Lec 1

* Dr Rahul Gopinath
* Topics overview
  + Intro, testing and stuff
  + Theory of testing, designing tests, coverage
  + Advanced testing techniques
  + 5 weeks of GoF patterns
  + Software verification theory
  + Specification language tools
* 4 tasks
  + 2.5% ea
* A1
  + testing
  + 10%
* A2
  + DPs
  + Improve an application
  + And write report
    - Focus on rationale
  + 20%
* 60% final
  + 40% hurdle
* Validation
  + Meets the requirements
  + Also called acceptance testing
* Types of testing
  + - This has been taught like 6 times
  + Unit
  + Integration
  + System
  + Acceptance
* Two types of unit TCs
  + Positive test
  + Negative
* Partition testing
  + Groups of inputs with common characteristics
  + From each group, choose a representative example
* Guideline testing
  + From previous experience of errors made
* Edges
  + Input edge cases
    - Including boundary limits
* Test all error messages
  + Buffer overflows
  + Repeated same inputs
  + Force invalids
* Test fixture
  + Context for the test
  + Eg testing data, input, setup process
* Junit
  + Assertions
  + @Test
  + @BeforeAll
* Self after
  + Testing more thorough when black box
    - Contrast with white
      * Can be more intricate

Lec 2

* “testing can show the presence of bugs, but cannot show the absence of bugs”
  + – Dijkstra
* infeasible to test completeness
  + how much testing is enough
* validation testing vs defect testing
* consider who does what testing
  + devs doing their own code
  + integration testing throughout the team
  + outsider does acceptance testing
  + independent testers as a specific role
* agile tests as they go
* functional testing
  + unit, integration, system, regression testing
  + acceptance
* non functional
  + performance
  + load
  + security
  + stress
  + reliability
  + usability
* static testing
* v model
  + a software dev life cycle testing model
  + test phase parallel to each dev phase
  + a verification stage has a validation stage
* equivalence partitioning
  + group inputs with common characteristics
  + then only use one input from each characteristic partition
  + boundary cases
* test case order
  + should have zero dependence on order
* design smells
* GRASP
  + General responsibility assignment software pattern
  + Responsibility driven design
* 5 basic principles
  + 3 5 8 11 3
  + Creator
  + Expert/ information expert
  + High cohesion
  + Low coupling
  + Controller
* Creator
  + Common problem in OOP: who creates class A
  + Assign an object to create an instance of a class if
    - B aggregates a
    - B contains a
    - B records a
    - B has the init data for a
    - low coupling, increased clarity, encapsulation & reusability
* expert
  + place the responsibility with the data
* dependency
  + dependency exists when a change in the supplier will cause change in the client
* coupling
  + the degree of connectedness between different elements
* low coupling principle
  + assign responsibilities to keep coupling low
* cohesion
  + how related are the responsibilities of the class
* clarify:
  + coupling is for inter-object relationships
  + cohesion is for intra-object relationships
* GRASP 5 + 4 extra
  + Tim the king tutor said know these ones too
  + Polymorphism
    - Use an interface or abstract class to group behaviours in expected places
  + Pure fabrication
    - Something that does not have direct use in the business context, but helps
  + Indirection
    - Talk to something else
  + Protected variation
    - Base implementations on interfaces
* SOLID
  + Single open liskov interface dependency
* API
  + Apis should be self documenting
  + Document everything anyways
* Interfaces are more flexible than abstract classes
  + Prefer them
  + Eg factory methods
* Massive method params
  + Use helper methods with getters
* Junit
* Test timeouts
  + Triggers a timeout and fails the test if it exceeds a time limit
  + @Test(timeout=1000)
    - Time in ms
* Rules
  + Add behaviour to a test
  + Temporary folders for deletion after a test
  + Errorcollector
  + Expectedexception
  + Externalresources
    - Use as another attribute in the class
    - @Rule  
      public Timeout globaltimeout = Timeout.seconds(10)
    - Have to import org.junit.rules.<rule>

