CS2103 Finals helpsheet

by m.zaidan
for AY24/25 S1

Requirements

Functional

Specify what system does

Non-functional

Specify constraints under which system is developed and operated

Examples

- Business/domain rules
- Constraint
- Technical
- Performance
- Quality
- Process
- Project scope

Gathering requirements

- Brainstorming
- User surveys
- Observation
- Interviews
- Focus groups
- Prototyping
- Product surveys

Specifying requirements

- Prose
- Feature list
- User stories
- Use cases
- Glossary

User stories

User story format

```
As a {user type/role}, I can {function} so that {benefit}.
```

If benefit is obvious, it can be omitted.

- Can be written at various levels (epics)
- Add conditions of satisfaction
- Label by priority, size and urgency

Use cases

```
System:
Use Case:
Actor:
Preconditions: (optional)
Guarantees: (optional)

MSS:

Extensions:
```

Use case

- should only describe external visible behaviou
  - 👎 LMS saves the file into the cache
  - 👍 (exclude this step)
- should only describe steps by intention, and not mechanics
  - 👎 User right-clicks on clear button
  - 👍 User clears the field

Main success scenario describes

- most straightforward
- assumes nothing goes wrong

Extensions

- if there is a number, it can happen just after that step
  - extension 3a. can happen just after step 3 of MSS
- if not, it can happen anywhere
  - extension *a can happen anywhere.

Can include other use cases by underlining it

- A does something (UC40)

Other inclusions

- Preconditions: state expected to be in before use case starts
- Guarantees: promised to happen after operation

Design

Abstraction

Establish a level of complexity interested in, and suppress more complex details below that level.

Coupling

Measure of degree of dependence between components

X is coupled to Y if a change to Y can potentially require changes in X.

High coupling means that the components are more dependent.

- Maintenance becomes harder
- Integration becomes harder
- Testing and reuse becomes harder

Types of coupling

- Content
- Common
- Control
- Data
- External
- Subclass
- Temporal

Cohesion

A measure of how strongly related and focused the various responsibilities of components are

Low cohesion

- lowers understandability of modules
- lowers maintainability
- lowers reusability

Examples:

- code related to concept is kept together
- code invoked close together in time are kept together
- code manipulating same data structure is kept together

Architecture

Styles

n-tier

Higher layers make use of services provided by lower layers

Client-server

At least one component is a client and server. Client components access services of server.

Transaction-processing

Divides workload to a number of transactions, given to dsipatcher controlling execution of each transaction.

Service-oriented

Combining functionalities packaged as programmatically accessible services.

Event-driven

Controls the flow of application by detecting events from event emitters, and communicating those to interested event consumers.

Patterns

Singleton

Context

Certain classes should have no more than just one instance

Problem

A normal class can be instantiated multiple times by invoking constructor

Solution

Make constructor `private`, to disallow instantiating at will.
Provide a `public` class-level method to access the singleton.

Consequences

✓ Pros

- easy to apply
- effective in achieving goal with not much work
- provides easy way to access singleton object from anywhere in the codebase

✗ Cons

- singleton objects acts like global variable, increases coupling
- hard to replace singleton objects with stubs in testing
- singleton objects carry data from one test to another

Implementation:

```
class Logic {
        private static Logic singleton = null;

        private Logic() {
                ...
        }

        public static Logic getInstance() {
                if (singleton == null) {
                        singleton = new Logic();
```

```
                }
                return singleton;
        }
}
```

Facade

Context

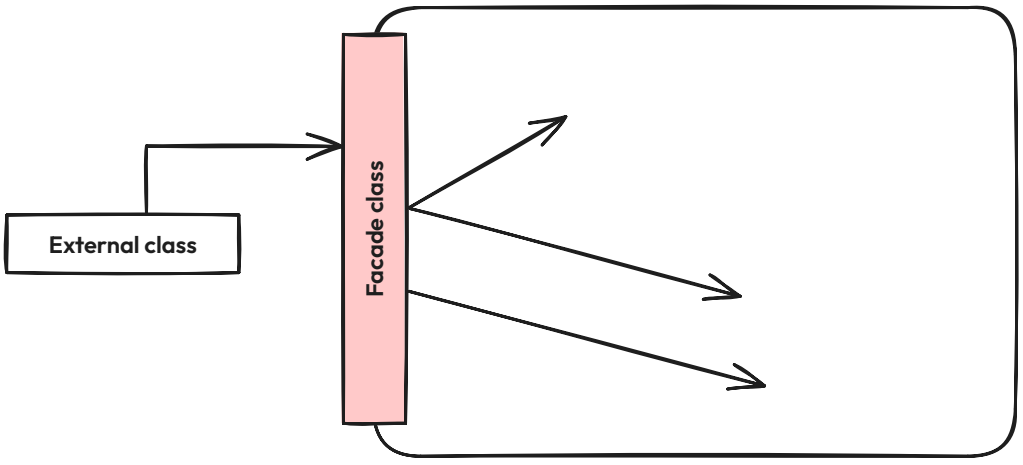
Components need to access functionality deep inside other components

Problem

Access should be allowed without exposing its internal details.

