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

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

UG
- Navigation does not work

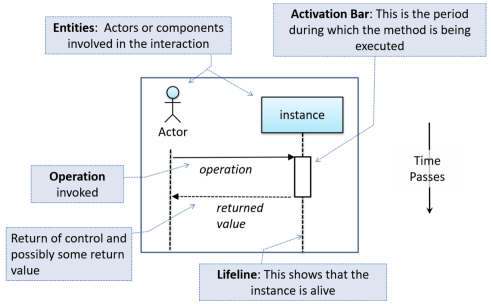
Visuals
- Not enough/unnecessarily repetitive pictures or improperly sized
- Visuals not integrated with the explanation

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

Models
- Simpler view of a complex entity

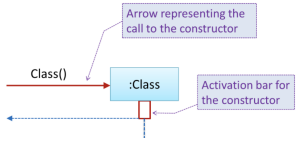
Sequence Diagram


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

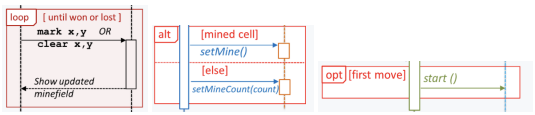
Activation bar
- Activation bar of method cannot start before method call arrives and method cannot remain active after method has returned
  - Arrow must start/end at the top/bottom tip of the activation bar
- Activation bar should remain unbroken from point method is called until return
- These are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

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

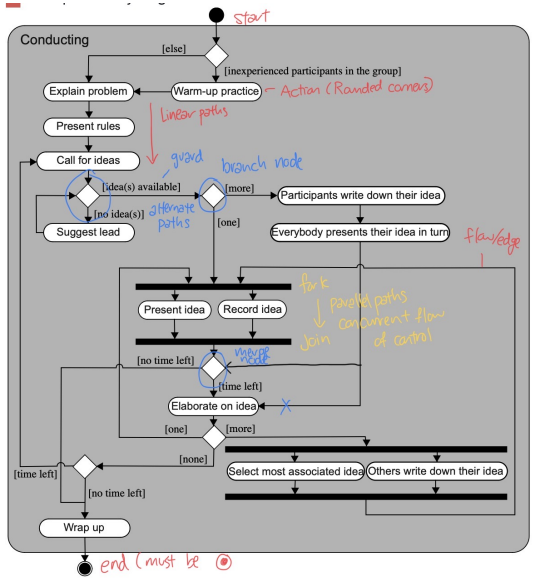


- X at the end of the lifeline of an object to show its deletion
- Method calls to static that are received by the class itself should have a $<< class >>$ at the top

Paths
- Note that the boxes are not rectangles



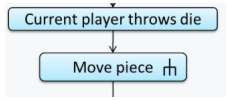
Activity Diagram
- Consist of start, action, flow/edge



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

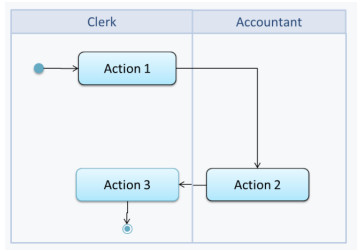
Parallel path
- Execution along all parallel path **MUST** be complete before execution on outgoing path of join

Rakes



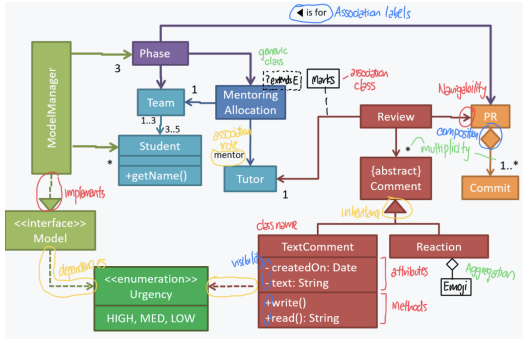
- Indicate that a part of the activity is given as a separate diagram

Swim lanes



- Partition activity diagram to show who is doing which action

Class Diagram



Class representation
- Can contain default values eg. +max:int=0
- Operations/methods and attributes compartment can be omitted if not important
- Underlines denote class-level attributes and methods
- Entities can only appear once in a diagram

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)
  - This is optional

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
- Variables of the 2 classes should not be shown in the association class

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.
   ...