Lec 3

* Testing types
* Advanced testing
  + Black/white box
  + Doubles
  + Contract
* Mockito
* Subsytems
  + How do you test every combination of subsystems
    - Exponential numbers
* Incremental vs big bang subsystem testing
  + Add components one by one
  + Or hit them all at once
* Incremental
  + Might not find that specific subsystems clash
  + Does let it be chunked, more likely to find edge cases
* Big bang
  + More general use
* Testing vs src
  + 3:1 l
    - 3 lines of test code per line of code
* Procedural interfaces
  + Message passing interface
    - Requests from component to another component
  + Shared memory interface
    - Typically embedded systems
    - More abstract procedural interface
* Interface misuse
  + Misunderstand the specs of the interface
    - Eg closing bufferedreader before filereader
  + Timing errors
    - With parallel processing and similar
* Incremental testing
  + Driver
    - Main object
  + Stub
    - Module that is being acted upon by the driver
    - Or otherwise isn’t integrated yet
  + More effort to implement
* Top down
* Bottom up
* Functional incremental
* Big bang is basically full stack testing
  + Easy to do
    - Best for small systems
* Mockito
  + Lets you mock objects to check behaviours
  + Test doubles
* Dummy object
  + Passed around but never used
  + Used to fill parameter lists
* Mockito
  + Tag the main class with @ExtendWith(MockitoExtension.class)
  + Add a param @Mock  
    ShoppingBasket sb;
  + Then you can use mock() to create a simulated object
    - Eg list fakes = mock();
    - This list can now be used for anything that needs a list param
* Stub object
  + Used a mock() to sim a list
  + When(mockedlist.get(int) ).thenReturn(output);
  + This defines the default behaviour of the mocked object
    - A stub
    - Then you can use this in the assert
* Spy object
  + Specific stub that records info
  + Create an object properly, eg linklist
    - Then make another list = spy(linklist);
    - Now use the spy list for your testing
    - Verify(spyobj).operation(param);
    - And then you can assert with the spyobj
  + Dummy makes a fake object, spy wraps a real object
* Contract test
  + Run tests with real components
* Other mocking frameworks exist
  + Mockito is a java one
* Task 1
  + 25 marks
  + 5 for 100% code coverage
    - Tests will be run against a bugged version of the src, 5 marks per testcase failed
    - Meaning my tests must be thorough enough to find all edges that are ok in the provided version

Lec 4

* Advanced testing techniques
* oracle
  + A mechanism that tests if a software test execution passes or fails
* When using mocking, we use a human oracle
  + The problem is its not scalable
* Human oracle
  + Individual written tests
* Formal specification
  + Something that can be defined and written down formally with absolute specifity
    - Then use this as a basis for testing comparisons
* Specification languages
  + These exists
    - One used to be taught called alloy
      * But they aren’t maintained
* Rather use property based testing
  + Use the formal spec as the oracle
  + Jqwik in java
* You will give a formula that is the connection between the input and output, and provide that to a generator
  + It will auto generate tests for that method
* Generator
  + Generates
  + Inputs that conform to the input specification
  + Checks the output constraints
  + Typically provide multiple input params
  + Effective generators should cover boundaries and interesting inputs
    - Should also give enough tests to give statistical confidence
* Note
  + Property based testing doesn’t actually prove the program is correct
    - We just check that the properties specified hold for a number of inputs
  + Specification languages do actually confirm
* Need to import net.jqwik.api\*;
  + @Property  
    void addTest(@ForAll @Positive int n) {  
    ShoppingBasket sb = new ShoppingBasket();  
    sb.addItem(“apple”, n);  
    sb.removeItem(“apple”, n);  
    assertEmpty(sb);
* Other param tags
  + @IntRange(min=1, max=10) int n
* To do more specific stuff use a provider
  + @Provide  
    Arbitrary<Integer> positive10() {  
    return Arbitraries.integers().between(1,10).filter(i -> i %2==0);  
    }
    - Then for the test method can use param @ForAll(“positive10”) int n
* Arbitrary can also be used for strings
  + @Provide  
    Arbitrary<String>name() {  
    return Arbitraries.of(<List>);}
* Can also do post filtering of generated parameters
  + Assume.that(condition)
    - This is really not optimal
      * Will fail the tests where the condition fails
      * Much rather do the filtering on the provider
* Can also give param to @Property
  + @Property(tries=1000)
* Param @ForAll(“name of arbitrary”)
  + Include quotation marks
* All of this isn’t actually that scalable
* Derived oracle
  + An oracle that can be derived from an artifact of the program
* Can test against other program
  + Eg made your own json parser, test against existing one
  + Some critical apps will have multiple teams dev versions and test against each other
* Regression derived test
  + For each bug that is fixed, you add a test case for that specific bug
    - Can make sure that you don’t reintroduce bugs
* Implicit oracle
  + Based on anomalies
  + Essentially just checking for unhandled exceptions

