Software Engineering

Derrick Chua, Jethro Kuan

May 2, 2018

Contents

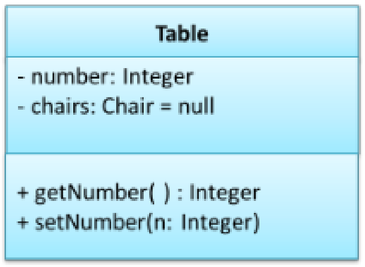
-- mode: Org; org-download-image-dir: "./images/software-engineering/"; --

1 Object-Oriented Programming

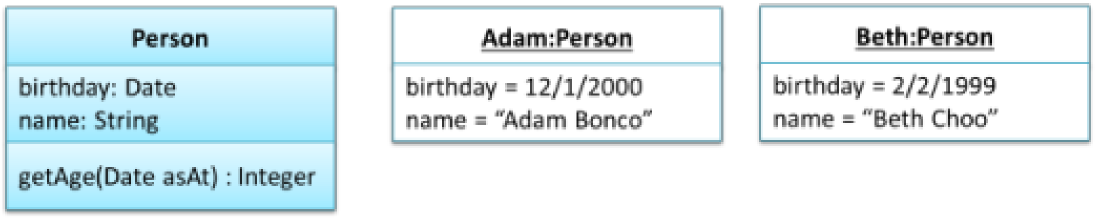
- Every object has both *state* (data) and *behaviour* (operations on data).
- Every object has an *interface* and an *implementation*.
  - Interface are for other objects to interact with.
  - Implementations support the interface, and may not be accessible to other objects.

1.1 Basic UML Notation

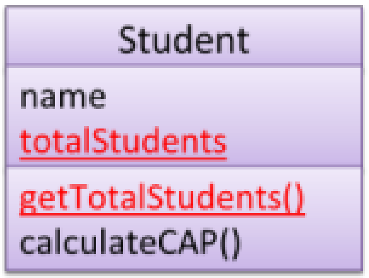
The class is denoted with 3 parts: The class name, its attributes, and its methods.



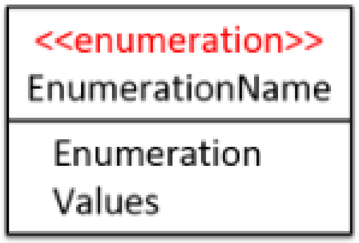
Instances of a class are denoted with 3 parts: its name (and class), and its attribute values.



Class-level attributes and variables are denoted by underlines. In the class diagram below, `totalStudents` and `getTotalStudents` are class-level._



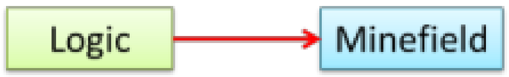
UML Notation for enumerations:



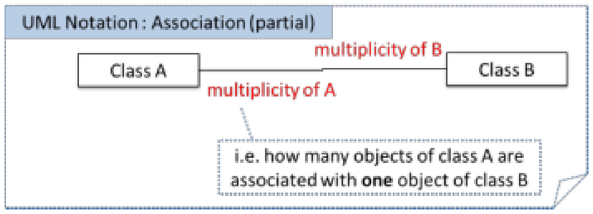
1.2 Associations

- A solid line indicates an association between 2 objects.
- The concept of *Navigability* refers to whether the association knows about the other class. Arrow heads are used to indicate the navigability of the association.

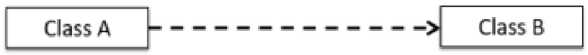
`Logic` is aware of `Minefield`, but `Minefield` is not aware of `Logic`



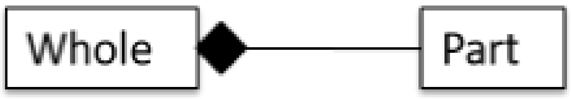
- Multiplicity is denoted on each end of the association.


UML Notation : Association (partial)
Class A
Class B
multiplicity of B
multiplicity of A
i.e. how many objects of class A are associated with **one** object of class B

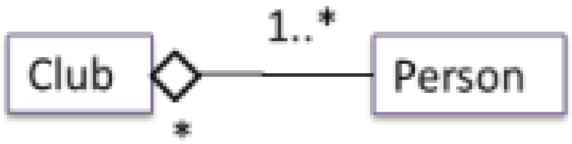
- Dependencies are weaker associations where interactions between objects do not result in a long-term relationship. A dashed arrow is used to show dependencies.



