CS2103T – Software Engineering
AY24/25, Y2S2
Notes

Sim Ray En Ryan

30 April 2025

Contents

1 Software Engineering

Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation. and maintenance of software. Unlike other forms of engineering, it is:

- Invisible and intangible
- Does not wear out
- Easy to modify on a large scale
- Free raw materials, relatively cheap equipment, and can be duplicated infinitely with almost no cost
- Easily moved from place to place
- Higher skilled workers with their own methods
- Difficult to judge quality
- Transcends geographical boundaries
- More fragile than a structures;

2 Programming Paradigms

A programming paradigm guides programmers to analyze programming problems, and structure programming solutions in a specific way.

- Procedural Programming
- Functional Programming
- Logic Programming
- Object-Oriented Programming (OOP)

2.1 OOP

OOP groups operations and data into modular units (objects), and combine the objects in structured networks to form a program. OOP is useful as it abstracts away lower details and work with larger entities. It also provides encapsulation, through packaging and information hiding.

2.1.1 Objects

Every object has a state and behaviour, and an interface and implementation. Objects tend to be based on a real-world counterpart, but does not have to be exactly the same.

2.1.2 Classes

Multiple objects can be instances of the same class and will differ in their attributes (for example, names in Persons). Since they are all Persons, the rest is the same as the behaviour of a Person should be the same.

2.1.3 Class-level Methods

Also known as static methods, they are accessed via the class name rather than an instance of the class. This should occur when the method used is not specific to each instantiation of the class, but rather a property or method of the class as a whole.

2.1.4 Enumerations

Enumerations can be used when a fixed set of values are considered a data type.

2.2 Associations

Connections between classes and objects are known as associations. Associations in an object structure can change over time. Associations between objects can also be generalized to associations between classes.

2.2.1 Navigability

An association does not necessarily mean both objects know about each other. It can be unidirectional or bidirectional, and two unidirectional associations do not add up to become a single bidirectional association.

2.2.2 Multiplicity

The multiplicity dictates how many objects take part in each association. Normal instance-level variables have a multiplicity of 0..1, since it can hold reference to a single object or be null. If this field is instantiated in the class, its multiplicity becomes 1, also known as a compulsory association.

2.2.3 Dependencies

A dependency is a need for one class to depend on another without having a direct association in the same class. It is weaker than an association as this association is only temporary.

2.2.4 Composition

A composition is a strong whole-part relationship. This means that when the reference to the whole is destroyed, the references to the parts are also destroyed. This association cannot have cyclical links. Composition is achieved with normal variables, and usually will not be visible to clients of the 'whole' object.

2.2.5 Aggregation

An aggregation is a weaker composition. It is different in that the contained object can exist even after the container is deleted.

2.2.6 Association Classes

An association class represents additional information about an association. For example, a Marriage class that happens between a Person and a Person

2.3 Inheritance

Inheritance allows the definition of a new class based on existing classes. The Base, Parent, or Superclass derives the Derived, Child, Sub, or Extended class. A superclass is more general than a subclass. Inheritance allows for the extraction of general parts of different classes into a general class, and that the subclass is-a superclass.

2.3.1 Multiple Inheritance

Multiple inheritance is when a class inherits directly from multiple classes. It is not allowed in Java.

2.3.2 Overriding

Overriding occurs when a subclass changes the behaviour inherited from the parent class. The type signature must be the same. The original can be called using super().method().

2.3.3 Overloading

Overloading occurs when there are multiple methods, with the same name, but different type signatures. Overloaded methods are used to indicate that multiple operations do similar things but with different parameters.

2.3.4 Interfaces

An interface is a behaviour specification, and a class implementing an interface is a is-a relation. An interface contains method declarations but not their definitions. An interface cannot store any state variables, but a class can inherit from as many interfaces as needed.

2.3.5 Abstract Classes

An abstract class is abstract because it is merely a representation of commonalities among its subclasses, and cannot be instantiated as its own. A class that has an abstract method immediately becomes an abstract class. A class can only inherit from one abstract class, but can store state variables.

2.3.6 Substitutability

Subclasses can substitute a superclass, but not vice-versa.