Lec 5

* System testing
  + Non functional properties
    - Security
  + Functional requirements
    - Does it do what you need
* Testing teams
  + Some places do, some don’t
* Tutorial
* Testing java function times
  + Long start = System.nanoTime();  
    method();  
    long end System.nanoTime();  
    long duration = end-start;
* Function testing
  + Use user stories
  + The world is the world, include missing data stuff
* Non functional testing
  + Security
    - Check for crash inputs
  + Usability
  + Performance
  + Reliability
* Load testing
  + Load test at the normal/expected system stress
  + Stress testing is when you try overload it
  + Spike testing
  + Endurance testing
  + All of these will help identify bottlenecks in software and hardware
* Java microbench harness
  + In build.gradle
    - Add a plugin { id “me.champeau.jmh” version “0.7.0”
      * Or other number?
    - Add a class jmh { iterations = 1
  + Jmh will follow the same structure in src as main and test
    - Src -> main, test, jmh

Lec 6

* GUEST lecture from citadel
  + Thanks dickheads
* OO and recap
* Encapsulation
  + Wrap data/state and methods into a class
  + Protect via setter and getter
* Inheritance
  + Sub class inherits
  + Reuse structure and behaviour
* Variable binding and polymorphism
  + Object has a single type when created
    - Cannot be changed
  + Reference vars must point to null or an object
  + Type of object and reference variable may differ
    - Eg Shape r = new Rectangle();
  + Understand runtime type != compiletime type
  + Polymorphism
* Virtual dispatch
  + Methods in java permit late binding
  + Aka override methods
* Abstract classes
  + Only contain method signatures, not body
  + Cannot generate instances of them
* Interfaces
  + Implements methods which can be accessed from classes that implement the interface
  + Interfaces can inherit from interfaces
  + Variables permitted ONLY if static and final
* OO
* GRASP
* Also 5 basic principles
  + Creator
  + Info expert
  + High cohesion
  + Low coupling
  + Controller
  + See lec 2

