# Object-Oriented Programming

What is OOP?

* A programming paradigm, guides programmers to analyse and structure their programs in a specific way
* Elements of design - objects and object interactions
* Java, JavaScript, Python support OOP programming
* (C – procedural programming, F#, Haskel – functional programming, Prolog – logic programming)

Objects

* Have both state (data) and behaviour (operations on data)
  + Interface (what other objects interact with), and implementation (supports the interface, not necessarily accessible to other objects)
* Model components of real systems (not exactly, but similar)
* Properties of objects
  + Abstraction: Abstract away lower level details, work with bigger entities
  + Encapsulation
    - (packaging) packs related data and behaviour into one self-contained unit
    - (info hiding) hides info from unintended users, only accessible through interface

Classes

* Contains instructions for creating a specific kind of object
* Each class member has the same attributes, but with different values
* Class-level members
  + Class-level attributes – a variable shared by all objects in that class (static)
  + Class-level methods – a method that is not specific to each object in that class

Enumerations

* A fixed set of values that can be considered a data type

Associations

* Connections between objects/classes for them to interact with each other
* Implemented using instance level variables
* Bidirectional associations require matching variables in both classes

Types of Associations

class Logic {

Minefield minefield; // 0..1 multiplicity

Config cg = new Config(); //1 multiplicity

}

class Minefield {

Cell[10] cell = new Cell(); // 10 multiplicity

…

}

* Navigability
  + The concept of which class in the association knows about the other class
* Multiplicity
  + How many objects take part in each association
* Dependency
  + One class depending on another without having a direction association with it
  + Can be caused by transient interactions between objects (does not keep the object it receives, eg received as a method parameter)
* Composition
* An association that represents a strong whole-part relationship
  + When the whole is destroyed, the parts are destroyed as well
* Aggregation
* Represents a container-contained relationship
  + Contained objects can survive without the container
* Implementation: eg container has an attribute Person, and a method setPerson(Person p)
* Association Classes
  + A class that represents additional info about an association (a design role)

Inheritance

* Allows classes to be defined based on an existing class
* Base/parent/super class <-> derived/child/sub/extended class
  + Common parts among similar classes can be extracted into more general classes
* Multiple inheritance
  + Not allowed in Java and C#, but allowed in Python, C++
* Overriding
  + A subclass re-implementing a method from the parent class
  + Same name, type signature, return type
* Overloading
  + Multiple methods with the same name but different type signatures (order of parameter type)
    - eg (int, double) vs (double, int)
* Interface (“implements”)
  + A collection of method specifications (method signature without implementation)
    - Abstract method: A method signature without a method implementation
    - Interfaces can only have abstract methods
  + A Java class can implement multiple interfaces
* Abstract classes (“extends”)
  + Cannot be instantiated, but can be sub-classed
    - A representation of commonalities among its subclasses
    - Can have both abstract and non-abstract methods
  + A Java class can only extend one abstract class
* Substitutability
  + Child class object can be substituted where a parent class object is expected
  + eg Staff staff = new Teacher(); // teacher is a subclass of staff
* Dynamic vs static binding
  + Dynamic binding (aka late binding)
    - Method calls are resolved at runtime
    - Overridden methods are resolved using dynamic binding
  + Static binding (aka early binding)
    - Method calls resolved at compile time
    - Overloaded methods are resolved using static binding

Polymorphism

* The ability of different objects to respond to identical messages differently
  + By targeting superclass objects, and based on the actual subclass responds differently
  + eg calling a method defined in superclass Animal, that is overridden in Dog and Cat subclasses and responds differently
* How
  + Substitutability
  + Overriding
  + Dynamic binding

# Design

What: Product design and internal design

Abstraction

* Establishes the level of complexity we are interested in and suppresses more complex details below that level
* Only shows details relevant to the current perspective or task at hand
* Examples
  + Data abstraction: abstracting lower level data items (eg ignoring attributes within an object)
  + Control abstraction: abstracts details of actual control flow
  + Architecture: a higher-level abstraction of the design of the software
  + Classes: abstraction over related data and behaviours

Coupling

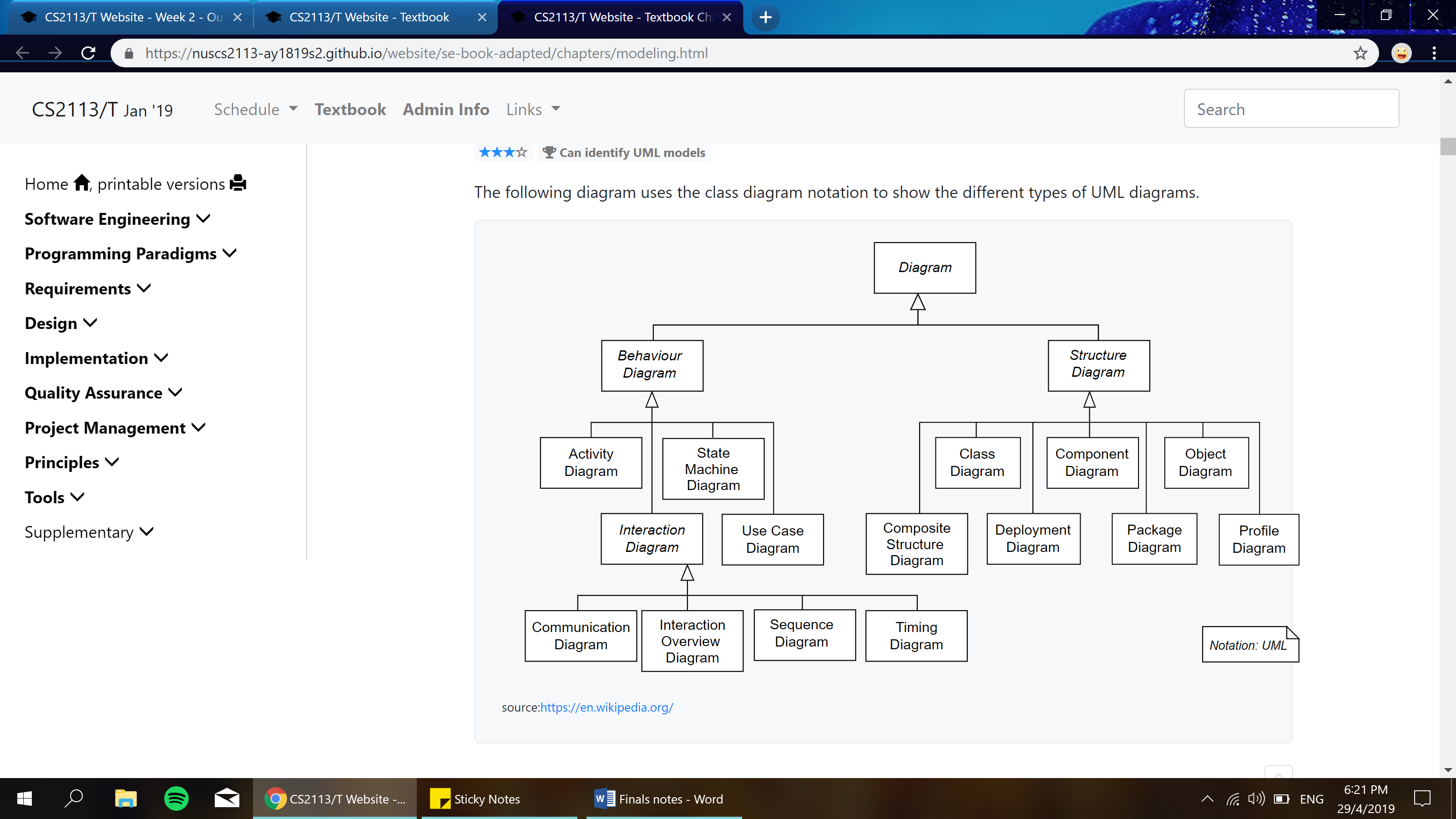
* A measure of degree of dependence between components/classes/methods/etc
* High/tight/strong coupling is discouraged:
  + Maintenance: a change in one module affects other modules
  + Integration: makes integration more difficult as multiple couple components must be integrated tgt
  + Testing and reuse of module more difficult
* How to reduce coupling
  + X is coupled to Y if a change to Y could *potentially* require a change in X
* Example of coupled classes:
  + A has access to the internal structure of B
  + A and B depend on the same global variable
  + A calls B
  + A receives an object of B as a parameter or return value
  + A inherits from B
  + A and B follow the same data format or communication protocol

