CS2103t
Software Engineering 22/23s2
github.com/securespider

PE

Bug reporting format
- Descriptive title with appropriate severity/type
- Steps to reproduce, expected, actual and screenshots

UG
- Navigation does not work

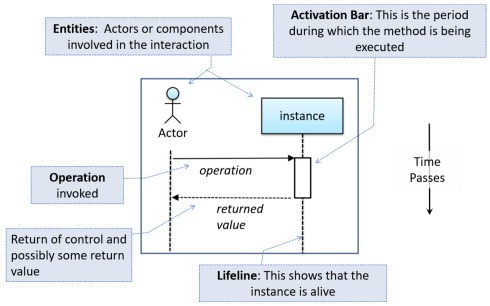
Visuals
- Not enough/unnecessarily repetitive pictures or improperly sized
- Visuals not integrated with the explanation

Content
- Not enough/too many examples (input/outputs)
- Target user/value proposition not specified clearly
  - Concise statement that explains to users how they benefit
- Feature is not named appropriately or doesnt make sense
- Feature is not explained well for new users
- Info not the same as the application (error message different)
- Messy, improperly formatted, repetitive

Product
Functionality Flaw
- Case sensitivity
- Error message does not match error
- Writing incorrect string types (age - five instead of 5)
- Giving values outside of boundary values (age - 999/0/-1)
- Long names/description/taglist/address
- try different date time formats/ dates that do not exist (leapyears)
- Are special characters legal?
- Test complicated features like undo and redo
- Giving duplicate objects/values
- Incorrect formats (age - 0001 != 1)
- Does not save correctly after closing and reopening
- How does product work if json file deleted mid-operation

Feature Flaw
- Does not solve stated problem of intended user
- Features not well optimized for fast typist or target users
- Feature does not fit well with the product

DG
General
Image
- Text in image not matching size of main text
- Image is not put close to where it is being described
- Lower level details of multiple components there without purpose

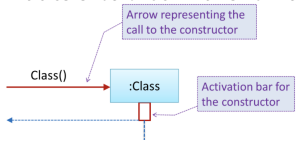
Sequence Diagram


General
- Stay at the highest level of abstraction instead of the interactions that happen inside each component
- Use visual representation
  - Associations and navigabilities using lines and arrows connecting classes instead of variables within classes

Arrows
- Must return control to caller
- Arrows representing method calls should be solid arrows whereas return should be dashed
- Return arrows are optional if it does not result in ambiguities or loss of relevant information

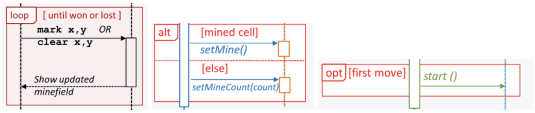
Activation bar
- Activation bar of method cannot start before method call arrives and method cannot remain active after method has returned
  - Arrow must start/end at the top/bottom tip of the activation bar
- Activation bar should remain unbroken from point method is called until return
- These are optional

Entities
- Entities should be in this format 'instanceName:Class'

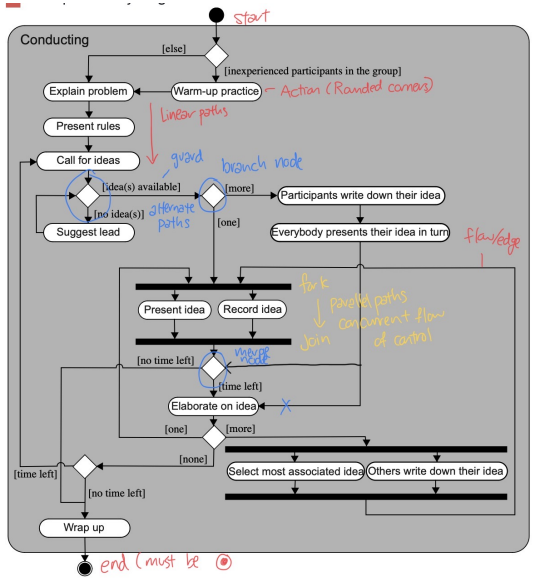


- X at the end of the lifeline of an object to show its deletion
- Method calls to static that are received by the class itself should have a $<< class >>$ at the top

Paths
- Note that the boxes are not rectangles



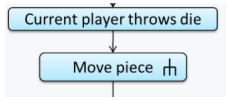
Activity Diagram
- Consist of start, action, flow/edge



Alternate paths
- Branch node shows start of alternate path with guard conditions
  - **guard** - Boolean condition has to be true for execution to take the path
- Merge node and else conditions can be omited
- Arrows **MUST** start from the corner