2.3.7 Dynamic and Static Binding

Overridden methods are resolved using dynamic binding, For example, animal.move() will depend on what Animal the animal is. In contrast, overloaded methods are resolved using static binding and compile time (since once compiled, there are no variable names, just addresses).

2.4 Polymorphism

Polymorphism is the ability of different objects to respond, each in its own way, to identical messages. It is based on:

- Substitutability
- OVerriding
- Dynamic Binding

3 Requirements

A software requirement is a need to be fulfilled by the software project. A software project can be:

- Brownfield: Replaces or updates an existing software product
- Greenfield: Totally new system from scratch

3.1 Requirements

Requirements come from stakeholders, which includes users, sponsors, developers, etc.

3.1.1 Non-functional Requirements

While functional requirements specify what the system should do, non-functional requirements specify the constraints under which the system is developed. NFRs are easier to miss than FRs, but are equally as critical to the success of a software.

- Data requirements
- Environment requirements
- Accessibility
- Fault Tolerance
- Scalability
- etc

3.1.2 Quality of Requirements

Requirements have to be unambiguous, testable, clear, correct, understandable, feasible, independent, atomic, necessary, and abstract. As a whole, they have to be consistent, non-redundant, and complete.

3.1.3 Prioritization

Requirements should be prioritized based on importance and urgency. For example:

- Essential, High, Must Have
- Typical, Medium. Nice to Have
- Novel, Low, Unlikely to Have

3.2 Gathering Requirements

3.2.1 Brainstorming

Generate many ideas without validating them, putting crazy ideas on the table without fear of rejections.

3.2.2 User Surveys

Solicit responses and opinions from a large number of stakeholders

3.2.3 Observations

Observing users in their natural work environment, such as how an existing system is used or common mistakes made.

3.2.4 Interviews

Interviewing stakeholders and domain experts provides useful information for project requirements.

3.2.5 Focus Groups

An informal interview within an interactive group setting.

3.2.6 Prototyping

A mockup, scaled down version, or partial system. It can be used to not only get feedback, but validate a technical concept, give a preview, or for early field testing.

3.2.7 Product Surveys

Studying existing products by looking for shortcomings or benefits of existing solutions.

3.3 Specifying Requirements

3.3.1 Prose

A textual description, with abstract ideas such as vision.

3.3.2 Feature Lists

A list of features grouped by aspect, priority, order of delivery.

3.3.3 User Stories

Short descriptions of a feature told from the perspective of a user, in the form "As a {user}, I want to {function} so that I can {benefit}.". The benefit can be omitted if it is obvious, and adjectives can also be added to the user. User stories capture user requirements in a convenient way for scoping, estimation, and scheduling. User stories are less detailed, and can also capture NFRs. To make good user stories:

- Clear any preconceived product ideas
- Define the target user as a persona
- Define the problem scope
- List scenarios to form narrative
- List user stories to support these scenarios

3.3.4 Use Cases

An interaction between the user and system for a specific functionality. This is represented by UML with ovals and users. Use cases capture the FRs of a system. A use case can involve multiple actors, an actor can be involved in many use cases, and a single system can play many roles. A use case only describes externally visible behaviour. The Main Success Scenario (MSS) describes the most straightforward interaction given nothing goes wrong. Extensions are added on to describe exceptional or alternative flow of events. Extensions are written with the number it is extending and a letter. Can be asterisk if it applies to any step, such as cancelling an action. The Use cases can include, or extend other use cases. 1a and 1b are two different extensions Preconditions specific the state that the system is expected to be in, and guarantees specify the outcome.

3.3.5 Glossary

All stakeholders have a common understanding of noteworthy terms, abbreviations, and acronyms.

3.3.6 Supplementary Requirements

Used to capture requirements that do not fit elsewhere.

4 Design

4.1 Software Design

- Product and External Design: To meet user requirements
- Implementation and Internal Design: To meet external behaviour

4.2 Design Fundamentals

4.2.1 Abstraction

Only details relevant to the current perspective or task at hand need to be considered. Multiple levels of abstraction can be nested.

- Data abstraction: Abstracting away lower-level data items
- Control abstraction: Abstracting away details of actual flow control

Use abstraction

4.2.2 Coupling

A measure of degree of dependence between components. High coupling is discouraged since:

- Maintenance is harder: Changing one module requires changing other modules
- Integration is harder: Have to integrate all modules together
- Testing is harder: Harder to isolate certain features.