Software Architecture

- Software architecture shows overall organization of system as a high-level design
- Contains a set of interacting components that fit tog for a specific functionality
- Simple and technically viable structure

Design

- Free-form diagrams with no standard notation
- Minimise variety of symbols
- Limit use of double-headed arrows to show interaction

N-tier architectural style

- aka multi-layered, layered
- Higher layer make use of services provided by lower

Event-driven architectural style

- Flow of application dependent on events from *emitters* and communicated to *consumers* (eg. GUI)

Other architectural styles

Client-server

Transaction processing Divides workload of systems into transactions controlled by *dispatcher*

Service Oriented Combining packages as *programmatically accessible services*

- Interoperability btw distrbuted system via XML

Software design pattern

- Reusable solution to a commonly recurring problem

Format

- Consist of context, problem, solution, consequences (if-any)

Singleton

Context Classes with **ONLY ONE** instance

Problem Normal classes can be instantiated multiple times by calling constructor

Solution Make constructor private and provide public static method to access single instance

- Method instantiates instance when executed for the first time
- Subsequent calls return single instance of class

Notation $<< Singleton >>$ above the class name in class diagrams

Abstraction occurrence pattern

Context Groups of similar entities sharing the same info but slightly different

Problem Duplication of data which can lead to inconsistencies

Solution Create an $<< Abstraction >>$ class that holds common information and have unique information in an $<< Occurrence >>$ class

- eg. Abstraction: BookTitle, Occurence: BookCopy (storing only the serial number)

Facade pattern

Context Components need to access functionality deep inside other components

Problem Internal details may be exposed when component is accessed

Solution Facade class sitting between component internals and users

- All access to component happens through the facade class

Command pattern

Context System required to execute number of commands doing specific tasks

Problem Other objects do not need to know command type to execute commands

Solution General $<< Command >>$ object that is passed around, stored, executed using polymorphism

Model View Controller (MVC) pattern

Context Application supporting storage/retrieval of info, displaying of info and changing stored info from external inputs

Problem High coupling from the above features

Solution Decouple data, presentation and control logic of application into 3 different components

Model Stores and maintains data

View Displays data, interacts with user and pulls data from model

Controller Detects UI events and executes commands which updates models/view if necessary

Observer pattern

Context Multiple objects affected by a change in another object

Problem Observed object should be decoupled from 'observing' objects

Solution Force communication through interface known to both parties

Design approach

Top-down/Bottom-up design

Top-down Design high-level before lower level

- Useful when designing big novel systems where high-level design need to be stable

Bottom-up Design lower-level and put them together to create higher-level

- Not usually scalable for bigger systems
- Useful when designing variation of existing system or repurposing existing components

Agile design

- Emergent, not defined up front
- Evolves to fulfil new requirements and take advantage of new technologies as appropriate
- Some initial architectural modeling

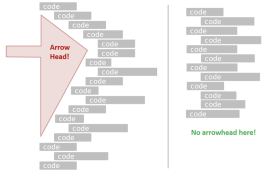
Code Quality

Readability

Avoid...

Long methods Methods should be less than 30 lines of code (LOC)

Deep nesting Less than 3 levels of indentation



Complicated expressions Avoid negations and nested parentheses

- Evaluate complicated expressions in steps with intermediate values

Magic numbers Unexplained numbers should be a named constant

Premature Optimization • May not know which parts are performance bottlenecks

- Complicate code further affecting correctness
- Hand-optimization may be slower for compiler to optimise

Reusing parameters Should redeclare local variables with same value as parameter - error prone

Should..