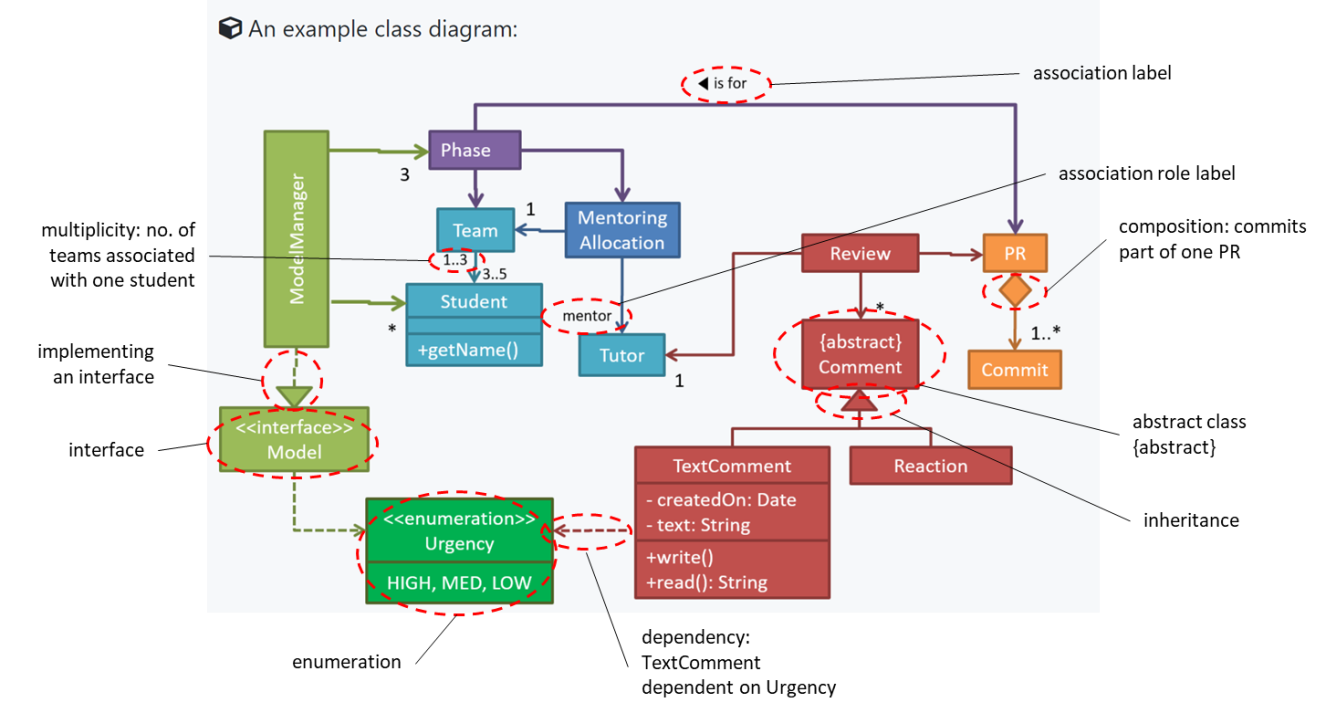
Cohesion

* Measure of how strongly-related and focused the various responsibilities of a component are
* Higher cohesion is better
  + Increases understandability of modules
  + Increases maintainability
  + Increases reusability of module as each one represents a logical unit of functionality

Modelling

* What: a representation
* Why: provides a simpler view of a complex entity, captures only a selected aspect of behaviour
* How they are used:
  + To analyse a complex entity related to software development
  + To communicate information among stakeholders
  + As a blueprint for creating software
* UML models



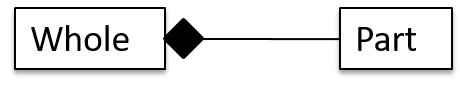
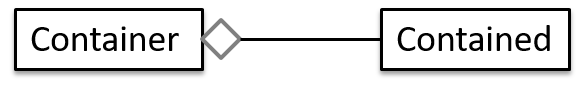
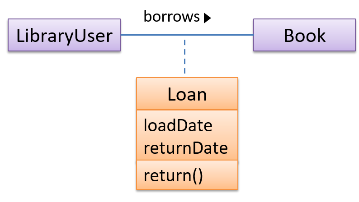
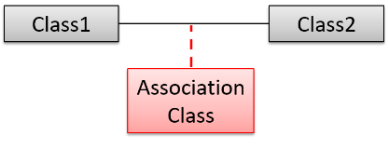
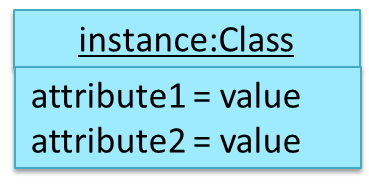
*Modelling Structures*