The following are some examples of A coupling to B:

- Content Coupling: A has access to the internal structure of B
- Global Coupling: A and B depend on the same global variable
- Control Coupling: A calls B
- Data Coupling: A receives object B as a parameter
- Subclass Coupling: A inherits B
- External Coupling: A and B follow the same communication protocol
- Temporal Coupling: When A and B happen at the same time

Do not try to remove all coupling. Just reduce as much as possible.

4.2.3 Cohesion

Cohesion measures how strongly related and focused the various responsibilities of a component are. High cohesion is encouraged, since low cohesion:

- Lowers understandability of modules
- Lower maintainability
- Lowers reusability as it does not represent logical units of functionality

This can be achieved by:

- Keeping code related to a single concept together
- Keeping code that is invoked close together (initializing) together
- Keeping code that is related to the same data structure togerher

Increase.

4.3 Modelling

A model provides a simpler view of a complex entity by only capturing selected aspects. Multiple models of the same entity may be needed to capture it fully. Models are useful to:

- Analyze a complex entity
- Communicate information among stakeholders
- As a blueprint for creating software.

4.3.1 UML Models

Unified Modeling Language was created by the Three Amigos, and can be split into UML Object diagrams and UML Class diagrams.

4.3.2 Class Diagrams

Class Diagrams describe the structure, but not the behavior of an OOP solution.

Classes are represented with the Class name, attributes, then methods in that order. Attributes and methods can be omitted if not important. The following prefixes indicate the visibility of each attribute or operation:

- +: Public
- -: Private
- #: Protected
- ~: Package Private
- There is no default visibility. A lack of specification means that it is unspecified.

Generic classes have their type parameter written in a box extending from the upper left corner of the class name.

Class level members are denoted with an underline.

Associations are drawn using solid lines between two classes, or can be shown as an attribute of the class. While a line is preferred, do not use both.

Labels describe the meaning of the association, and indicates the direction of the association.

Roles describe the role played by the classes

Multiplicities are shown using a numbers, .., and * to indicate any number. For example, 0..1 is optional, 1 is compulsory, 1..* is one or more. There is no default multiplicity.

Navigability is shown by the arrowhead

Notes are used to add additional information, in the form of a box connected to a class using dashed lines.

Inheritance is shown using a triangle pointing to the parent class. The triangle may or may not be filled.

Composition is shown using a solid diamond symbol on the whole.

Aggregation is shown using a hollow diamond symbol. Aggregation may add more confusion than clarity.

Dependancy is shown using a dashed arrow.

Enumerations are shown with an ¡¡enumeration¿¿ header, its name, followed by its possible values.

Abstract classes and methods can be shown using italics or the {abstract} keyword.

Intefaces are shown using the ¡¡interface¿¿ keyword, and inheritance is shown using a dashed line.

Association classes are shown by drawing a dashed line below the association link of two classes

4.3.3 Object Diagrams

Object diagrams show the object structure at a given point in time. The class name and object name are underlines, with car1:Car. There is only the attribute compartment, which can be omitted. The name can also be omitted to have :Car if car1 is not important.

4.3.4 Conceptual Class Diagrams

A CDD does not contain solution specific classes, without any of the implementation. It represents the class structure of the problem domain, and is a subset of a class diagram notation as it omits methods and navigability.

4.3.5 Deployment Diagrams

A deployment diagram shows the physical layout of the software.

4.3.6 Component Diagram

A component diagram shows how a system is divided into components.

4.3.7 Package Diagram

A package diagrams shows packages and their dependencies

4.3.8 Composite Structure Diagrams

A composite structure diagram shows a class's internal structure.

4.3.9 Activity Diagrams

An AD starts from a dot, with an arrow pointing to the next action.

Branches are represented by a diamond shape, and each path has [guard conditions]. Exactly one guard condition must be true. A branch is closed using a merge node, also represented by a diamond shape. The merge node as well as [Else] can be omitted.

Forks and joins are represented by a horizontal line, and denote the start and end of outgoing and incoming edges. All outgoing edges must eventually join back.