Make code obvious Explicit type conversions, parentheses to show groupings, enumerations

Structure code logically Lay out code to adhere logical structure with classes, methods, indentation, linespacing

Not trip up reader Unused parameters in method, barely similar/different things, multiple statements in same line

KISS Keep things simple (brute-force may be better than complicated

SLAP Single Level of Abstraction Principle

Bad (readData(); and salary = basic + rise + 1000; are at different levels of abstraction)

```
readData();
salary = basic + rise + 1000;
tax = (taxable ? salary * 0.07 : 0);
displayResult();
```

Good (all statements are at the same level of abstraction)

```
readData();
processData();
displayResult();
```

Prominent happy path Restructure code to make happy path unindented (using guard clauses)

Style guide

Naming

- Names representing packages should be in lowercase
- Class/enum names must be nouns and written in PascalCase
- Variable names should be in camelCase
- Constant names should be all UPPERCASE using underscore to separate_words
- Methods should be verbs and in camelCase
- Boolean variables/methods should sound like booleans (eg. isVisible, hasSomething())
  - Use prefix such as 'is', 'has', 'was', 'can'
- Plural form should be used on names representing a collection of objects
- Iterator variables can be called i, j, k etc

Layout

- Indentation should be 4 spaces
- Line length should be less than 120 char
- Indentation for wrapped lines should be 8 spaces more than parent lines (2 tabs)
- Egyptian/K&R style eg. while (done) {
- Methods definition: 'public void someMethod() throws SomeException {'
- Note there is no indentation for 'case' clauses
- The explicit //Fallthrough comment should be included whenever there is a 'case' statement without a break statement

Statements

- Every class should be in some package
- Imported classes should always be listed explicitly (no wildcard imports)
- Array specifiers should be attached to type not the variables (eg. int[ ] a $>$ int a[ ])
- Loops must be wrapped by curly brackets
- Conditionals should be placed on separate lines

Javadocs

- Opening /** on a separate lines
- First sentence should be a short summary of method
  - Starts with strong verbs 'returns', 'sends', 'adds'
- Subsequent * is aligned with the first *
- Space after each *
- Empty line between description and parameter section
- Punctuation behind each parameter description
- No blank line between documentation block and method/class
- @return can be omitted if method does not return anything

Unsafe shortcuts

- Use default branches (final default and else statements)
- Don't recycle variables or parameters
- Avoid empty catch blocks
- Delete dead code
- Minimize scope of variables (limit global variables and keep variable definition within blocks)
- Minimize code duplication

Comments

- Do not repeat the obvious
- Write to readers
- Explain what and why not how

Refactoring

- Process of improving a program's internal structure in small steps without modifying its external behaviour
- May improve performance and uncover bugs
- Refactor $\neq$ rewriting/bug fixing

Consolidate Duplicate Conditional Fragments

Context Same fragment of code in all branches of conditional expression

Solution Move fragment outside the expression

Extract Method

Context Code fragments that can be grouped together

Solution Turn fragment into a method whose name explains purpose of method

Integration

- Combining parts of software product to form a whole

Timing

Late and one time Integration done when all components are completed

- Not recommended due to possible component incompatibilities

Early and frequent Evolve each part in parallel in small steps

Extent

- Big bang vs incremental

Build Automation

- Automating steps of build process using scripts (compiling, linking, packaging)
- Help in dependency management

Continuous Integration and Continuous Deployment (CI/CD)

CI Integration, building and testing happens automatically after code changes

CD Changes deployed to end-users as code changes

Reuse

- Robustness of software system can be enhanced while reducing manpower and time requirement
- Reused software may have bugs or not mature enough

Application Programming Interface (API)

- Specifies the interface for other programs to interact with software component

Libraries

- Collection of modular code that is general and can be used by other programs
- Ensure that library functionality fits your needs
- Check license if it allows reuse in the way you use

Framework

- Reusable implementation of a software providing generic functionality
- Customisable to produce a specific application
- Similar overall structure and execution flow for specific category of software system
- Meant to be customised/extended rather than 'as-is'
- **Hollywood principle** - Inversion of control, framework code calls your code

Platform

- Runtime environment for application
- Bundled with libraries, tools, framework and tech