* Class Diagrams
  + A screenshot of a cell phone

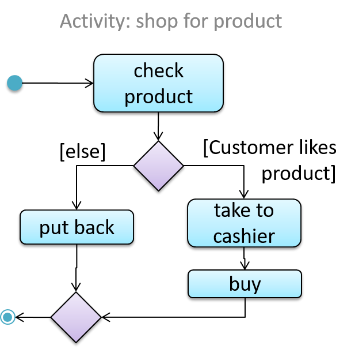
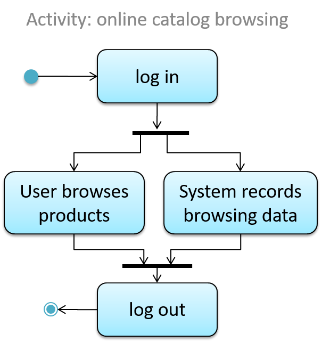
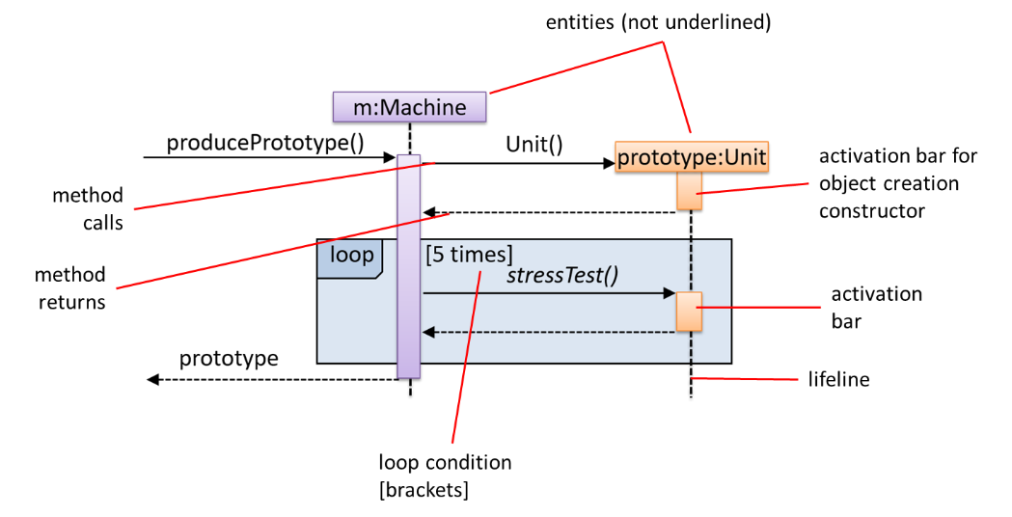
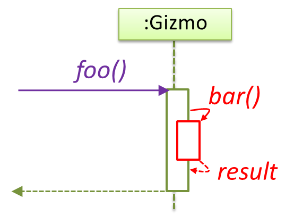
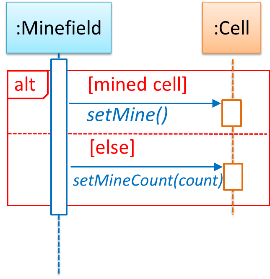
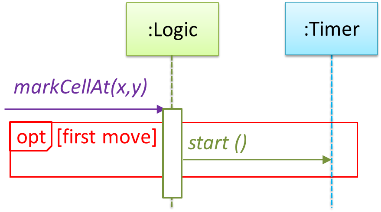
    Description generated with very high confidenceRepresenting a class:

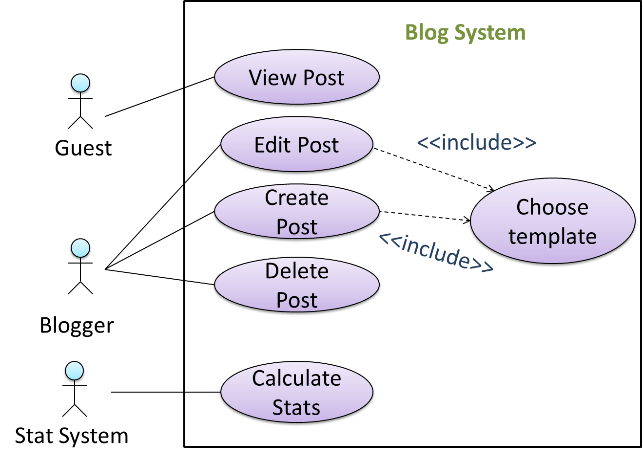
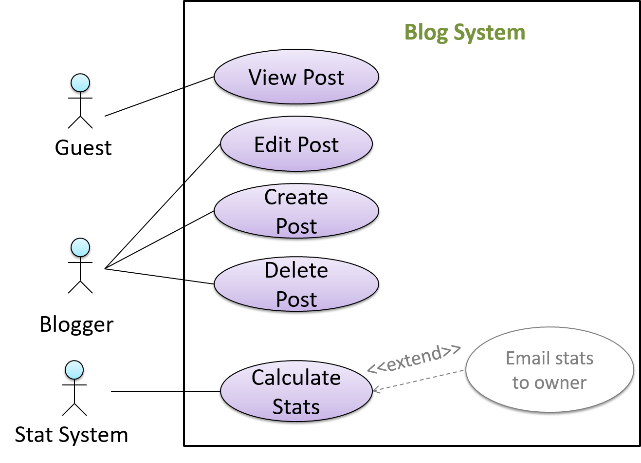
A screenshot of a cell phone

