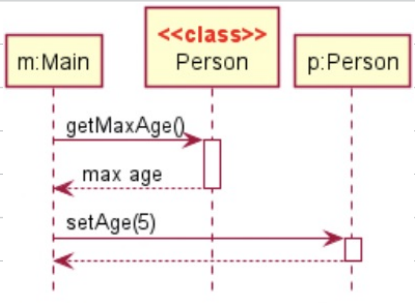


* if-else statement

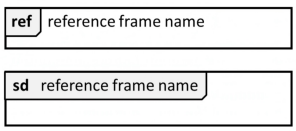
optional path

Calling a static method

→ add a «class» to show that participant is the class itself



reference frames



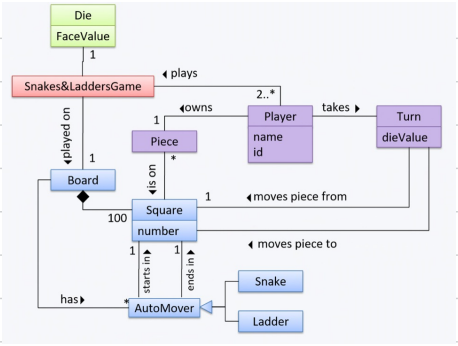
→ use to break complicated code to parts or omit details

parallel paths

CS2103T: Modeling : OODM & activity diagrams

What are OODMs

- OODM are class diagrams that are used to model the problem domain
  - ↳ similar to class diagrams but NO Navigability



Description: Snakes and ladders game is played by two or more players using a board and a die. The board has 100 squares marked 1 to 100. Each player owns one piece. Players take turns to throw the die and advance their piece by the number of squares they earned from the die throw. The board has a number of snakes. If a player's piece lands on a square with a snake head, the piece is automatically moved to the square containing the snake's tail. Similarly, a piece can automatically move from a ladder foot to the ladder top. The player whose piece is the first to reach the 100th square wins.

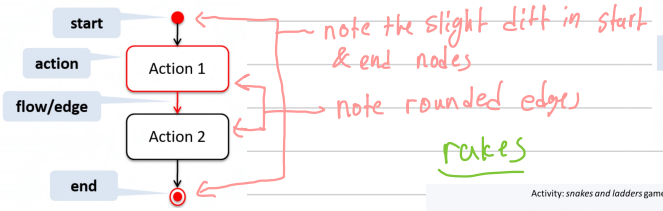
Problem statement!

↙ OODM to visualize problem statement

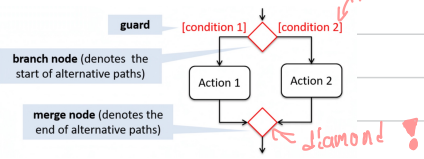
activity diagrams notations

- an activity diagram captures an activity of actions & ctrl flow that makes up the activity

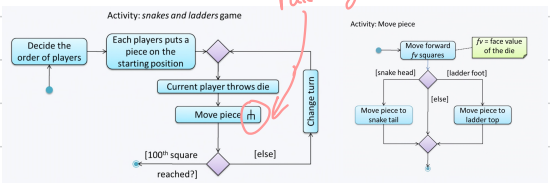
basic notation



alternate paths

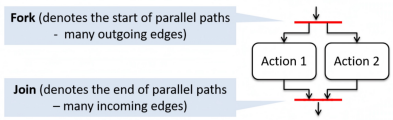


rakes



The *rake* symbol (in the `Move piece` action above) is used to show that the action is described in another subsidiary activity diagram elsewhere.

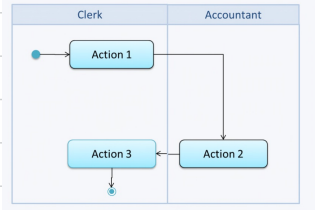
parallel paths



Swimlanes

↳ used to partion Ad to show who is doing what action

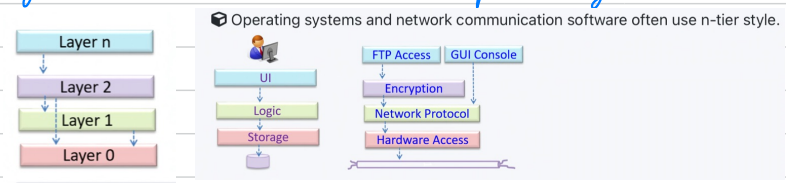
A simple example of a swimlane diagram:



CS2103T : Architectural Styles

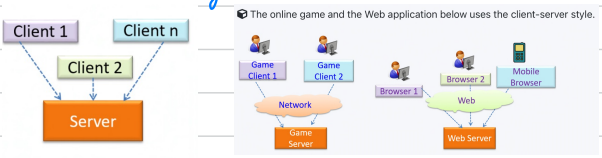
n-tier style

- higher levels make use of services provided by lower levels → lower levels independent of higher levels → often used in OS



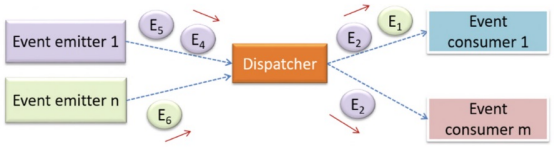
Client - server style

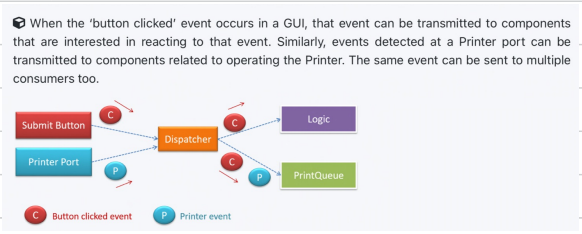
- The client-server archi. has atleast 1 component playing the role of a server and at least one client component accessing the services of the server → often used in distributed systems



Event - Driven style

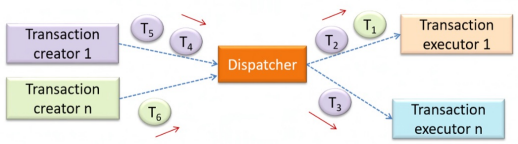
- Event-driven style controls the flow of the application by detecting events from event emitters & communicating these events to interested events consumers → often used in GUIs

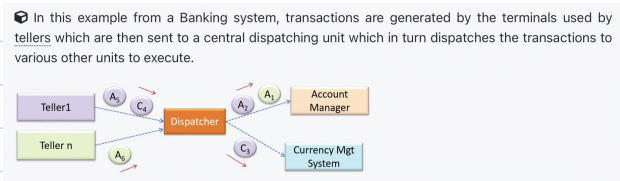




Transaction processing style

- A transaction processing style divides the work load of the system down to a number of transactions each transaction is given to a dispatcher that controls the execution of each transaction





Service oriented style

- The service-oriented architecture style builds applications by combining functionalities packaged as programmatically services

CS2103T : Design

Design approach

- Top down, btm up or mix
- agile → design emerges as project goes on

Design fundamentals

- abstraction → Data & ctrl
- coupling → Measure of the degree of dependence → high is bad → integration is harder → maintainence
  ↳ Testing & reuse is harder
- Cohesion → measure of how strongly related & focused the various responsibilities of a component are → lower cohesion is bad
  ↳ Lowers understandability of modules
  ↳ Lowers maintainability
  ↳ lowers reusability → not atomized

recall from CS2030

- Single responsibility principle → if a class has one responsibility, it changes only if that responsibility ∆
  Open-closed " → open to extension closed for modification → seperate specification from implementation
  Liskov substitution " → Subclass should not be more restrictive than the behaviour of super
  Interface segregation "
  Dependency inversion "

additional design principles

- Law of demeter
  ↳ an object shld only interact with objects that are closely related to it
  ↳ " " " " have limited knowledge about another object
  * prevent navigating to internal structures of other objects
- Seperation of concerns
  ↳ seperating code s.t each section of code addresses a specific concern

CS2103T: Design patterns

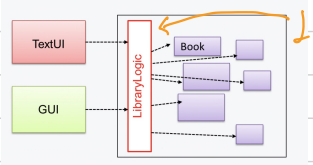
Singleton

- Sometimes certain objects should not have more than one instance
  - ↳ make constructor private & have the class itself have a static reference to itself

Pros	Cons
easy to apply	Global variable = ↑ coupling
effective in achieving goal w/ min effort	difficult to replace with stubs for testing
easy way to access singleton object	Singleton obj carry data from one test to another

Facade pattern

→ component need to access functionality deep inside other components, but accessor shld not be exposed to its internal details
  - ↳ include a facade class that sits between the component internals and users of the component such that all access to component is thru facade class