Rakes are used to represented nested diagrams, that this subsidiary activity diagram is shown elsewhere.

Swimlane Diagrams partition the actions by actors.

4.3.10 Sequence Diagrams

Sequence diagrams model interactions between various entities in a system and scenario. Time moves from top to bottom. It contains the following:

- Entities including actors or components
- Activation Bar where method is executed (Not broken during nested calls)
- Operation: Method calls are solid arrows, pointing to start of activation bar
  - For constructors, the arrow will point to the entity, and the bar extends below
- Possible return value: Method returns are dashed arrows, pointing from end of activation bar
- Lifeline showing that the instance is still alive

Non-ambiguous activation bars or return arrows may be omitted.

Loops 'loop' is written at the top right, followed by the [condition]. Then, the looped section is encased in that box.

Self References and Call Backs For self references, a bar is drawn adjacent to the original, with arrows for operations and possible return values pointing at itself. For call backs, a bar is also drawn adjacent to the original, this time with arrows point from original object.

Conditionals For alternative paths, 'alt' is written at the top right, followed by [conditions], each sub-box separated by a dotted line. Only one sub-box can occur. For optional paths, 'opt' is written at the top right, followed by [condition]

End An X is used at the end of a lifeline to show its deletion. In Java, this refers to when it is no longer being references and is ready to be garbage-collected

Static Methods ¡¡class¿¿ is used to denote that this entity is the class itself, and not an instance. An additional column is made.

Parallel Paths 'par' is written at the top right. Programs written with these are likely multi-threaded

Reference Frames 'ref' is written at the top right, followed by a name. Then. 'sd' is written at the top right, followed by the same name to extract out certain sequence diagram details.

4.3.11 Use Case Diagrams

Model the mapping between features of a system and its user roles.

4.3.12 Timing Diagrams

Focuses on timing constraints.

4.3.13 Interaction Overiew Diagrams

A combination of activity and sequence diagrams.

4.3.14 Communication Diagrams

Are sequence diagrams with an emphasis on data links instead of sequences of interactions.

4.3.15 State Machine Diagrams

Models state induced behavior, such as calling the same method on the same type over and over again.

4.4 Software Architecture

A software architecture shows the overall organization of the system.

4.4.1 Architecture Diagrams

A very high level design of a system. There is no fixed style for architecture diagrams. For larger projects, diagrams may need to be split up to show levels right below it. There is no standard, but symbols and double-headed arrows should be minimized.

4.4.2 Architectural Styles

n-tier Higher layers makes use of services of the lower layers

Client-server Such as with a game or web application

Transaction Processing divides the workload of the system to number of transactions, gives it to a dispatcher, which sends them to executors.

Service oriented builds applications by combining functionalities of programmatically accessible services.

Event Driven controls flow by event emitters and communicating those events to event consumers.

4.5 Software Design Patterns

In software development, there are certain problems that recur in a certain context, and can be defined:

- Context: Where the situation occurs
- Problem: The main difficulty to be resolved
- Solution: The core of the solution
- Anti-patterns: Commonly used solutions which are incorrect
- Consequences: Pros and Cons

4.5.1 Singleton Pattern

When a class should have no more than one instance, it is a singleton. A problem arises since a normal class can be instantiated multiple times. The solution is to make the constructor private. A static method is used to either create this object only once, or to return the already created instance. It is easy to apply, effective with minimal effort. However, the singleton now acts like a global variable, and is difficult to replace the Singleton object with a stub.

4.5.2 Abstraction occurrence pattern

When multiple object share a lot of features, but also differ. Using an instance for each results in too much information and difficulties in managing. An anti-pattern is by making new classes for each of these objects, which causes problems as now there are too many classes. A solution will be to represent it with two classes.

4.5.3 Facade pattern

When a component needs to access deep functionality, but access should not expose internal details. A Facade class can be used as a wrapper for inner functions or attributes, such as Storage in iP.

4.5.4 Command pattern

When there are many commands that can be executed, but the code should be able to execute them without knowing what commands they are. This can be solved using a general

Command type object.

4.5.5 Model view controller pattern

Many programs have highly interlinked storage, display, and modifying capabilities. This results in high coupling. This can be decoupled by separating into the three logical components:

- View: Display, interacts with user, and pulls relevant data
- Controller: Detect UI events
- Model: Stores the relevant data.