Description generated with very high confidence

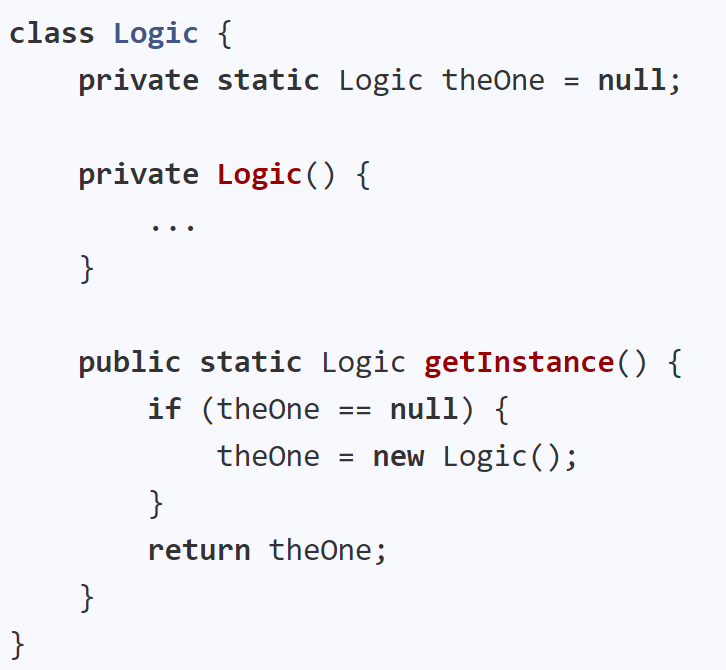
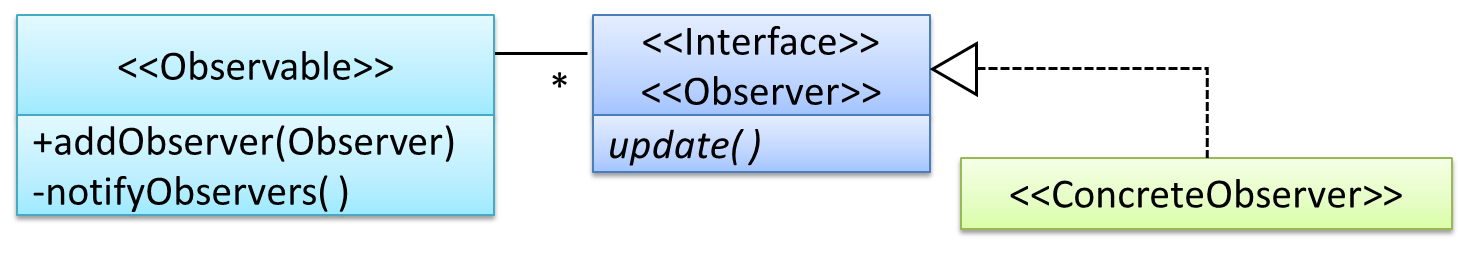
* + Visibilities
    - + : public
    - - : private
    - # : protected
    - ~ : package private
  + Class-level members
    - Underline to denote class-level attributes or methods
  + Associations
    - A solid line to represent an association
    - Labels: describe the meaning of the association, is read in the direction of the arrow head
    - Role labels: indicate the role played by the classes in the association
    - Multiplicity: located closer to the class, indicates how many of that class are associated with one object of the other class
    - Navigability: arrowhead pointing from the class that is aware, to the other class
  + Inheritance
    - Triangle and solid line to indicate class inheritance
  + Composition
    - Solid diamond symbol
  + Aggregation
    - Hollow diamond symbol
  + Dependency
    - Dashed arrow
  + Abstract classes/methods
    - {abstract} ClassName
    - +method1() {abstract}
  + Interfaces
    - <<interface>>, implemented using a dashed line
  + Association classes
* Object Diagrams
  + Representing an object
    - Underline class name and object name
    - Object name can be omitted if unnamed eg :Car
    - Object association: solid line between objects
    - Omit methods
    - Omit multiplicities
  + Multiple object diagrams can correspond to a single class diagram

*Modelling Behaviours*

* Activity diagram
  + Models workflows
  + Notation
    - Start node and end node
    - Action: rounded rectangle
    - Control flow: an arrow showing the flow between actions
    - Branch node, merge node: diamonds
    - Guard condition: in square brackets
    - Fork node and join node: a bar; indicates actions that happen in parallel
* Sequence diagrams
  + Shows the interactions between multiple objects for a given scenario
  + Notations
    - Activation bar must start right before method is called, end right after method returns
    - Activation bar should remain unbroken from the point method is called till it returns
    - Object deletion
      * X at the end of lifeline
      * Java supports automatic memory management, automatically deletes objects
    - Self-invocation: a method of an object calling another of its own methods
    - Alternative paths: if-else conditions
    - Optional paths: if conditions
* Use case diagrams
  + Models the mapping between features of a systems and its user roles



Design Patterns

* An elegant reusable solution to a commonly recurring problem in a given context in software design
  + Some recurring design problems include: What is the best architecture? How do I update the UI without coupling the backend to the UI?
  + Solutions are discovered and refined over time, and are known as “design patterns”
* Format
  + Context, problem, solution (general details), anti-patterns (opt; commonly used solutions that are inferior to the design pattern), consequences (opt; pros and cons), other useful info (opt)
* Singleton design pattern
  + Context: Certain classes should only have one single instance (singletons)
  + Problem: A normal class can be instantiated multiple times using the constructor
  + Solution: Make the constructor private, provide a public class-level method to access the single instance
  + Pros: Easy to apply, effective, easy to access the singleton object from anywhere in the code base
  + Cons: The singleton object acts as a global variable and increases coupling across the code base, difficult to replace with stubs in testing, difficult to make tests independent
* Façade pattern
  + Context: Components need to access functionality deep inside other components
  + Problem: Access to the components should be allowed without knowing its internal details
  + Solution: Use a façade class that acts as a go-between the component internals and its users
    - eg For a library: use a LibraryLogic class for UI to access books in the library
* Command pattern
  + Context: A system has to execute a number of commands, each doing a different task
  + Problem: Some part of the code should execute these commands without knowing their type
  + Solution: Have a general <<Command>> object that can be passed around, stored and executed without knowing the type of command, via polymorphism
    - Command class should be an abstract class or interface
* Model View Controller (MVC) pattern
  + Context: Applications that support storage/retrieval of info, displaying info to the user on multiple UIs, and changing the storage info based on external input
  + Problem: High coupling from abovementioned interlinked nature
  + Solution: Decouple data, presentation and control logic by separating them into 3 components: Model, View and Controller
    - UI typically combines view and controller
* Observer pattern
  + Context: Object(s) that want to ‘observe’ another object
  + Problem: Observed objects do not want to be coupled to objects observing it
  + Solution: Force communication through an interface known to both parties
* Pros and cons of design patterns:
  + Pros: provide high-level vocabulary to talk about design; increases experience in knowing the context and patterns of common problems
  + Cons: overhead of adding more classes or increasing the levels of abstraction (caution for overusing patterns)