Lec 7

* Why use SQL
  + Databases ensure data security and access
  + SQL is used for DBs
* Basic operations
  + CRUD
  + Create
  + read
  + Update
  + Delete
* Relational
  + DB2, MySQL
* Hierarchical
  + IMS
* Graph based
  + Neo4j
* OO
  + ObjectDB
* Flat file
  + Berkely DB
* Others too
* Common relational DBs
  + SQLite is the simplest
  + File based, doesn’t need a DB, works as a layer to files
* ACID
  + Atomicity
    - All or nothing, meaning any requests are either 100% completed or aborted entirely
  + Consistency
    - All rows and columns are consistent, the DB is always consistent across operations
  + Isolation
    - Parallel transactions cannot interact
  + Durability
    - After committing, the commit remains
    - Unlike in some files where there are possible reverts or shadows in memory
* Learn to use SQLite
* We use JDBC to interface java and DBs
  + run from a main class using @SprintBootApplication  
    SprintApplication.run(class.class, args);
* using packages from java.sql.
* PreparedStatement
  + Include ? in the string and then the PreparedStatement.setString() will swap the first ? in the string for the value given
  + Can do multiple in one prepared statement
* Also need to add an implementation in dependencies in gradle file for jdbc
* Representation state transfer aka REST
  + Makes your web apps into ‘law abiding’ web objects
  + When we use sites, we expect stuff to work a certain way
  + And we should be able to interact with everything going on, modular too
  + REST is a way to make sure our web apps have default behaviours and use appropriate infrastructure
* Use HTTP protocols
  + Eg get post put delete
* REST
  + For build APIs that are
  + Backwards compatible
  + Evolvable
  + Scalable
  + Securable
* Example
  + Request from a browser to a web server
  + That server calls to a database according to the request received
* REST
  + Resource identification through URI
    - Uniform resource identifier
  + uniform interface for all resources
    - using HTTP verbs
    - GET PUT POST DELETE
  + Self descriptive messages through meta data
  + Hyperlinks to define the application state
    - Address the resources explicitly
* Resource
  + A resource is something interesting/worth checking out in the system
  + Requests are sent to URI
  + Never use the source itself, always a representation
    - But also never use a session state, make sure updates are applied live
* URI
  + Must be descriptive
  + Eg <http://spreadsheet/cells/a2,a8>
  + Opposite of opaque URIs like the typical ones on browsing
    - Can’t really get any useful information
* Links
  + Connectedness is good
  + Each resource should contain links to other resources
    - As necessary
* HTTP verbs
  + Get
    - Retrieve a representation of a resource
  + Head
    - Get metadata for existing resource
  + Put
    - Creates to a new URI with given name, or modifies existing one
    - Meaning if we run the same put command multiple times, we still only get 1 result
  + Post
    - Creates resource, will generate a URI automatically
    - Different to put in that post adds while put overwrites
    - Post functions more typically like .add(), where it adds regardless of current items
  + Delete
    - Deletes
    - Stops accessibility
  + Options
    - See which verbs are ok for the resource
* HTTP status codes
  + Provides metadata about state of resources
  + 1xxx is metadata
  + 2xx everything is fine
  + 3xx redirection
  + 4xx client did something wrong
  + 5xx server did something wrong
  + We will know some of them
* REST weaknesses
  + Possible to have design mismatches
  + Specificity in name resources and URIs can be annoying and tough to maintain
  + Different access points need to all function on the same DB
* Tiers of enterprise application
  + Presentation tier
    - UI
  + Logic tier
    - Processes commands, calculations, processes data
  + Data tier
    - Data base, stores and lets logic tier get the datas
* Transactions should be executed as wholes
* Unit of work relational pattern
  + Maintain a list of objects that are affected by a transaction
  + Coordinates concurrency problems
  + Can use this as an intermediary step, whereby one big patch will be applied to the DB as a whole
    - Instead of hundreds of individual calls
* DB calls
  + Write calls are super expensive
    - All other access must be limited during a write call
  + Using batched access makes the writes less frequent
* Lazy load
  + An object doesn’t contain all of the data you need
  + But you can lazily load what is in an object as you need it
  + Eg create a customer object but don’t start populating their preferences until they ask
* Lazy load varieties
  + Lazy initialisation
    - Default all to null, and then locally update as it is required
  + Virtual proxy
    - Use a light copy, minus most of the detailed stuff
  + Value holder
    - Simple version of lazy
  + Ghost
    - Partial load of object

Lec 8

* Enterprise design patterns
  + Flyweight
  + Bridge
  + Chain of responsibility
* Flyweight
  + Used when making a massive number of similar objects
  + Share bits that are similar
    - Eg 100k
  + Intrinsic and extrinsic
    - Intrinsic is where the space saving occurs
  + When creating a new object, have a look in a hashmap for objects stored by their common factor
    - Eg looking for rectangles of different colours, HashMap<Color, Rect>
    - And when we find a rectangle that already exists, we will return the same one every time
* Flyweight rectangles
  + MyRect class holds x, y, color
  + Myrectfactory holds a hashmap for <Color, MyRect>
  + And when creating a new object,
* Bridge
* Chain of responsibility
* LLMs
  + Large language models
  + Prompt for testing
  + Open assistant.io is free and open
* “Try work with LLMs on assignments, but just say that you have used them” – GOAT Rahul
* Unit of work
  + Recap: keep a list of objects in a DB that need updating and then make one larger access to the DB
  + Reduce lockup times
* Pro cons
  + Reduces calls
  + Makes concurrency super hard
  + Fine grained consistency is lost
* Repository DP
  + Typically used with unit of work
  + Abstracts data access and persistence logic
  + Centralise handling
  + Places layers into the UoW process
  + Follows CRUD philosophy
* Lazy load
  + Different ways to not do heavy initialisation up front
  + Virtual proxy
    - Stores commands but doesn’t perform them on the object, just waits until the actual content is 100% required
  + Value holder
    - Holds a reference to object
    - Other methods delay loads, this performs ops on a local representation and loads them afterwards too
  + Value object
    - A class to implement a custom type
      * Instead of saying float distance;
      * Make a class for distance and have options for meters/feet
      * Good for safety and consistency
      * But higher overhead costs
* Off topic
  + Llama is a facebook ai, which isn’t dafka open source but fb will send you the weights and some code for it
  + So you can run it locally
    - Holy shit
    - Needs a GPU though
* Martin fowler self-testing gpt code