4.5.6 Observer pattern

When objects need to know a change happens to another object. The observed object should not be coupled with the observing object, and thus the communication can happen through an interface known to both parties

4.5.7 Others

The above design patterns can be combined, along with:

- Creational: Separates an application from how its objects are created
- Structural: Compositing objects into larger structures
- Behavioral: Defining how objects interact and distributing responsbility.

4.6 Design Approaches

4.6.1 Multi-level Design

Represent a program by various high and lower levels.

4.6.2 Top-down and Bottom-up design

Top-down by first defining the high-level operations first. Useful when designing big and novel systems.

Bottom-up by making lower-level components first then putting then together. Useful when there are already existing components.

Usually, will be a mix of the two.

4.6.3 Agile Design

By letting a system design emerge overtime, with only a initial architectural modeling at the beginning.

5 Implementation

5.1 IDEs

Integrated Development Environments may have the following features:

- Syntax Coloring
- Auto Completion
- Code Navigation
- Error Highlighting
- Code Generation
- Compiler or Intepreter
- Debugger

5.1.1 Debugging

The process of discovering defects in a program.

Print debugging is bad since you have to insert and remove print statements, introducing errors, and may be forgotten about and difficult to trace.

Manual tracing is also bad since it is difficult, time consuming, and error prone. You are unlikely to spot the bug if you did not spot it while writing code.

Debuggers allow stepping through code and examining state.

5.2 Code Quality

Produce code to be of high quality, and to maximize readability.

5.2.1 Lines of Code

The less, the better. Avoid very long methods (30 LoC) and deep nesting (3) by abstracting out functions. This is also related to Single Level of Abstraction Principle, where different level of code can be extracted to their relevant levels.

5.2.2 Making Code Obvious

Do not try to calculate everything in one line (although it is possible). Use a named constant instead of magic numbers. Use explicit type conversion, all necessary brackets, enumerations, etc.

5.2.3 Structure

Structure code logically by adding indentation and line spaces appropriately, such as by separating groups of related statements. Avoid:

- Unused parameters
- Similar things that look different
- Different things that look similar
- Multiple statements in the same line

5.2.4 On Optimization

Optimizing prematurely is not effective, as part optimized may not be the bottleneck, and will regardless complicate the code, and may actually be slower than if the compiler did it. In general, make it work, make it right, then make it fast.

5.2.5 Happy Paths

Should be made prominent. Can be achieved using guard clauses or continues.

5.2.6 Standards

Ultimately, everyone should follow the same style guide to make it look like it was written by one person. Refer to CS2103T Coding Standard.

5.2.7 Naming Standards

- Use nouns for things (classes and variables)
- Use verbs for actions (methods and functions)
- Use plurals for mult-valued variables
- Have descriptive names (isValid instead of flag)
- Do not use numerals to differentiate variables
- Use standard English, avoid slang and foreign words

5.2.8 Others

Switch cases should always include the default branch, and it should be for the intended default action.

Recycling variables should not be used as it is the same when compiled.

Dead code should be removed.

Catch statements should not be empty.

Scope of Variables should be minimized. Only declare when it is first used.

Repeat code should be abstracted

Comments should be sufficient and minimal. They should not rephrase the code, but should be meant for other programmers explaining what the code is for and why, without the how.

5.3 Refactoring

Restructuring of code in small steps without modifying external behavior. It allows hidden bugs to become easier to spot, and improve performance since the compiler may be able to optimize it better. Some common refactors (from the refactoring catalog) include:

- 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 Loops
- Split Temporary Variable

5.4 Documentation

Documentation is written for both users and maintainers, and is best kept in a text format. It should be top down, breadth first explanation, accurate, comprehensive, and comprehensible. Use diagrams and examples.

5.4.1 JavaDoc

A tool for generating API documentation in HTML format from comments in Source Code

5.5 Error Handling

5.5.1 Exceptions

Deal with unusual but expected situations. In Java, the code being executed creates an exception object, which is handled in the runtime system by something in the call stack.