Design Approaches

* Multi-level design
  + Design of bigger systems shown on multiple levels (eg high-level design, UI, Logic, Storage)
* Top-down vs Bottom-up design
  + Top-down: Design high-level first (normally for big systems)
  + Bottom-up: Design lower level components first (if redesigning an existing system)
  + Typical approach: a mix
* Agile design
  + Evolves over time; contrasted with full upfront design

# Software Architecture

What

* The structure(s) of the system – software elements, their externally visible properties and the relationships among them (very high-level design)

Architecture diagrams

* Free-form, no standard notation
* Basic guidelines
  + Minimise the variety of symbols
  + Avoid misusing double-headed arrows to show component interaction

Architectural styles

* N-tier
  + Higher layers make use of services provided by lower layers
  + Lower layers are independent of higher layers
  + eg Communication between OS and network software
* Client-server
  + At least one component that plays the role of server, with at least one client component accessing the server’s services
  + Often used in distributed applications (eg game clients & game server, browsers and web server)
* Event-driven
  + Controls the flow of the application by detecting events from event emitters, communicates to interested event consumers
  + Often used in GUIs

# Requirements

Software requirements: a need, to be fulfilled by the software product

* Brown-field project: a product that replaces/updates existing products
* Green-field project: developing a totally new system

Functional requirements

* Specify what the system should do

Non-functional requirements

* Specify the constraints under which the system is developed and operated
* Examples:
  + Data requirements eg size, persistency
  + Environment requirements eg technical environment
  + Accessibility, capacity, regulations, documentation, stability, etc
* Easier to miss, but critical to success of the software

Qualities of well-defined requirements

* Unambiguous, testable, clear, correct, understandable, necessary, etc

Gathering requirements

* Brainstorming
* User surveys, observation (of target users), interviews, focus group discussions
* Prototyping: a scaled down version or partially reconstructed system
  + Can uncover requirements related to how users interact with the system
* (Existing) Product surveys

Specifying requirements

* Using prose to describe product requirements
* Feature lists: a list of features grouped according to some criteria
* User stories: short, simple descriptions of a feature from a perspective of a user
  + As a user, I can do \_\_\_ so that \_\_\_.
  + Can be high level (epics), and broken down to multiple user stories of normal size
  + Can add conditions of satisfactions to specify requirements for user stories to be counted true
  + Usage:
    - Captures user requirements in a convenient way for scoping, estimation and scheduling
    - Provides more detail than traditional requirement specifications
    - Can capture non-functional requirements as well
* Use cases: a sequence of actions a system performs to yield an observable result
  + Describes an interaction between the user and the system for a specific functionality
    - *Who* doing *what*
  + Use case diagrams: illustrate use cases visually
  + Captures the functional requirements of a system
  + Describes the externally visible behaviour, not the internal details of the system
  + Main Success Scenario (MSS): the most straightforward interaction for a given use case assuming nothing goes wrong
  + Extensions: add-ons to the MSS to describe alternative/exceptional flows of events
  + A use case can include another use case to ‘hide’ low-level steps
  + Preconditions: the specific state we expect the system to be in before the use case starts
  + Guarantees: what the use case promises to give at the end of its operation
* Glossary: to ensure all stakeholders have a common understanding of noteworthy terms

# Implementation

Integrated Development Environments (IDEs)

* Supports all development-related work within the same tool
* Generally has a source code editor with features like colouring, auto-completion, etc, a compiler, a debugger, and other tools like version management support, modelling support, etc
* Debugging: process of discovering defects in the program
  + Bad ways to debug:
    - Inserting print statements: incurs extra effort, unnecessary program modifications which risk introducing errors, may appear in the production version accidentally
    - Manually tracing code: difficult, time-consuming and error-prone
  + Good way to debug: using a debugger

Code Quality

* Maximise readability
  + Understandability is important in writing code – to be read, understood and modified by other developers
  + Avoid long methods
  + Avoid deep nesting (eg if() { if() { if() { … } } })
  + Avoid complicated expressions (eg if(length < max || previous != current && now != null))
  + Avoid magic numbers (eg “return 3.1425926”; “static final double PI = 3.1425926” instead)
  + Make code obvious (eg use explicit type conversion, use braces, use enumerations, even if they can be done away with)
* Structure code logically
* Avoid confusing things
  + Unused parameters, making things similar but different, putting multiple statements together
* KISS (keep it simple, stupid)
  + Don’t try to write clever code; keep it simple: easily debugged
* Avoid premature optimisation
  + Optimise parts with actual code bottlenecks
  + Early optimisation can affect correctness and understandability, may make runtime longer
* SLAP (Single Level of Abstraction Per method)
  + Each code fragment should have the same level of abstraction
* Make the happy path prominent
  + The ‘good’/main path should be clear, as unindented as possible
* Following a coding standard
  + So that the entire code base looks like it was written by one person
  + Naming
    - Using nouns for things (classes/variables), verbs for actions (methods/functions)
    - Using standard words
    - Name should explain the entity
    - Not too long or short
    - Avoid misleading or ambiguous names
  + Avoid unsafe shortcuts when coding
  + Always include a ‘default’ branch in case statements/final ‘else’ in if-else statements
  + Don’t recycle variables or parameters for convenience
  + Avoid empty catch blocks
  + Delete dead (aka redundant) code
  + Minimise scope of variables
    - Minimise use of global variables
    - Define variables in the smallest possible scope
  + Minimise code duplication
  + Comment minimally but sufficiently
  + Don’t repeat the obvious (eg repeating the name of a variable as a comment)
  + Always write to the reader, not as a personal note
  + Explain what and why, not how

Refactoring

* The process of improving a program’s internal structure in small steps without modifying its external behaviour
  + Not rewriting (“in small steps”), and not bug fixing
* Pros:
  + Easier to spot hidden bugs
  + Improves performance sometimes
* Commonly used refactorings:
  + Consolidating duplicate conditional fragments (ie taking out the same code in if and else statements)
  + Extract method (Grouping a code fragment into a separate method)
  + Removing double negatives
  + Replacing magic numbers
  + Decomposing/consolidating conditionals