Solution

Include Facade class that sits between the component internals, and users of the component, such that all access happens through the class.



external class doesn't know anything about internal details

Command Pattern

Context

A system is required to execute a number of commands, where each do a different task.

Problem

It is preferable that some part of the code executes these commands without having to know each command type.

Solution

Essential element is to have a general `<<Command>>` object that can be passed around, stored, executed, without knowing the type of command (via polymorphism).

Abstraction Occurrence Pattern

Context

There is a group of similar entities that appear to be occurences of the same thing, with a lot of common information, but differ in significant ways.

Problem

Representing objects as a single class would be problematic because it results in duplication of data, which can lead to inconsistencies.

[!note] Solution
Let a copy of an entity be represented by two objects instead of one, with common information in one class, and unique information in another.

Anti-pattern

Segregate common and unique information into a class hierarchy, and hard-codes common data in the class. Problematic as if there is an addition of new data, it will have to update the source code.

Model-View-Controller

Context

Most application support

- storage/retrieval of information
- displaying of information
- changing stored information

Problem

High coupling can result from interlinked nature of features

Solution

Decouple data, presentation, and control logic.

- View: Display data, interact with user, and pulls data from model
- Controller: Detects UI events such as mouse clicks/button pushes, and takes follow up action. Updates model/view
- Model: Stores and maintains data, updates view if necessary

Observer

Context

Object is interested in being notified when a change happens to another object.

Problem

Coupling could happen to the observed object.

Solution

Force communication through interface known to both parties.

Approaches

- Multi-level
  - Design at different levels (from macro/micro perspectives)
  - Top-down and bottom-up
    - Top-down: Design high-level first, flesh out lower later
    - Bottom-up: Design low level first, combine them to create higher-level systems
    - Mix
- Agile
  - Not defined up front,
  - Modeling is just enough to start working.

Code quality

Readability

- Avoid long methods
- Avoid deep nessting
- Avoid complicated expressions
- Avoid magic numbers
- Make code obvious
- Structure code logically
- Avoid tripping up reader (unused parameters, similar things that look different)
- Keep It Simple, Stupid
- Avoid premature optimisations
- SLAP (Single Level of Abstraction Principle)
- Make intended path prominent and clear

Follow standards

- Adopt a style guide

Naming

- Nouns for things, verbs for actions
- Standard words
- Name should explain variable
- Name should not be too long or short
- Name should not be misleading

Avoid shortcuts

- Use default branch
- Don't recycle variables
- Avoid empty catch blocks
- Delete dead code
- Minimise scope of variables
- Minimise code duplication

Commenting

- Do not repeat obvious
- Write to reader
- Explain what and why, not how

Refactoring

> **Refactoring**
>
> Process of improving a program's internal structure without modifying external behaviour

Methods:

- consolidate conditional expression
- decompose conditional
- inline method
- remove double negative
- replace magic literal
- replace nested conditional with guard clauses
- replace parameter with explicit methods
- reverse conditional
- split loop
- split temporary variable

When to refactor:

- code smells
  - long methods
  - large classes
  - primitive obsession
  - temporary fields
  - shotgun surgery

Refactoring is too much when benefits no longer justify cost.

Documentation

- Top-down, not bottom-up
- Aim for comprehensibility
- Aim for just enough documentation

Error-handling

- Exceptions
- Assertions
- Logging
- Defensive programming
  - If things can go wrong, they will go wrong
  - Examples:
    - Enforce compulsory associations
    - Enforce 1-1 associations
    - Enforce referential integrity

Integration

Approaches:

- Early and frequent
  - preferred over late and one time
- Incremental
  - preferred over big-bang
- Top-down/bottom-up/sandwich

Reuse

- APIs
- Libraries
  - meant to be use as is
- Frameworks
  - meant to be extended
  - inversion of control
- Platforms
- Cloud computing