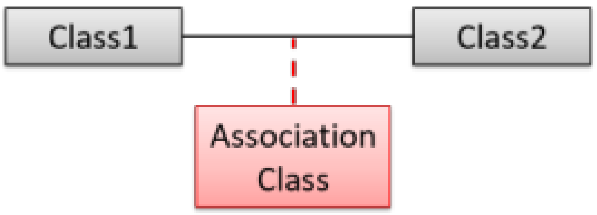
- Composition represents a strong whole-part relationship. When the whole is destroyed, parts are destroyed too. There cannot be cyclical links in composition.



- Aggregation represents a container-contained relationship.



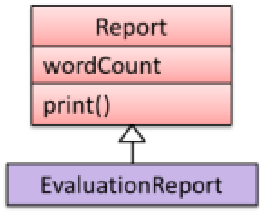
- An association class represents additional information about an association. It is a normal class but plays a special role from a design point of view.



1.3 Inheritance

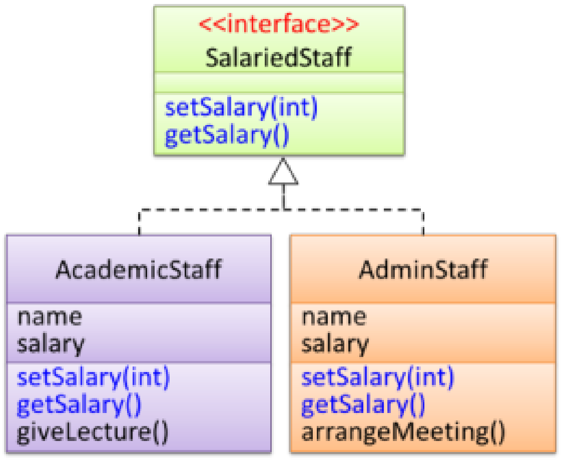
Inheritance allows you to define a new class based on an existing class. This helps group common parts among classes. This is denoted by an arrow.

Example: The `EvaluationReport` inherits the `wordCount` attribute and the `print()` method from the *base class* `Report`.

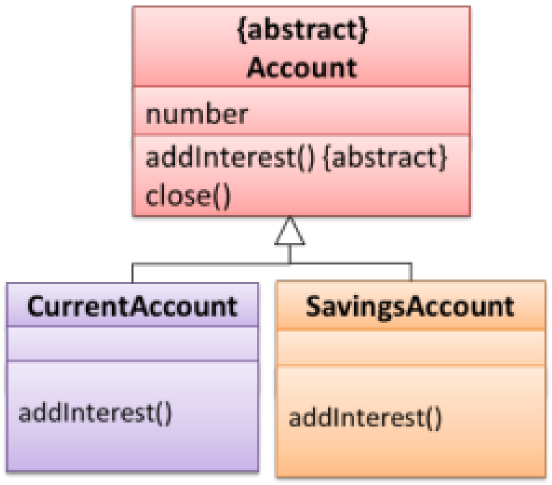


1.4 Interfaces

An *interface* is a behaviour specification. If a class implements the interface, it is able to support the behaviours specified by the interface. A class implementing an interface results in an is-a relationship. In the example below, `AcademicStaff` is a `SalariedStaff`.



An abstract method is the method interface without the implementation. It is denoted with the `{abstract}` annotation in the UML diagram.



1.5 Polymorphism

Polymorphism is the ability of different objects to respond, each it its own way, to identical messages. The mechanisms that enable polymorphism are:

Substitutability write code that expects parent class, yet use that code with objects of child classes.

Overriding Operations in the super class need to be overridden in each of the subclasses.

Dynamic binding Calls to overridden methods are bound to the implementation of the actual object's class dynamically during runtime.

2 Modelling Behaviour

2.1 Activity Diagrams

Actions Rectangles with rounded edges (Steps)

Control flows Lines with arrowheads (Flow of control from one action to another)

Alternate paths Diamond shapes

- Branch or merge nodes
- Each control flow leaving branch node has guard condition
- Only 1 alernative path can be taken at any time.