5.5.2 Assertions

Define assumptions about a program state so that the runtime can verify it. Assertions are disabled by default in Java, and can be enabled with -enableassertions flag. Use liberally as it does not really affect performance, and make sure code still works when assertions are disabled.

5.5.3 Logging

In Java, can use java.util.logging.*. There are multiple levels:

- Level.INFO
- Level.WARNING

These levels can be used with the logger.log(level, String message) method to log things about a program.

5.5.4 Defensive Programming

If things can go wrong, it will go wrong. Write code to prevent such scenarios from happening. It is not necessary to be defensive all the time.

5.5.5 Design by Contract

It is responsibility of caller to ensure all preconditions are met.

5.6 Integration

Combine parts of a software to form a whole. Can be done with "Late and one time"" (not recommended) or "Early and frequent". Related to "Big Bang" and "Incremental integration".

5.6.1 Incremental Integration

Top Down integration is when higher level components are integrated first.

Bottom up is the opposite.

Sandwich is the middle of the two.

5.6.2 Build Automation

Build scripts can compile, link, and package all source codes together. Some build tools also have dependency management tools.

Continuous Integration is where integration, building, and testing happens automatically after each code change

Continuous Development is CI but also deployed to end-users at the same time

5.7 Reuse

Reusing tried and tested components, such as libraries. While practical, it has the following tradeoffs:

- Overkill solution for a simple implementation
- Unstable library code
- Unmaintained library code
- Licenses and restrictions
- Bugs, missing features, vulnerabilities, and malicious libraries

5.7.1 APIs

Application Programming Interfaces specifies the interface which other programs can interact with a software component.

5.7.2 Libraries

A collection of modular code that can be used by other programs, such as String and ArrayList. Your code calls the library code.

5.7.3 Frameworks

Reusable implementation of a software providing generic functionality. It is different from libraries as the framework calls your code, and are meant to be customized and extended unlike libraries.

5.7.4 Platforms

Platforms provide a runtime for applications, such as OS, JavaEE, or .NET.

5.7.5 Cloud Computing

The delivery of computing as a service over the network, where hardware and software are located and managed by another place (server farm).

Infrastructure as a service Iaas delivers computer infrastructure as a service

Platform as a service PaaS provides platforms for developers to build on

Software as a Service SaaS allows applications to be accessed over the network.

6 Quality Assurance

6.1 QA

Ensuring that the software being built has the required levels of quality. Is composed of:

- Validation: Are the requirements correct?
- Verification: Are the requirements implemented correctly?

6.1.1 Code Reviews

Systematic Examination of code with intention of finding where code can be improved. It can detect functionality defects as well as coding standard violations, as well as verify non-code artifacts and incomplete code. Includes:

- Pull Request Reviews
- In pair programming
- Formal inspection

6.1.2 Static Analysis

Code is analyzed without running, to look for unused variables, unhandled exceptions, etc. High end tools can locate potential bugs, memory leaks, inefficient code.

6.1.3 Formal Verification

Using mathematical techniques to prove correctness of program. Can be used to prove absence of errors, commonly used in safety-critical software. CS3234 moment.

6.2 Testing

Executing a set of test cases. At minimum, specifies the input to the software under test and the expected behavior or output. A test case failure indicates a potential bug. A programs testability is an indication of how easy it is to test an SUT.

6.2.1 Regression

Modifying a system may result in unintended and undesirble effects known as regressions. Regression testing is retesting of software to detect these. Usually automated.

6.2.2 Developer

Tests done by developers themselves as opposed to dedicated testers or end users. It is easier to do early (and thus by developers) since there is a lower search space, and bugs may hide other bugs. The cost of fixing bugs increases as the software gets closer to development

6.2.3 Unit

Testing each unit (method, class, subsystem) to ensure each part works correctly. A proper unit test should test the unit in isolation. A stub will isolate the SUT from its dependencies, can be made by creating Stub classes that are very simple and should not contain bugs.

6.2.4 Integration

Testing when different parts work together.

6.2.5 System

Testing the whole system based on the specified external behavior of the system, as well as NFRs, including:

- Perfomance testing: Load speeds, response times
- Load testing: Ensure system works under heavy load
- Security testing: Hacking
- Compatibility testing: Test if system works with other systems
- Usability testing: Test intuitiveness of system
- Portability testing: Test if system works in multiple environments

6.2.6 Alpha Beta

Alpha testing is done by users under controlled conditions.