Library logic acts as facade class

Command pattern

→ System required to execute multiple commands each doing diff tasks
  - ↳ general command class → use concept of polymorphism → logic that executes command does not need to know the specific types of command

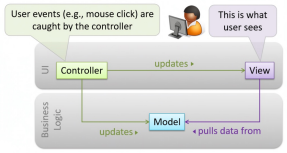
MVC pattern

→ some applications support storage/retrieval of information, displaying of info to user & changing data based on external outputs. → but can link to high coupling due to interlinked features

Decouple data, presentation, and control logic of an application by separating them into three different components: *Model*, *View* and *Controller*.

- *View*: Displays data, interacts with the user, and pulls data from the model if necessary.
- *Controller*: Detects UI events such as mouse clicks, button pushes and takes follow up action. Updates/changes the model/view when necessary.
- *Model*: Stores and maintains data. Updates views if necessary.

The relationship between the components can be observed in the diagram below. Typically, the UI is the combination of view and controller.



Observer pattern

→ sometimes observing an object directly ↑ coupling
  - ↳ force comms thru an observer interface known by both classes

CS2103T: Testing

preface

- We are familiar with unit testing & how stubs can help to isolate the unit. What are other types of testing

Integration testing

- testing whether different parts work together as expected → testing the "Glue" of the code

System testing

- testing the whole system against system specifications

acceptance testing

- testing the system to ensure it meets user requirements

System Testing	Acceptance Testing
Done against the system specification	Done against the requirements specification
Done by testers of the project team	Done by a team that represents the customer
Done on the development environment or a test bed	Done on the deployment site or on a close simulation of the deployment site
Both negative and positive test cases	More focus on positive test cases

* difference between acceptance & system testing
* passing systems testing != passing acceptance testing

alpha beta testing

- Alpha testing → performed by users, under controlled conditions
  beta → " under natural work conditions

Exploratory & scripted testing

→ exploratory → devise test cases on-the-fly based on results of previous test cases
  scripted → write test cases based on expected behaviour of SUT

Test cases

- exhaustive testing → not practical
  ↳ each test case cost $$$
  ↳ ∴ each test case must be designed to make best use of testing resource
  ↳ each test case must be effective & efficient
  finding more bugs with same amt of test cases ↲ ↳ finding same amt of bugs with lesser test cases
  ↳ to be E&E → test case shld cover potential fault not already covered by existing test cases

equivalance partition

- SUTs do not treat each input in a unique way → all possible inputs processed in small & distinct no. of ways
- equivalance partition refers to a grp of test case that are likely to be processed by SUT in the same way

Boundary value analysis

- BVA is a test case heuristic that is based on the observation that bugs occur due to incorrect handling of eps
  ↳ choose 3 values around boundary → boundary itself / boundary -1 / boundary +1

Combining multiple inputs

- some SUTs can take multiple inputs ↳ testing all possible combination → effective but not efficient

input comb strats

- all combis, at least once / all pairs
- Each valid input at least once in a tve test case
- No more than one invalid input in a test case

CS2103T: Code quality

Naming conventions	- use nouns for classes/vars & verbs for methods/fns - Distinguish between single-value & multivalued variables - use name to explain → accurately & as concise - related things named similarly, while unrelated things should not
readability	- Avoid long methods, deep nesting, complicated expressions, magic num & strings - KISS → keep it simple stupid - SLAP → single level of abstraction per method - prominent happy path → happy path = execution path when everything goes well
unsafe practice	- missing a default branch, recycling variables or parameter, empty catch blocks not deleting dead code, not scoping variables properly, code duplication
commenting	- Do not repeat the obvious - Write to the reader not yourself - explain what & why not how

CS2103T: Reuse

APIs	- APIs are application programming interface → it specifies the interface through which other programs can interact with a software component
Library	- collection of modular code that is general & can be used by other programmers
frame works	- A software frame work is a reusable implementation of a software providing generic functionality that can be selectively customized to produce a specific application. ↳ some frame works provide a complete implementation of a default behaviour which makes them immediately usuable ↳ A frame work facilitates the adaptation & customization of a desired functionality
frame work vs Library	- libraries cannot be modified, frameworks are meant to be modified - your code calls the library, a frame work calls your code
platform	- provides a runtime env for applications