Parallel paths bar

- Forks and join
- Indicate start and end of concurrent flows of control

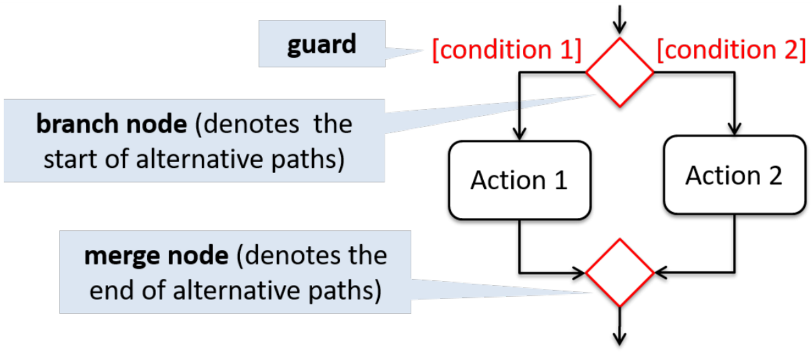
Part of Activity rakes

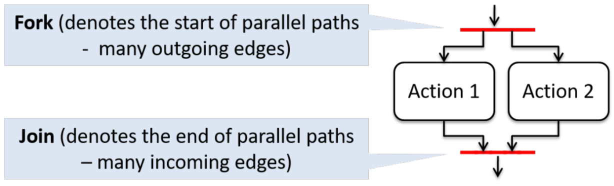
- Indicate that part of activity is given as separate diagram

- In actions

Actor partitions swimlanes

- Partition activity diagram to show who is doing which action (Who label at the top, as columns)


guard
[condition 1]
[condition 2]
branch node (denotes the start of alternative paths)
Action 1
Action 2
merge node (denotes the end of alternative paths)


Fork (denotes the start of parallel paths - many outgoing edges)
Action 1
Action 2
Join (denotes the end of parallel paths – many incoming edges)

2.2 Sequence Diagrams

Method calls Solid arrows

Method returns Dotted arrows (optional)

Loops labeled boxes

Activation bar (optional)

- Method is running and in charge of execution
- Constructor is active
- Dotted lines after activation bar shows a lifeline, i.e. it is still alive

Deletion Use a X at end of lifeline of an object

Self Invocation Draw a second bar within the activation bar for inner method and an arrow to show self invocation

Alternative paths `Alt` frames (boxes) with dotted horizontal lines to separate alternative paths

Optional paths `Opt` frames

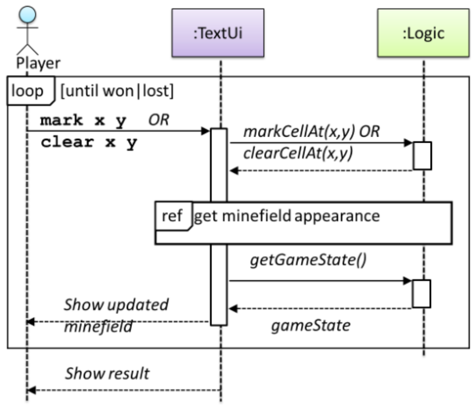
Reference frames frames

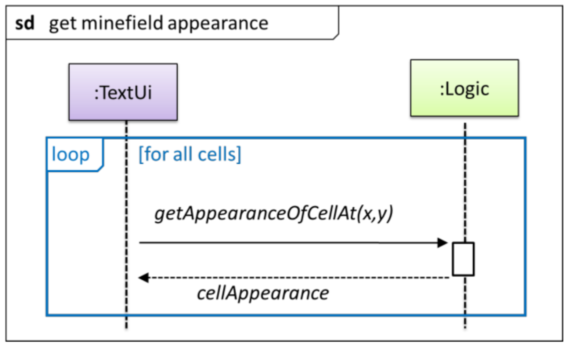
- `ref` frame to omit details/ Show frame in another sequence diagram
- `sd` frame to show details

Parallel paths For multi-threading, as multiple things are being done at the same time

Note:

- No underlined object names (e.g. :Object)





3 Software Requirements

Requirements come from *stakeholders*: parties that are directly affected by the software project.

functional requirements specify what the system should do

- data requirements: availability etc.

non-functional requirements specify the constraints under which system is developed

- business and domain rules: the size of the group cannot be more than 5
- constraints: should be backwards compatible
- technical requirements: should work on 32/64-bit environments

Good requirements are: unambiguous, testable, clear, correct, understandable, feasible, independent, atomic, necessary, implementation-free

User stories follow the format: `As a _, I can _ so that _`. They occur on different levels, high-level stories are called epics.