Lec 9

* DP
* Visitor
  + YOUTUBE THIS
* Enterprise design patterns
* State
* Imperative programming involves altering state
  + Values linked to fields
* Infrastructure
  + Provides support for computation
* Resource
  + Data
* States
  + Resource state
  + Session state
    - Over a period of time
  + Local computation state
    - Immediate variable value
* States are important for concurrency
* Difference between concurrency and parallelism
  + Concurrency supports simultaneous execution of two tasks in some fashion
    - interleaving
  + Parallelism is when they actually run simultaneously
    - Generally based on hardware resources
* Concurrency also allows running multiple processes simultaneously, as long as they are independent at the time
* Definition: task is a set or sequence of computations
  + Programs are broken down into tasks
* Data parallelism
  + Performing the same task to different data simultaneously
    - Eg searching half of a dataset in 2 processes
* Independent tasks or datasets
  + If a task is dependent on the result of another, they cannot be run in parallel
  + No concurrency is possible
* Concurrency problems
  + Interleaved computations on shared resources
  + Desynchronisation
    - If one dependent task attempts to run prior to its dependency being fulfilled
    - this can be solved by leaving notes/flags
    - different thread places a note that it is processing something already, and the dependency shouldn’t try rerun the task
* critical section
  + a part of the program that should only be run by a single thread
* transactions
  + complete action that is indivisible
    - transfer cash between accounts
  + maintain integrity despite failures
  + they are good for high volume
  + avoid downtime and data loss
* transactions ACID test
  + atomic
    - all or nothing
  + consistent
  + isolated
    - no problems with concurrency
  + durable
    - changes made will not be lost by other failures
  + AtCoIsDu
* java concurrency
  + java can run multiple threads
  + you should be familiar with threads
    - thanks
* threads
  + subclass thread
  + public class testThread extends Thread {code here }
  + Thread thread = new testThread();
  + thread.start();
  + thread.currentThread();
  + thread.pause(100)//100 ms
* concurrency patterns
  + active object
  + thread pool
  + read write lock
  + optimistic lock
  + pessimistic lock
* active object
  + separate method invocation from method execution
  + allows system to remain responsive during time consuming ops
  + components
    - proxy
      * interface for clients to interact with
    - scheduler
      * manages and prioritises client requests and forwards to server
    - servant
      * performs the work
    - activation list
      * list that is sorted by the scheduler and stores the servant work
* callbacks
  + a callback is a function that is executed when a task is completed
  + eg just takes a string of the function and prints it
  + instead of just using System.out.println
    - use a class, that can also do other stuff
* Thread pool pattern
  + DP for achieving concurrent execution
  + Also known as replicated works
  + This is for when tasks are much slower than in active object
    - So executing them concurrently is much better for performance
  + Threads are maintained by thread pool
  + Synchronised queue
* Thread pool advantages
  + Improved performance by reusing worker threads
  + Resource management easier controlled
  + Throttling and balancing
* Thread pool disadvantages
  + Limited parallelism by number of worker threads
  + Complexity
  + Resource contention
    - Too many threads and not enough specs means no point
    - Can lead to deadlock if threads are stopped and reliant on other threads to complete work
* Optimistic lock design pattern
* Pessimistic lock enterprise pattern
* Closing stuff
  + The main part of the class is active object and thread pool
    - Locking is just a know exists
  + Concurrency keeps changing
    - Use the recommended method for the language and spec
* Exam
  + 10 multiple choice questions
    - 1 mark each
    - SOLID GRASP sorta stuff
  + 5 mark short answer quests
    - Eg explain REST
    - Where is SOLID/GRASP used in an example
  + 10 mark program question
    - Find bugs
    - Different test cases in a triangle program
    - How can we apply a testing concept to a program
    - What is REST and where is it used
    - Implement a DP for a problem
  + Thread pools and active objects
    - Know the principles of locking