Quality Assurance

- Ensuring software being built has required levels of quality

Verification Requirements implemented correctly

Validation Requirements are correct

Code reviews

- Systematic examination of code for improvements
- PR review, pair prgming, formal inspection

Formal verification

- Mathematical techniques to prove correctness of prgm

Testing

- Executing a set of test cases by specifying input to **software under test (SUT)** and expected behaviour
  - Failure is a mismatch btw expected and actual behaviour

Testability - Indication of how easy it is to test SUT depending on design and implementation

Exploratory vs scripted

- **Scripted** - Writing set of test cases based on expected behaviour and perform testing based on test cases
  - Systematic $\rightarrow$ more bugs detected within sufficient time
- **Exploratory testing** - Devise test cases on the fly based on results of past test cases
  - Simultaneous learning, test design and execution
  - Dependent on tester experience and intuition
  - Quick error discovery

Regression Testing

- **Regression** - Modification of system that results in unintended and undesirable effects on system
- Re-testing software to detect regression
- Test all related components even if tested before
- More effective when done frequently after each small change (automation)

Developer testing

- Testing done by developers as opposed to end-users/professionals
- Early bug detection $\rightarrow$ easier and cheaper to fix
- Do not wait til the end cus of large search space or major reworks
- Bugs may hide other bugs

Unit testing

- Testing individual units to ensure each piece works correctly using Stubs
- For each class/method separately

Stubs

- Same interface as the component it replaces but with a simple implementation that is unlikely to have bugs
- Should have same responses as component for predetermined inputs
- Isolates SUT from dependencies to test unit in isolation

Integration testing

- Testing whether different parts of the software work together
- Aims to discover bugs in the interactions in components/"glue code"

System testing

- Take whole system and test against system specifications
- Based on specified external behaviour of system /NFRs

Alpha beta testing

- Deploying to users to test
- **Alpha** - Performed under controlled conditions set by SE team
- **Beta** - Given to selected subset of users to test in natural work settings

Dogfooding

- SE team use own product IRL to test

Acceptance testing

- Test system to ensure it meets user requirements
- Defined at the beginning of project based on user case specification

Automation

Test Drivers

- Code that 'drives' SUT for purpose of testing
- Invokes SUT with test input and verifies if behaviour is as expected (throwing errors)

JUnit

- Tool for automated testing of java programs

GUI testing

- Recommended to move logic out of GUI
- Tools: Visual Studio, TestFX, Selenium

Test Coverage

- Metric used to measure how much testing exercises the code

Function/method coverage Based on no. functions executed

Statement coverage Number of lines of code executed

Decision/branch coverage Based on decision points exercised (if else)

Condition coverage For all boolean sub-expressions - evaluated to T and F with different test cases

Path coverage Possible paths through a given part of the code executed

- More complicated than statement and branch

Entry/exit coverage Possible calls to and exits from operations in the SUT

Dependency injection

- 'Injecting' objects to replace current dependencies with different object
- Insert stubs to isolate SUT (Used with unit test)
  - Polymorphism - stub extends/inherits expected class

Test-Driven Development

- Writing tests before writing SUT thereby defining precise behaviour of SUt using test code

Test Case design

Equivalence partition

- Identify groups of inputs that are likely to be processed in the same way
- Ensure all partitions are tested, limiting number of inputs from each partition

Boundary value analysis

- Testing at boundaries of equivalence partitions
- Choose 3 values around boundary (before, at, after)

Combining inputs

All combinations

At least once 3 test case for 3 attributes

All pairs For any given pair of inputs, all combinations btw them are tested

- Only need to test pairwise interactions
- Less cases than all combinations strategy

Random strategy Random subset of the other strategies

Heuristic

- Each valid input at least once in a positive test case
  - Ensure that the valid input results in positive results
- No more than 1 invalid input in a test case
  - Else would not know which input is the "wrong" one

Revision Control (RC)

- Managing multiple versions of a piece of info
- Track history for better collaboration
- Mistake recovery