Beta testing is done by a subset of users under natural conditions.

6.2.7 Dogfooding

Developers using their own product. Observations need to be deliberately collected and processed.

6.2.8 Exploratory vs Scripted

A mix of both testings is better.

Exploratory testing derives test cases on the fly, and creates new test cases based on past results. Its efficacy is based on tester's prior experience and intuition

Scripted testing is a set of all test cases based on expected behaviour of SUT. It is more systematic, and likely to discover more bugs given sufficient time.

6.2.9 Acceptance

Testing system to ensure it meets user requirements. It has a higher focus on positive test cases (showing the program can do what it is supposed to do)

6.2.10 Automation

An automated test case can be ran programmatically.

CLI apps can be automated by redirecting inputs and outputs

Test Drivers are code that drives the SUT

JUnit is a tool for automated testing of Java programs

GUI testing is much harder than CLI or API testing. Moving as much logic as possible out of GUI should make GUI testing easier. Some automated GUI testing tools include TestFX, Visual Studio's record replay, Selenium.

6.2.11 Coverage

- Function coverage: How many functions out of how many tested
- Statement Coverage: How many statements or LoC tested
- Branch Coverage: All possible branches tested
- Condition Coverage: All subexpressions in conditions tested.

Can be measured with coverage analysis tools.

6.2.12 Dependency Injection

Injecting objects to replace current dependencies with a different object. Often used with stubs to isolate SUT from dependencies. This can be achieved using polymorphism, where both Class and ClassStub is a ParentClass.

6.2.13 TDD

Test Driven Development is writing tests before writing the SUT, while evolving the functionality and tests in small increments.

- Decide behavior to implement
- Write test case to test behavior
- Tests should fail
- Implement behavior
- Rewrite behavior until all tests pass

6.3 Test Case Design

Exhaustive testing is not practical because it requires infinite test cases. Test cases should be designed to be effective (find bugs) and efficient (less test cases for same amount of bugs). Thus, test cases should be targeting potential faults that are not already addressed by existing test cases.

6.3.1 Positive and Negative

Positive test cases are tests designed to produce an expected behavior.

Negative test cases are designed to produce a behavior that indicates an invalid or unexpected situation.

6.3.2 Black and Glass Box

Black box , also known as specification based or responsibility based approached, is where test cases are designed exclusively based on SUTs specified external behavior.

Glass box , also known as white box or structured or implementation based approach, is where test cases are designed based on the code of SUT.

Gray box is where the design uses some important information about the implementation.

6.3.3 Equivalence Partitions

Most SUT process all possible inputs in a small number of distinct ways. EP is a technique where dividing possible inputs into partitions results in less redundant testing, as well as data coverage.

6.3.4 Boundary Value Analysis

Bugs in programs usually y arise from incorrect handling of boundaries. BVA suggests that tests with values near the boundaries are more likely to find bugs. This is usually one value in the boundary, one value right above the boundary, and one right below.

6.3.5 Combining Test Inputs

When SUT have different combinations of inputs, testing each possible combination is inefficient.

- All Combinations
- At least once: Each test input should be used at least once
- All Pairs: Identify most important 2 parameters, and brute force those
- Random: Use one of the above, and pick randomly

7 Project Management

7.1 Revision Control

Revision control is the process of managing multiple versions of a piece of information. This helps with tracking history and evolution of project, collaboration, recovering from mistakes, and working simultaneously on multiple versions of a project. A repository is a database that stores the revision history.

7.1.1 History

In a repo, certain files can be set to be ignored (typically passwords or log files). After that, changes can be added to a staging area, then committed. Each commit is identified by a hash, and can be tagged. Commits can also be diff to find the difference. To restore a state, a previous commit can be checkout.

7.1.2 Remote Repository

A repo can be cloned, and the original repo is known as the upstream. You can also pull from one repo to another, to copy commits in the first to the second. The two repos must have a shared history. You can also push new commits from one repo to another, similar to pulling but in the other direction. A fork is a remote copy of a remote repo. A pull request is a mechanism to contribute code to a remote repo.