Reusability

* Enhance robustness of a software system, while reducing manpower and time
* Costs of using many libraries/frameworks/platforms:
  + Reused code may be an overkill
  + May not be mature or stable enough – may possibly change and break the software
  + Non-mature software may die off, no longer maintained
  + Licenses restrict the way software can be developed
  + Reused software may have bugs, missing features or vulnerabilities
  + Malicious code can sneak into your product if these dependencies get compromised
* Application Programming Interface (APIs)
  + Specifies the interface through which other programs can interact with a software component
    - Contract between a component and its clients
  + Examples
    - Class API – collection of public methods that can be used
    - Github API – collection of Web requests formats the Github server accepts, and its responses
  + Writing components’ APIs early let the development team develop components in parallel, more predictable behaviours
  + Designing APIs: use UML sequence diagrams to analysed required interactions, get required API
* Libraries
  + A collection of modular code that is general, can be used by other programs
  + How to make use of a library
    - Ensure it fits your needs
    - Check the license, if it can be reused
    - Download the library and make it accessible to your project
    - Call the library API from the code to make use of it
* Frameworks
  + Reusable implementation of a software, or part of it, providing generic functionality that can be selectively customised to produce a specific application
    - Overall structure and execution flow
  + Can provide a complete implementation of a default behaviour (immediately usable)
  + Facilitates adaptation and customisation of functionalities
  + Example: Eclipse
    - An IDE framework that can be used to create IDEs for diff programming languages, reusing most of the existing features
    - Default behaviour: Java IDE
  + Examples:
    - JavaFX: a framework for creating Java GUIs
    - TkInter: a framework for creating Python GUIs
    - For web-based apps: Drupal (PHP), Django (Python), Ruby on Rails (Ruby), Spring (Java)
    - For testing: Junit (Java), unittest (Python), Jest (JavaScript)
  + Difference from libraries
    - Libraries are used as they are, while frameworks are meant to be customised/extended
    - Your code calls the library code, while framework code calls your code (inversion of control)

Platforms

* Provides a runtime environment for applications
  + Often bundled with various libraries, tools, frameworks and technologies
* Examples:
  + OSes (Windows PC, iOS)
  + JavaEE and .NET
    - Both frameworks and platforms, used to develop enterprise applications
    - Provide customisable infrastructure services such as connection pooling, load balancing, remote code execution, transaction management, authentication, security, messaging, etc
    - JavaEE: used to write enterprise applications; uses Java Virtual Machine (JVM) that can run on different OSes
    - .NET: uses Common Language Runtime (CLR), usually run on Windows

# Integration

What

* Combining parts of a software product to form a whole

Approaches

* Late and one-time VS Early and frequent
  + LAOT: Wait till all components are completed, integrate finished components near the end
    - Not recommended
  + EAF: Integrate early and evolve each part in parallel in small steps, frequently reintegrating
    - Writing a walking skeleton first, then fleshing out features one at a time
* Big-bang VS incremental integration
  + BB: Integrating all components at the same time
    - Not recommended – uncovers too many problems together, makes bug fixing super complex
  + I: Integrating a few components at a time
    - Surfaces integration problems in a more manageable way

Build Automation

* BA tools automate the steps of the build process, usually using build scripts
  + aka: Some steps in the building process happen automatically when the code is ‘built’ in the IDE
* Some popular build tools:
  + (Java) Gradle, Maven, Apache Ant, GNU Make, (JavaScript) Grunt, (Ruby) Rake
  + Some also serve as dependency management tools: download and update the correct version of third-party libraries
* Continuous integration (CI) and continuous deployment (CD)
  + CI: Integration, building and testing happen automatically after each code change
    - An application of build automation
  + CD: Changes are integrated continuously AND deployed to end users
  + CI/CD tools:
    - Travis, Jenkins, Appveyor, CircleCI

# Error Handling

Why

* To recover from unexpected errors well

Exceptions (‘throw’ ‘catch’)

* Dealing with unusual, but not entirely unexpected situations the program may encounter at run time
* Encapsulate details of the situation in an *Exception* object, throw the object, have another piece of code to catch it and deal with it
  + Exception object handed off to runtime system
  + Runtime system tries to find something to handle it in the call stack (exception handler)
  + Exception handler ‘catches’ the exception; if appropriate exception handler not found, program terminates
  + Helps to separate code to deal with usual and unusual situations

Assertions (‘assert’)

* Used to define assumptions about the program state so that the runtime can verify them
  + Indicates a possible bug in the code: a mistake, not an unusual event
  + eg when the execution comes to this point, this variable cannot be null
* If an assertion failure is detected at runtime, the program will typically take drastic actions like terminating execution with an error message
* assert x==0 : “x should be 0”
  + Fails with message shown if x is not 0
* Can be disabled without modifying the code
  + Java assert is disabled by default (but not Junit assert)
* Junit assertions are more powerful, customised for testing, than Java *assert* (functional code)
* Should be used liberally, but not to do useful work as they can be disabled

Logging

* The deliberate recording of certain info during a program execution for future reference
* Useful for troubleshooting
* Typically supported by programming environments: differing logging intensities (amt of info to record), easily disabled and enabled
  + Java: import java.util.logging.\*

# Quality Assurance

QA

* Through testing, static analysis, code reviews, formal verification
* Checking 2 aspects: validation and verification
  + Validation: are we building the correct system? (doing according to requirements)
  + Verification: are we building the system correctly? (correct implementation of requirements)

Code reviews

* Systematically examining code to find where it can be improved
  + Pair programming: two programmers working on the same code at the same time (one writing, one checking)
  + PR reviews
  + Formal inspections
* Adv:
  + Can detect functionality defects as well as other problems like violations of coding standards
  + Can verify non-code artefacts and incomplete code
  + Do not require test drivers or stubs
* Disadv: manual and therefore error prone

Static Analysis

* Analysis of code without actually executing the code
* Can find unused variables, unhandled exceptions, style errors, stats, etc
* Most modern IDEs have static analysis capabilities eg highlight unused variables while you are typing
* Example static analysers in Java:
  + CheckStyle, PMD, FindBugs

Formal Verification

* Uses mathematical techniques to prove the correctness of a program
* Can be used to prove the absence of errors
  + However, only proves the compliance with the specs, not the actual utility
  + Requires highly specialised notations and knowledge, expensive
* Mostly only used in safety-critical software like flight control systems