Lec 10

* Design patterns
  + Abstract factory
  + State
  + Command
  + Mediator
    - Less assessable focus on these, know they exist
* Abstract factory
  + Object creational
  + Provide an interface for creating families or related or dependant objects
  + Also known as kit
* State
* Command
* Patterns for interactive applications
  + MVC is the main pattern we should understand
* Model view controller
  + Splits user interface interaction into three distinct roles
* Model
  + Object that represents information about the domain
    - Eg customer object
    - Should represent how you would think about the data involved
* View
  + Representation of the data in UI
* Controller
  + Handles user interactions, updates the model
* MVC
  + View depends on the model but not vice versa
  + Can have different views
  + View is observer pattern of model
  + Between the view and the controller, separation and dependency is less important
  + One view and two controllers
    - Editable and non editable controller
    - Use strategy DP
    - In practice, generally one controller per view
* Page controller
  + Object that handles request for a specific page
  + We can use springboot for this
  + Handles view and model
  + Ideally one module per page
  + Controllers link to each action
* MVP
  + Model view presenter
  + Different way to arrange interactive application compared to MVC
  + MVC is pretty abstract
    - Plenty of different ways, dependant on framework
  + Presenter is the middleman between view and model
    - Handles user inputs and updates the view and model
  + Not specifically important to learn, know this exists though
  + Common for desktop and mobile apps
* MVA
  + Model view adapter
  + Adapter is a mediator, model and view communicate via adapter
* MVVM
  + Model view ViewModel
  + Viewmodel is abstraction of view
    - Exposes properties and commands for data binding
* Flux
  + Emphasises unidirectional data flow
  + Contains dispatcher
  + Stores
  + Views
  + All react components
* Redux
  + JS/react
  + State management library
  + Maintain app state in a single immutable data store
  + Actions make a new one
* All of these models have slight differences
  + Mainly in how coupled the actors are
  + Understand that MVC exists, and all these versions are different implementations
  + There is a listener, a data class, and something that presents it
* “like wine tasting, unless you’re really familiar, its all the same enough thing”
* Supervising controller
  + Only has view and controller
  + Very limited
* Passive view
  + Only view and controller
  + Limited model and view
* Presentation abstraction control
  + Separates concerns into hierarchy of cooperating components
    - Presentation
    - Abstraction
    - Control
  + PAC introduces more abstractions
  + Allows construct user interfaces at any level
* DCI data communication/context interaction
  + Decomposes app into hierarchy of abstractions

Lec 11

* Test coverage
  + If there aren’t meaningful assertions and tests, coverage is useless
  + You can easily write code to hit all lines but not actually test for anything meaningful
* Fault seeding
  + Take the functional codebase and test suite, and deliberately fuck up N things
  + Then see if the test suite catches them
  + Effectiveness = bugs caught/bugs made
* Problems with fault seeding
  + Fault interaction
  + Individual faults may disproportionately be impacting the tests
  + Manual generation is a pain and takes time and effort
  + Manual faults might not mirror real world faults
* Mutation testing
  + Mutate the formulae slightly
  + Example
    - D = b^2 – 4\*a\*c
    - So make versions with
      * D = b^3 – 4ac
      * D = b^2 + 4ac
      * D = b^2 + 4 + ac
  + Much simpler to introduce
  + Rerun the test suite for each mutant
  + Any mutant that does not pass all tests is killed
  + If a mutant survives, it proves that the test suite is not sensitive enough
* Finite neighbourhood assumption
  + We assume that small mutations will be sufficient to make changes to actually fail tests
  + We still need to try hit different parts of the program
* Coupling effect
  + Mutants are functionally the same as their siblings
  + Makes the testing more computationally expensive
* Gradle pitest plugin
  + In build.gradle plugins {id ‘info.solidsoft.pitest’ version ‘1.9.0’ }
  + Run with gradle pitest
* Writing good test cases – one liners
  + Keep it simple
  + Readable
  + Avoid magic strings
  + Write tests during dev
  + Add tests corresponding to fixes
  + Tests should not be removed
  + Idempotent tests
    - Repeatable
  + avoid abstractions
  + test interfaces
* adequacy
  + a test suite is adequate if it achieves x% predefined amount of test cases
* subsumption
  + a criterion subsumes another if a test suite adequate for one is adequate for the other