7.1.3 Branching

Branching is the process of evolving multiple versions of the software in parallel. A branch can be merged into another branch. Merge conflicts occur when trying to merge two branches that have modified the same code.

7.1.4 CRCS vs DRCS

A centralized CRS is where everyone shares one repo. A decentralized RCS allows multiple remote and local repos working together.

Forking flow: The main repo is a remote repo, each member forks the main, and creates pull requests from their fork to the main.

Feature branch flow: Each feature is done in a separate branch on each local repo.

Centralize flow: All changes are done on the master branch.

7.2 Project Planning

7.2.1 Work Breakdown Structure

A WBD depicts information about tasks and their details in terms of subtasks. It may also track effort, which is measured in man time. The tasks should all be well defined.

7.2.2 Milestones

A milestone is the end of a stage which indicates significant progress. Each intermediate product is a milestone.

7.2.3 Buffers

A buffer is time set aside to absorb unforeseen delays.

7.2.4 Issue Trackers

Used to track task assignment and progress.

7.2.5 Gantt Chart

A 2D Char, drawn as time against tasks. A solid bar represents the main task, while a gray bar represents a subtask. A diamond indicates an important deadline.

7.2.6 Program Evaluation Review Technique

A PERT chart uses graphs to show sequence of tasks. It also shows which tasks can be done concurrently, or the shortest and critical paths.

7.3 Teamwork

7.3.1 Egoless Team

Every team member is equal, and a consensus must be reached on ever decision.

7.3.2 Chief Programmer Team

There is a single authoritative figure, who is assisted by experts.

7.3.3 Strict Hierachy Team

Each member has a fixed assigned tasking, and only reports to one boss. This reduces communication overhead.

7.4 SDLC Process Models

Sofware Development Lifecycle includes requirements, analysis, design, implementation, and testing.

7.4.1 Sequential Models

This model is linear, and follows a strict order of:

- Specify Requirements
- Design product
- Implement Product
- Test Product (and fix bugs)
- Deploy Product

This model works better for well-understood problems, but may not be that effective in real world problems.

7.4.2 Iterative Models

Produce software by going through several iterations. This is similar to a sequential model but repeated many times. Along with the two models below, a mixture of them can be used.

Breadth First: Evolve all major components and functionality in parallel. Each iteration produces a working product.

Depth First: Fully flesh out one component.

7.4.3 Agile Models

Requirements are prioritized based on needs of user, and clarified regularly with the whole team. A high level rough design is used from the start. Transparency and responsibility is emphasised.

7.4.4 Extreme Programming

Delivering software as the users need, allowing for responses to changes in customer responses.

7.4.5 Scrum

A process skeleton that contains roles:

- Scrum master: Project Manager
- Product Owner: Represents stakeholders
- Team

Each project is divided into iterations known as sprints, which are preceded by a planning meeting, and creates a potentially deliverable product increment.

7.4.6 Unified Process

- Inception: Requirements, Planning
- Elaboration: Refine, High Level
- Construction: Implementation, Release
- Transition: Deploy, Familiarize

7.4.7 CMMI

Capability Maturity Model Integration states the five maturity levels for a process and provides criteria to determine if the process of an organization is at a certain maturity level:

- Initial
- Managed
- Defined
- Quantitatively Managed
- Optimizing

8 Principles

8.1 Single Responsibility Principle

Each class should only have one responsibility.

8.2 Open-Closed Principle

Classes should be open for extension, but closed for modification.

8.3 Liskov Substitution Principle

A subclass should not be more restrictive than its superclass.

8.4 Interface Segregation Principle

No client should be forced to depend on methods it doesn't use.

8.5 Dependency Inversion Principle

High level should not depend on low level. Details should depend on abstractions.

8.6 SOLID Principles

The five principles above.

8.7 Separation of Concerns Principle

Separate code into distinct sections, reducing overlaps and coupling, increasing cohesion.

8.8 Law of Demeter

Principle of least privilege. Objects should have limited knowledge of another object, and only interact with objects it needs to interact with.

8.9 YAGNI Principle

Do not add code "just in case" it may be needed: You aren't gonna need it.

8.10 DRY Principle

Don't repeat yourself.

8.11 Brooks' Law

Do not add people late into a project, as that makes the project later.