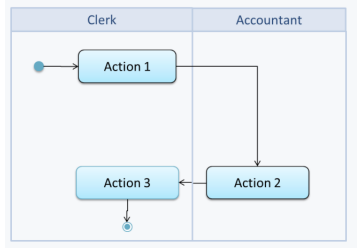
Parallel path
- Execution along all parallel path **MUST** be complete before execution on outgoing path of join

Rakes



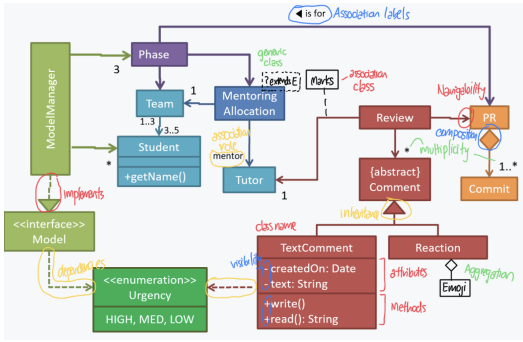
- Indicate that a part of the activity is given as a separate diagram

Swim lanes



- Partition activity diagram to show who is doing which action

Class Diagram



Class representation
- Operations and attributes compartment can be omitted if not important
- Underlines denote class-level attributes and methods

Visibility
\+ Public
\- Private
\# Protected
~ Package private

Associations
- Solid line that can have additional decorations like labels, roles, multiplicity and navigability
- Multiplicity (eg. m..n - between m and n inclusive, n - exactly n, * - 0 or more objects)
  - If class A has multiplicity 2, means 2 objects of A associated to 1 object of other class
- Navigability - There is a reference from one class to another (can be unidirectional or bidirectional)

Composition vs Aggregation
- Composition represents strong whole-part relationship
  - If whole is destroyed, parts are destroyed too (eg. Person = whole, name = part)
  - Commonly used when parts of a big class carved into smaller class for better management
  - Solid diamond
- Aggregation represents container-contained relationship
  - Containee object can exist even after container object is deleted (eg. Person in a team)
  - Hollow diamond

Dependencies vs association
Association Object keeping reference of another
Dependency Class accessing some method/value of another but no association

Inheritance
- Abstract classes/methods should either be italicised or with '$\{abstract\}$' keyword

Association Classes
- Represents additional information about association
- Should be dotted from the association btw 2 classes

Object Diagram

Objects

- Class name and object name (optional) must be underlined in the format 'objectName:ClassName'
- Should not include methods, only attributes that are relevant to the task

Non-functional requirements

- Specify the constraints under which the system is developed and operated

Example requirements

Data Size, volatility, persistency (shouldnt be more than 20MB, no crashes, respond within 2s)

Environment Technical environment which the system would operate in or need to be compatible in (work on 32-bit systems with java installed)

Characteristics

- Unambiguous, testable, clear, feasible, atomic (indivisible), necessary, **implementation-free**
- Consistent, non-redundant, complete

User Stories

- Short simple descriptions of a feature told from perspective of person who wants the capability
- Must be in the format 'As a {user type/role} I can {function} so that {benefit}'
  - Benefit can be omitted if obvious
- User story should not include any implementation details

Use cases

- Interaction between the user and system for specific functionality of system
- Should only describe externally visible behaviour not internal details of a system
  - This is wrong: LMS *saves file into cache* and indicate success
- Step should give the intention of the actor instead of the mechanics
  - UI details should be omitted to give UI designer flexibility in implementation
- Can include other use case which **MUST BE underlined** (inclusions)

Main Success Scenario (MSS)

- Most straightforward interaction for a given use case, assuming nothing goes wrong
- Should be self-contained (complete usage scenario)

Extensions

- Add on to the MSS that describes exceptional/alternative flow of events
- Extensions should be numerically marked based on when the event may happen
  - Extensions marked 3a. happens just after step 3 of MSS (3a1, 3a2...)
  - Extensions marked *a happens at any step (*a1, *a2...)
  - Subsequent extensions will be 3b, 4a or *b...

Format

Software System: Online Banking System
Use case: UC23 - Transfer Money
Actor: User
Preconditions: User is logged in. State that the system is expected to be before starting use case
Guarantees: - Expected outcome/output after use case

- Money will be deducted from the source account only if the transfer to the destination account is successful.
- The transfer will not result in the account balance going below the minimum balance required.

MSS:

1. User chooses to transfer money.
2. OBS requests for details for the transfer.
   ...