4 Design

Software design has 2 main aspects:

1. product/external design: designing the external behaviour of the product to meet the user requirements.
2. implementation/internal design: designing how the product will be implemented to meet the required external behaviour.

abstraction technique for dealing with complexity, establishes a level of complexity we are interested in, and suppressing more complex details below that level.

4.1 Coupling

Coupling is the **measure of the degree of dependence** between components. Highly coupled components are:

- **harder to maintain**: change in one module can cause changes to other modules
- **harder to integrate**: multiple components have to be integrated at the same time
- **harder to test**: dependence on other modules

Coupling comes in various forms:

Content Coupling one module modifies or relies on the internal workings of another module.

Common/Global Coupling two modules share the same global data

Control Coupling one module controls the flow of the other

Data Coupling one module sharing data with another module (e.g. passing params)

External Coupling two modules share an externally imposed convention

Subclass Coupling a class inherits from another class

Temporal Coupling two actions are bundled together because they happen to occur at the same time

4.2 Cohesion

Cohesion is a **measure of how strongly-related and focused the various responsibilities of a component are**. Low cohesion can:

- impede the understandability of modules
- lower maintainability because a module can be modified due to unrelated causes
- lowers reusability because they do not represent logical units of functionality

Cohesion can be present in many forms:

- Code related to the same concept are kept together
- Code invoked close together in time are kept together
- Code manipulating the same data structure are kept together

5 Software Architecture

Software architecture shows the **overall organization of the system and can be viewed as a very high-level design**.

5.1 Architectural Styles

1. n-tier
   - n layer
   - Higher layer communicates to lower tier
   - Must be independent
2. Client-server
   - At least one client component and one server component
   - Commonly used in distributed apps
3. Event-driven Style
   - Detect events from emitters and communicating to event consumers
4. Transaction processing style
   - Divides workload down to a number of transactions which are given to a dispatcher which controls the execution for each transaction
5. Service-oriented architecture (SOA)
   - Combining functionalities packaged by programmatically accessible services
   - e.g. Creating an SOA app that uses Amazon web services
6. Pipes and Filters pattern
   - Break down processing tasks by modules (streams) into separate components(filters), each into 1 task
   - Combine them into a pipeline by standardising format of data each component sends and receives
   - Bottleneck - Slowest filter
   - Components can be run independently
   - Used when processing steps by an application have different scalability requirements
7. Broker pattern
   - Broker component coordinates communication, such as forwarding requests, as well as for transmitting results and exceptions
   - Used to structure distributed software systems with decoupled components interacting by remote service invocations
8. Peer-to-peer
   - Partitions workload between peers (both 'client' and 'server' to other nodes)
9. Message-driven processing

- Client sends service requests in specially-formatted messages to request brokers(programs)
- Request brokers maintain queues of requests (and maybe replies) to screen their details

6 Software Design Patterns

Software Design Patterns are **elegant reusable solutions to commonly recurring problems within a given context in software design**.

Design patterns are specified with: context, problem, solution, anti-patterns, consequences and other useful information.