Repository

- Database where meta-data about revision history are stored

Staging Specifying files to track and ignore

Commit Save snapshot of current state of tracked files in RC history

Diff Compare changes btw 2 points in history

Checkout Restore state of working directory at a point in the past

- Commits are uniquely identified by auto-generated hash or tagged

Remote repository

- Repos hosted on remote computers

Clone Create a copy of that repo in a location on your computer wit all version history

Upstream repo Original repo that was cloned

Pull Receive new commits in second repo from upstream repo

Push Copy new commits to destination repo with write access and a shared history

Fork Remote copy of a remote repo without write permissions

Pull req Mechanism for contributing code to a remote repo with shared history

Branching

- Evolving multiple versions of software in parallel

RCS Revision Control Software

Merged New commit that maps all changes in other branch to curr

Conflicts Merging 2 branches that changed the same parts

- RCS cannot decide changes to keep
- Manual conflict resolution

CRCS and DRCS

Centralized RCS Central remote repo shared by team

- Members pull and push changes btw local repo and central repo

Distributed RCS Multiple remote repo pulling and pushing in arbitary ways

Forking

- All team members fork main repo and create PR from fork to main repo for changes

Software Development Life Cycle (SDLC)

- Provides roadmap for software developers to manage the development effort
- Requirements, analysis, design, implementation and testing

Waterfall/Sequential model

- Models software development as a linear process through the development stages
- Each stage of the process should produce some artifacts to be used in the next stage
- Useful model when the problem statement is well-understood and stable
  - IRL requirements are rarely understood at the beginning and keep changing

Iterative models

- Multiple iterations that produces an improved version of the product
- Feedback is fed to the next iteration and improved

Breadth-first All major components evolved in parallel

Depth-first Fleshing out individual components before moving to other features/components

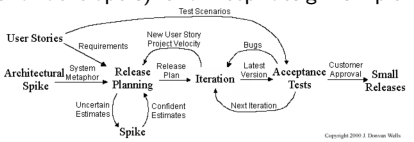
Agile

- Requirements prioritised based on needs of users and are clarified regularly
- Transparency and responsibility sharing among team members
- Team works on rough plan and high level design that evolves

Examples

Extreme Programming (XP)

- Priority on customer satisfaction, teamwork
  - Delivers software as needed
  - Respond to changing customer requirements continuously
- Communicate with stakeholders (managers, customers and developers) and keep design simple and clean



Scrum

- Process skeleton containing sets of practices and roles
  - Scrum master, product owner, team
- Divided into iterations called Sprints (timeboxed to specific duration and effort)
- Preceded by planning meeting where tasks are identified and commitments made

Unified Process

- Four phases - Inception, Elaboration, Construction and Transition

Inception Understand problem and requirements by communication with customer

Elaboration Refine and expand requirements

Construction Major implementation effort to support use case

Transition Ready the system for actual production use

- Flexible, customisable process model framework

Capability Maturity Model Integration

- Process improvement approach by defining maturity levels for processes
- Criteria to determine maturity level of processes

Quiz

2

- Module/SE details, Software Development Life Cycle (SDLC)
- Integrated Development Environment (IDE), Revision Control (git terms)
- Testing (regression)

5

- Object diagram
- Requirements (stakeholders, brownfield, NFR, quality, priority)
  - Gathering requirements via brainstorming, wireframes
- Writing DG (glossary, feature list, user stories)
- Code quality (refactoring, naming, comments)
- Assertions, streams

6

- Sequence diagrams
- Architecture diagrams (free-form)

7

- Use cases (MSS)
- Design principles, abstraction, coupling, cohesion
- Integration and project management

8

- Diagrams (class, sequence, object)
- Testing (types, coverage)

9

- Diagrams (OODM, Activity)
- Principles and SDLC

10

- Design patterns (singleton, facade, command)
- Defensive programming, test case design

11

- Design patterns (MVC, Observer)
- Architecture styles
- Testing heuristics, QA (validation vs verification), reuse (platform, frameworks)