Testing

* Operating a system/component under specified conditions, observing or recording the results, and then making an evaluation of some aspect of the system/component
* Executing a set of test cases
  + 1. Feed the input to the software under test (SUT)
  + 2. Observe the actual output
  + 3. Compare actual output with the expected output
* Testability: how easy it is to test the SUT, depends on design and implementation
  + The higher the testability, the better the quality of software achievable
* Types
  + Unit testing
    - Testing individual units to ensure each piece works correctly
    - Stubs used to *isolate* the SUT from its dependencies
    - (stub: such simple implementation that it is unlikely to have bugs, hardcoded)
  + Integration testing
    - Testing whether different parts of the software work together as expected
    - Run using the actual dependencies, not stubs
  + System testing
    - Take the whole system and test it against the system specification
    - Based on the specific external behaviour of the system
    - Includes testing against non-functional requirements as well (eg performance testing, security testing, load testing, etc)
  + Alpha and beta testing
    - Alpha testing: by users, under controlled conditions set by the software dev team
    - Beta testing: by a selected subset of target users in their natural work setting
    - “Open beta release”: a release of not-yet-production-quality-but-almost-there software
  + Dogfooding
    - Creators of a product using their own product to test it
  + Developer testing
    - Necessary throughout to find bugs even before the full product is complete, so that search space is smaller, and also to minimise amount of rework; also to minimise chances of product being delayed
    - The earlier a bug is found, the cheaper and easier it is to fix
  + Exploratory vs scripted testing
    - Scripted: Writing a set of test cases based on the expected behaviour, then performing the tests based on that set
    - Exploratory: Devising test cases on the go, based on the results of past test cases
      * aka reactive testing, error guessing, bug hunting
    - Mixture of both is best: exploratory finds bugs in the short term, whereas scripted finds more bugs in the long term
  + Acceptance testing
    - User acceptance testing (UAT): test whether the delivered system meets user reqs
  + Acceptance vs System testing

|  |  |
| --- | --- |
| System testing | Acceptance Testing |
| Done against system specs | Done against requirements specs |
| Done by project team testers | Done by team that reps customers |
| Done in development environment | Done on deployment site, or simulation |
| Tries both negative and positive test cases | Focuses on positive test cases |

* + - System may work in system tests but fail in acceptance testing
    - System testing done first
  + Regression testing
    - Re-testing a software to detect regressions (modifications that result in unintended and undesirable effects)
    - Normally automated as it is carried out after each small change
* Test automation
  + Runs and is determined programmatically
  + Reduces effort, increases precision
  + Test drivers: code that drives the SUT for testing
  + Tools
    - JUnit
    - GUI testing: TestFx, VisualStudio, Selenium to test web app UIs
* Test coverage
  + Measures the extent that testing exercises the code
    - Function/method coverage
    - Statement coverage
    - Decision/branch coverage (both if and else tested)
    - Condition coverage (testing each Boolean sub-expression)
    - Path coverage (the possible paths through a given code)
    - Entry/exit coverage (covering all possible calls to and exits from operations in the SUT)
  + Measured with coverage analysis tools
* Dependency injection
  + Injecting objects to replace current dependencies
  + Often used to inject stubs to isolate SUTs from its dependencies

Test Case Design

* Effective and efficient (E&E)
  + Effective: finds a high percentage of existing bugs
  + Efficient: has a high success rate
* Positive vs negative test cases
  + Positive: designed to produce an expected/valid behaviour
  + Negative: designed to produce behaviour in an invalid/unexpected situation
* Black box vs glass box
  + Black box: designed based on the SUT’s specified external behaviour
  + White/glass box: designed based on the SUT’s implementation (code)
  + Gray box: designed using some info about the implementation
* Equivalence partitions (EP)
  + A group of test inputs that are likely to be processed by the SUT in the same way
  + Advantages: Avoid testing too many inputs from one partition, and ensure all partitions are tested
* Boundary Value Analysis (BVA)
  + To handle boundaries of EPs
  + Choose 3 values around the boundary: from the boundary, just below and just above

|  |  |
| --- | --- |
| Equivalence Partition | Possible boundary values |
| [1-12] | 0, 1, 2, 11, 12, 13 |
| [MIN\_INT, 0] | MIN\_INT, MIN\_INT+1, -1, 0, 1 |
| [any non-null String] | Empty string, a string with max possible length |
| [prime numers]  [“F”]  [“A”], [“D”], [“X”] | No specific boundaries |
| [non-empty Stack]  (assuming a fixed stack size) | Stack with: one element, two elements, no empty spaces, only one empty space |

* Combining test inputs strategies
  + All combinations (very large num of test cases)
  + At least once (includes each test input at least once)
  + All pairs (all combinations between a pair is tested)
    - Variation: test all pairs of inputs, but only for inputs that could influence each other
  + Random (generates test cases using other strategies, picks one randomly)
* Heuristic: ‘Each valid input at least once in a positive test case’
  + A valid input – must appear in at least one positive test case
* Heuristic: ‘No more than one invalid input in a test case’
  + Negative test cases – each should only have one invalid input

# Documentation

Documentation for developers

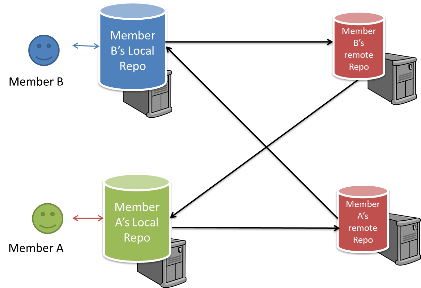
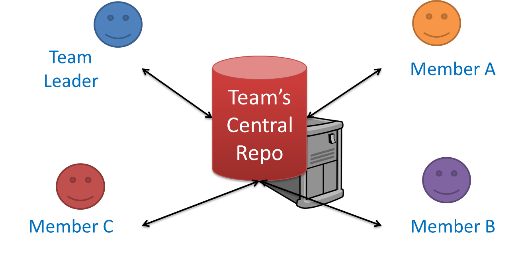
* As a user
  + API documentation (how to use APIs)
  + Tutorial-style instructional: explain functions or methods independently, and how APIs can be useful
* As a maintainer
  + How a system of component is designed, implemented and tested, to maintain and evolve the code (Dev Guide)