6.1 Singleton

- **Context**: certain classes should have no more than 1 instance.
- **Problem**: a normal class can be instantiated multiple times by invoking the constructor
- **Solution**: make the constructor of the singleton class `private`, provide a public class-level method to access the single instance.
- Pros:
  - Easy to apply
  - Effective with minimal work
  - Access singleton from anywhere
- Cons:
  - Global variable, increases coupling
  - Hard to test as they cannot be replaced with stubs
  - Singletons carry data from one test to another

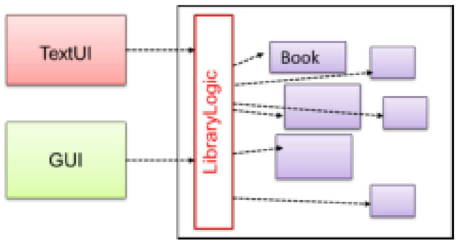
6.2 Abstraction Occurrence

- **Context**: Group of similar entities that appear to be occurrences of the same thing, sharing a lot of common information, but differ in many ways.
- **Problem**: representing objects as a single class would result in duplication of data, leading to inconsistencies in data.
- **Solution**: Let a copy of the entity be represented by multiple objects, separating the common and unique information into 2 classes.

6.3 Facade Pattern

- **Context**: Components need access to functionality deep inside other components
- **Problem**: Access to component should be allowed without exposing internal details

- **Solution**: Create a Facade class that sits between the component internals and users of the component that access the component happens through the facade class.



6.4 Command Pattern

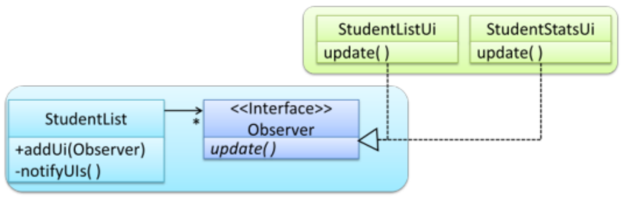
- **Context**: A system is required to execute a number of commands,s each doing a different task.
- **Problem**: Prefer to have code executing command to not have to know each command type
- **Solution**: Have a general `Command` object that can be passed around, stored and executed without knowing the type of command.

6.5 MVC Pattern

- **Context**: applications support storage/retrieval of information, display, and updating stored information
- **Problem**: want to reduce coupling between interlinked nature of the above features
- **Solution**: *View* displays data, interacts with the user. *Controller* detects UI events, and updates the model/view when necessary. *Model* stores and maintains the data, updates the views if necessary.

6.6 Observer Pattern

- **Context**: An object is interested in getting notified when a change happens to another object
- **Problem**: the observed object does not want to be coupled to objects that are 'observing' it
- **Solution**: Force the communication through an interface know to both parties.



7 Implementation

Debugging process of discovering defects in the program.

- inserting temporary print statements incur extra effort, manually tracing through code is difficult and time consuming. We should use a debugger tool, which allows pausing and stepping through execution of the code.

7.1 Code Quality

There are various dimensions of code quality, including **run-time efficiency, security, and robustness**. The most important perhaps is **readability**.

Some basic guidelines:

1. Avoid long methods
2. Avoid deep nesting
3. Avoid complicated expressions
4. Avoid Magic numbers
5. Make the code obvious, e.g. by using explicity type conversion
6. Structure code logically
7. Do not trip up the reader, with things like unused parameters in the method signature
8. Practice KISSing
9. Avoid premature optimizations
10. Make the happy path prominent
11. SLAP (Single level of abstraction per method) hard
12. Make the happy path prominent

It is also good to follow a **coding standard**.

Comments should explain the what and why, and not the how. Write comments minimally but sufficiently, not repeating the obvious and writing with the reader in mind.

7.2 Refactoring

Refactoring **improves a program's internal structure in small steps without modifying its external behaviour.** It is not rewriting, and not bug-fixing.

Common refactors include:

- Consolidate duplicate conditional fragments
- extract method

7.3 Error Handling

Exceptions are events that occur during the execution of a program, that **disrupt the normal flow of the program's instructions**.

Exception objects encapsulate the unusual situation so that another piece of code can catch it and deal with it. Exception objects propagate up the method call hierarchy until it is dealt with.

7.4 Assertions

Assertions are used to define assumptions about the program state so that the runtime can verify them. If the runtime detects an assertion failure, it typically takes some drastic action, such as terminating the program. Assertions can be disabled without modifying the code.

7.5 Logging

Logging is **The deliberate recording of certain information during program execution for future reference**, and is useful for troubleshooting problems. Most languages come with a logging mechanism, and the logger has different levels: SEVERE, INFO, WARNING etc.