Quality assurance

- Code reviews
  - 👍 Detects functionality defects
  - 👍 Verify non-code artifacts
  - 👍 Does not require test drivers
  - 👎 Manual, so error-prone
- Static analysis
- Formal verification
  - 👍 Can prove absence of errors
  - 👎 Expensive
  - 👎 Requires highly specialised notation and knowledge

Testing

- Regression
- Developer
- Unit
- Integration
- System
- Alpha and beta
- Dogfooding
- Acceptance

Regression Testing

Re-testing of software to detect regressions

Unit testing

Testing of individual units to ensure each piece works correctly

Use of stubs

Stubs allow for isolation from SUT

Integration testing

Testing that focuses on interactions between units

Using a hybrid of unit and integration tests allow for less use of stubs.

System testing

Tests against system specifications

Acceptance testing

Test system to ensure it meets user requirements

Differences between system and acceptance testing:

- System is done against system specification, acceptance against requirements specification
- System is done by testers of project team, acceptance is done by a team representing customer
- System is done on developer environment/test bed, acceptance is done on deployment site or close simulation of deployment site
- System has both negative and positive, acceptance focuses more on positive test cases

Alpha testing

Testing performed by users under controlled conditions

Beta testing

Testing performed by a selected subset of target users in natural work setting

Dogfooding

Creators use own product.

Testing can be done in two ways

- scripted: write a set based on expected behaviour
- exploratory: devise test cases on-the-fly

Test coverage:

- function/method coverage
- statement coverage
- decision/branch coverage
- condition coverage
- path coverage

Dependency injection

Process of injecting objects to replace current dependencies with a different object.

Test case design

Positive vs negative test cases

Positive test cases produce expected/valid behaviours, negative test cases indicate invalid/unexpected situations (test for error handling).

Types of test case design:

- Black-box (specification/responsibility-based)
  - Based on specified external behaviour of sut
- White box (glass-box, structure/implementation-based)
  - Based on implementation of SUT
- Gray box
  - Uses some information about implementation

Equivalence partitions

Test inputs likely to be processed by SUT in same way

Boundary value analysis

Test case design heuristic is based on observation that bugs often result from incorrect handling of boundaries.

When testing EPs, values near boundaries ar emore likely to find bugs (according to BVA).
(Corner cases)

Test input combinations (in order of size, ascending)

- At least once
- All pairs
- All combinations

Heuristics to use:

- Each valid input at least once in positive test case
- Test invalid input individually before combining

Revision control

DRCS vs CRCS

- Centralised (CRCS) uses a central remote repo
- Distributed (DRCS) allows multiple remote/local repos

Forking workflow

- Official version is kept in a remote repo
- All team members fork and create pull requests

Feature branch flow

- No forks, everyone is pushing and pulling to same remote repo
- A branch is created for different features

Centralised workflow

- All changes done straight to master branch

SDLC

- Sequential (waterfall)
  - Linear
- Iterative
  - Each iteration produces version of new product
    - Breadth-first approach evolves all major components and functionality areas in parallel
    - Depth-first approach focuses on fleshing out only some components or some functionality area
- Agile

Principles

SOLID Principles

- SRP
  - Single responsibility principle
- OCP
  - Open-closed principle
- LSP
  - Liskov's substitution principle
- ISP
  - Interface segregation principle
- DIP
  - Dependency inversion principle

Separation of concerns

To achieve better modularity, separate the code into distinct sections, such that each section addresses a separate concern.

This principle allows for higher cohesion and lower coupling.

Single responsibility principle

A class should have one, and only one, reason to change.

The class should only change when there is a change to its singular responsibility.

Liskov Substitution Principle

Derived class must be substitutable for base classes.

LSP implies that a subclass should not be more restrictive than the behaviour specified by superclass.

Open-closed principle

A module should be open for extension but closed for modification. Modules should be written so that they can be extended without requiring them to be modified.

Aim: make code entity easy to adapt and reuse without modifying it itself.

Can be done by separating specification (interface) of a module from implementation

Law of Demeter

Object should have limited knowledge of another, and should only interact with objects that are closely related to it.

Interface segregation principle

No client should be forced to depend on methods it does not use.

Dependency inversion principle

High level modules should not depend on low level modules.
Abstractions should not depend on details. Details should depend on abstractions.

You Aren't Gonna Need It Principle

Do not add code simply because it might be needed in the future.

Don't Repeat Yourself principle

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Brooks' law

Adding people to a late project will make it later: additional communication overhead outweighs the benefit of adding extra manpower, especially if done near a deadline.