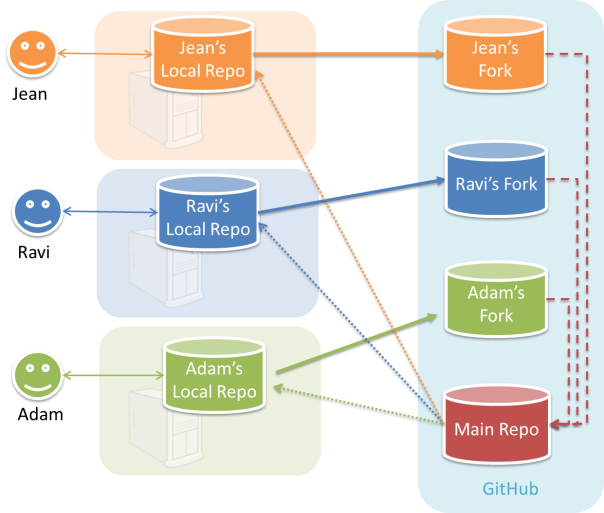
Guidelines

* Top-down, not bottom-up
  + Reader can travel down the path they are interested in until they reach the component of interest, without having to read the whole document
  + Explain from the highest level of abstraction, progressively going down to lower level details
* Aim for comprehensibility (understandability)
  + Using diagrams (but only when necessary, and not for completeness)
  + Using examples
  + Using simple and direct explanations
  + Getting rid of useless statements
* Document minimally but sufficiently
  + Aim for ‘just enough’, provide only just enough guidance to get started
  + Don’t duplicate info in the code already; provide higher level info that is not readily visible in the code or its comments
  + Don’t duplicate chunks of text
* Tools
  + Javadoc – generates API documentation in HTML format from doc comments (@param stuff)
  + Markdown – a lightweight markup language with plain text formatting syntax
  + Asciidoc – similar to Markdown, but more powerful

# Project Management

Revision Control

* The process of managing multiple versions of a piece of information
* Makes it easier to:
  + Collaborate with other people
  + Recover from mistakes
  + Work simultaneously on different versions of one project
* Revision Control Software (RCS) aka Version Control Software (eg Git)
* Repositories
  + The database of the history of a directory being tracked by an RCS software
  + Can have multiple repos in one computer
* History
  + Can specify which files to track and to ignore
  + Staging: specify changes to commit
  + Committing: a snapshot of the current state of the tracked files
    - Each commit is a recorded point in history with an auto-generated hash
    - Can be tagged with a more identifiable name eg v1.0.2
  + Diff: See what changed between two points in history
  + Checkout: restore the state of the working directory at a point in the past
* Remote repos
  + Copies of a repo hosted on remote computers
  + Clone remote repo: creating a copy of a remote repo on your computer
  + Push: new commits in your clone to the remote repo
  + Pull: Receive new commits from remote repo
  + Fork: Remote copy of a remote repo
  + PR: a mechanism for contributing code to a remote repo
* Branching
  + Evolving multiple versions of the software in parallel
  + Merge conflicts: merging two branches that changed the same part of the code, manually resolved
* Ways to do RCS: DRCS vs CRCS
  + Centralised RCS
    - Uses a central remote repo that is shared between the team
    - Older RCS tools; do not support local repos
  + Distributed/Decentralised RCS
    - Allows multiple remote repos, can vary in workflow
  + CRCS and DRCS:
* Forking workflow
  + Official version of the software is kept in a main repo (remote), all members create PRs from their own forks of the main repo to the main repo



Project Planning

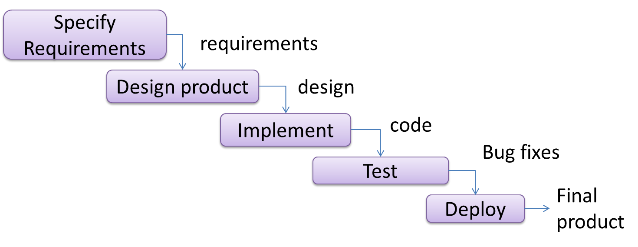
* Work Breakdown Structure (WBS)
  + Depicts info about tasks, and their details in subtasks
  + Task ID | Task | Estimated effort | Prerequisite task
* Milestones
  + The end of a stage that indicates significant progress
  + High-level plan for the whole project
* Buffers
  + Time set aside to absorb unforeseen delays
* Issue/bug trackers
  + Keep track of project tasks: track task assignment and progress

Teamwork

* A close up of a scale

  Description generated with high confidenceTeam structures
* Better to have roles and responsibilities to have someone in charge of different parts of the project

SDLC Process Models

* SDLC: Software Development Life Cycle
  + Stages of SDLC: Requirements 🡪 Analysis 🡪 Design 🡪 Implementation 🡪 Testing
* Sequential model (aka waterfall)
  + Useful when the problem statement is well understood and stable
  + Requires each stage to be completed before continuing
* Iterative model
  + Goes through several iterations of SDLC
  + v1.1, 1.2, 1.3 🡪 v2.1, 2.2, 2.3, etc.
  + Each version goes through all development stages
  + Can take a breadth-first or depth-first approach
    - Evolve major components in parallel, or focus on some parts
* Agile model
  + Requirements prioritised based on the needs of the user and clarified regularly, factored into the development schedule
  + Work based on a rough project plan and high-level design that evolves as project progresses
  + Emphasis on complete transparency and responsibility sharing among team members
  + Examples:
    - XP: stresses customer satisfaction and responding to customer’s changing requirements, emphasizes teamwork, communication, simplicity, feedback, respect, courage
    - Scrum: A process skeleton that contains sets of practices and predefined roles
      * 3 main roles in scrum:
        + Scrum Master (maintains the process, like a project manager)
        + Product Owner (the stakeholders and the business)
        + The Team
      * Encourage co-location of team members and verbal communication
      * Divided into units of development called Sprints
        + Tasks and goals are set in a planned meeting
      * Daily meetings called “daily scrum”
      * Key principle: customers can change their minds about what they want and need

# Principles

Single Responsibility Principle (SRP)

* A class should have one, and only one, reason to change (ie has only one responsibility)

Open-Close Principle (OCP)

* Modules should be written so that they can be extended (open for extension), without requiring them to be modified (closed for modification)
* Often achieved by separating specification (via an interface) from implementation

Liskov Substitution Principle (LSP)

* Derived classes must be substitutable for their base classes
* A subclass should not be more restrictive than the superclass
  + eg parameters passed in overridden methods should not be more restrictive

Interface Segregation Principle (ISP)

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

Dependency Inversion Principle (DIP)

* High level modules should not depend on low level modules but abstractions instead
* Abstractions should not depend on details

Separation of Concerns (SoC)

* To achieve better modularity, separate code into distinct sections such that each section addresses a separate concern; reduces functional overlaps among code sections, limits the ripple effect when changes are made
* Leads to higher cohesion and lower coupling
* Can be applied on class levels or higher levels (eg n-tier architecture)

Law of Demeter (LoD)

* An object should have limited knowledge of another object (don’t talk to strangers)
* An object should only interact with objects closely related to it (Principle of least knowledge)
* Aims to prevent objects navigating internal structures of other objects
* Objects that it should invoke methods on:
  + Itself
  + Objects passed as parameters
  + Objects created/instantiated in the method
  + Objects in direct association