7.5.1 Build automation

1. Gradle
    - Automates tasks such as:
        - Running tests
        - Manage library dependencies
        - Analyse code for style compliance
    - Gradle configuration is defined in build script build.gradle
    - Gradle commands are run in gradlew (wrapper) which runs the following commands by default:
        - clean
        - headless
        - allTests
        - coverage
    - Dependencies are updated automatically by other relevant Gradle tasks

7.5.2 Continuous Integration (CI)

1. Integration, building and testing happens automatically after code change
2. Travis CI
3. Continuous Deployment (CD) - Changes are integreated, and deployed to end-users at the same time (e.g. Travis)

7.6 Defensive Programming

- Leave no room for things to go wrong
- Enforce compulsory associations(perform checks for null)
- Enforce 1-to-1 associations (Initialise an association as a new object first, before assignment)

- Enforce referential integrity (Inconsistency in object references) (invoke the peer method when one is called)

7.7 Design-by-Contract

DbC is an *approach for designing software that requires defining formal, precise and verifiable interface specifications for software components*.

- Meet interface specifications for different components (preconditions must be met) to fulfil contract

7.8 Integration Approaches

1. Late and one-time
   - Wait till all components are completed and integrate all finished components near end of project
   - Not recommended due to possible component incompatabilities, which can lead to delivery delays
2. Early and frequent
   - Integrate early and evolve each part in parallel, in small steps, re-integrating frequently
3. Big-Bang vs Incremental Integration
   - Big-bang can lead to many problems at the same time
4. Top-Down vs Bottom-Up
   - Top-Down require stubs
   - Bottom-up require drivers
   - Sandwich for both to 'meet' in the middle

7.9 Reuse

By reusing tried and tested components, the robustness of a new software system can be enhanced while reducing the manpower and time requirement. There are costs associated with reuse.

7.10 APIs

An Application Programming Interface (API) specifies the interface through which other programs can interact with a software component.

7.11 Libraries and Frameworks

A library is a collection of modular code that is general and can be used by other programs. A software framework is a reusable implementation of a software providing generic functionality that can be selectively customized to produce a specific application. Libraries are meant to be used 'as is' while frameworks are meant to be customized/extended. Your code calls the library code while the framework code calls your code.

7.12 Platforms

A platform provides a runtime environment for applications. A platform is often bundled with libraries, tools and frameworks.

8 Quality Assurance

QA ensures that the software being built has the required levels of quality. This is achieved through:

code reviews the systematic examination of code with the intention of finding where the code can be improved

static analysis analysis of the code without actually executing the code (e.g. Linters)

formal verification mathematical techniques, used to prove the correctness of a program. it can only be used to prove the absence of errors, but only proves compliance with the specification, and not the actual utility of the software.

8.1 Testing

When testing, we execute a set of test cases, containing the **input** and the **expected behaviour**. Test cases can be determined based on the specification.

8.1.1 Unit Testing

- testing individual units to ensure each piece works correctly. In OOP, this includes writing one or more unit tests for each public method of a class.
- A proper unit test requires the unit to be testing in isolation, hence stubs are created for the dependencies.
- **Dependency injection** is the process of replacing current dependencies with another object, commonly seen with stubs. Polymorphism can be used to implement this.

8.1.2 Integration Testing

- testing whether different parts of the software work together as expected. It aims to discover bugs in the "glue code" related to how components interact with each other.

8.1.3 System Testing

- Takes the whole system and tests it against the system specification
- System test cases are based on the specified external behaviour of the system
- System testing includes testing against non-functional requirements

8.1.4 Others

- alpha testing is performed by the users, under controlled conditions set by the software development team
- beta testing is performed by a selected subset of users of the system in their natural work setting
- dogfooding is the creators of the product using their own product
- developer testing is done by the developers themselves, so as to locate the cause of test case failure or fixing bugs
- regression testing is the retesting the SUT to detect regressions when a system is modified.

8.1.5 Exploratory vs Scripted Testing

- Exploratory testing devises test cases on-the-fly, creating new test cases based on the results of past test cases
  - dependent on the tester's prior experience and intuition
- Scripted testing is a set of test cases based on the expected behaviour of the SUT
  - more systematic, and hence likely to discover more bugs given sufficient time

8.1.6 Acceptance Testing

- test the delivered system to ensure it meets the user requirements

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

8.1.7 Coverage

Coverage is the metric used to measure the extent to which testing exercises the code.

function/method coverage based on the functions executed

statement coverage based on the number of lines of code executed

decision/branch coverage based on the decision points exercised

condition coverage based on the boolean sub-expressions

path coverage in terms of possible paths through a given part of the code executed

entry/exit coverage in terms of possible calls to and exits from the operations in the SUT

8.1.8 Test Case Design

black-box designed exclusively based on the SUT's specified external behaviour

white-box test cases are designed based on what is known about the SUT's implementation

gray-box uses some important information about the implementation.

Equivalence partitions are **groups of test inputs that are likely to be processed by the SUTs in the same way**. This can be determined by identifying:

1. target object of method call
2. input parameters of method call
3. other data objects accessed by the method, such as global variables.

Boundary Value analysis is a *test case design heuristic that is based on the observation that bugs often result from incorrect handling of boundaries of equivalence partitions*.

Other heuristics include:

- each valid input at least once in a positive test case
- no more than 1 invalid input in a test case

9 Software Engineering Principles

9.0.1 Law of Demeter

1. An object should have limited knowledge of another object
2. An object should have limited interaction with closely related classes, if foo is coupled to bar, which is coupled to goo, foo should not be coupled to goo
3. Reduces coupling

9.0.2 SOLID

Single Responsibility Principle every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class.

Open-Closed Principle software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"; that is, such an entity can allow its behaviour to be extended without modifying its source code.

Liskov Substitution Principle Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing 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; both should depend on abstractions. Abstractions should not depend on details.

YAGNI a principle of extreme programming (XP) that states a programmer should not add functionality until deemed necessary.

DRY Don't repeat yourself, i.e. No duplicate implementations

Brook's Law Adding people to a late project makes it later

10 Software Development Life Cycles

SDLC consists of different stages such as:

- Requirements
- Analysis
- Design
- Implementation
- Testing

10.0.1 Sequential models

1. Software development as linear process
2. Useful for problems that are well-understood and stable
   - Rarely applicable in real-world projects
3. Each stage provides artifacts for use in next stage

10.0.2 Iterative models

1. Several iterations
2. Each iteration is a new version
   - Each iteration is a complete product
3. Either breadth-first (all major components in parallel) or depth-first (Flesh out some components at a time)
4. Most projects use both, i.e. iterative and incremental process

10.0.3 Agile models

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan
- Requirements based on needs of users, clarified regularly, factored into developmental schedule when appropriate
- Rough project plan, high level design that evolves as the project goes on
- Strong emphasis on transparency and responsibility sharing among members

10.1 Popular SDLC process models

10.1.1 Scrum

- Scrum master
- Development team
- Product Owner
- Divided into Sprints (basic unit of development)
  - Preceded by planning meeting
  - Potentially deliverable product increment is done during Sprint
  - Creates self-organising teams by encouraging co-location of team members
  - Customers can change their minds about their wants and needs
  - Sprint backlog
    - To do
- Daily scrums
  - What did you do?
  - What will you do?
  - Are there any impediments?

10.1.2 Extreme Programming (XP)

1. Stresses customer satisfaction
2. Empowers developers to respond to changing customer requirements
3. Emphasises teamwork
4. Completes software project via:
   - Communication
   - Simplicity
   - Feedback
   - Respect
   - Courage

10.1.3 Unified process

- Inception
  - Understand problem and requirements
  - Communicate
  - Plan

- Elaboration
  - Refine and expands requirements
- Construction
  - Major implementation to support use cases
  - Refine and flesh out design models
  - Testing of all levels
  - Multiple releases
- Transition
  - Ready system for actual production use
  - Familiarise end users with the system

10.2 CMMI (Capability Maturity Model Integration)

- Determine if process of an organisation is at a certain maturity level
- Initial
  - Processes unpredictable, poorly controlled and reactive
- Managed
  - Processes characterized for projects and reactive
- Defined
  - Processes characterized for organisations and proactive
- Quantitatively Managed
  - Processes measured and controllers
- Optimized
  - Focus on process improvement