Lec 13/exam

* On the exam
* Multiple choice x10, 1 mark ea
* Acronyms x3, 5 marks ea
  + Eg CRUD
* Program based Qs x2, 20 marks ea
  + Here is a short codebase
  + Can you get full coverage?
    - /are all code lines accessible
  + Find the bugs
  + Write unit tests
  + Negative tests x2?
* Detail Qs x3, 5 mark
  + Name and explain SOLID
* Essays x3 10mark
  + Explain UoW pattern
  + Specify its methods

Content for me to revise

* W1
  + Positive and negative test cases
    - Positive test checks correct function
    - Negative test checks for error catching
  + Techniques to choose testcases
* W2
  + Functional and non functional testing
  + SOLID
    - Substitution principle
    - Open closed
    - Liskov
    - Dependency inversion
  + GRASP
    - General responsibility assignment software pattern
  + YAGNI
    - You aint gonna need it
    - don’t add functionality until it is required
  + DRY
    - Don’t repeat yourself
    - Anything that is written more than once should be abstracted
* W3
  + Regression testing
    - When you find a fix a bug, create a test case for that bug specifically
  + Gray box
    - Where you know stuff like function/class names but not details
      * White is where you know line by line, you can target like that
      * Black is just external
  + Kinds of coverage
    - Line coverage
      * overall %
    - Statement coverage
      * Number of statements covered
      * Impacted by more complex/multi operation lines
    - Function coverage
      * Only checks % of funcs
    - Branch coverage
      * Each if else is counted as 2 branches
      * How many branches are ‘opened’
* W4
  + Test oracles
    - Something that determines whether a test passes or fails
    - Human
      * Can see input output
    - Derived oracles
      * Derived from the things the program outputs
      * Pseudo
        + Where it is hard to implement an explicit oracle, use an approximation of the software to check how it might look

Sorta like a basic heuristic, eg using 1 rule ML algo as a comparison/baseline

* + - * Inverse
        + If a function has an inverse, apply them both and the input should == output
        + assertTrue(data, decompress(compress(data)) )
    - Implicit
      * If it crashed, we can gauge it didn’t work
    - Approximate
      * If CPU use rockets and stays up, we can gauge it doesn’t work
      * Or other invariants
  + Fuzzing
    - Generate a bunch or random inputs that expose flaws
    - Generation based
      * Use a generator to create them
      * Follow a template, all should follow expected behaviour
    - mutation based
      * mutate provided seeds slightly
    - hybrid
      * Combination
      * Mutation plus intelligent algos
* W7
  + CRUD
    - Create read update delete
    - Four basic operations for any persistent storage application
      * “4 major functions inherent to relational DBs”
  + ACID
    - Set of properties for ideal DBMS
    - Atomicity
      * All or nothing, meaning any requests are either 100% completed or aborted entirely
    - Consistency
      * the DB state is never invalid, no transaction can invalidate the state in any way
    - Isolation
      * Parallel transactions cannot interact
    - Durability
      * After committing, the commit remains
  + REST/representational state transfer
    - Strengths and weaknesses
  + ReST principles
  + REST weakness
    - Rest is synchronous
      * It stops and waits for data before doing anything
      * So if an asynchronous client sends data
        + And in that instant the synchronous REST API makes a call to the client
        + The client will pass the previous data, but the message that was just sent will not be received until another call is made
        + The request will only fetch things logged prior to the request
  + Web technologies
    - HTTP/S
      * A server gets a security certificate from a certificate authority
      * Server and client generate a session key
      * Data is sent and encrypted through the session key
      * Data confidentiality, integrity through encryption checking
    - HTTP caching
      * Cache previously requested resources at local client, intermediary server, or original server
      * Improves speeds
      * Checks for latest versions
    - HTTP authorisation
      * Basic with username/password
      * Digest
        + Sends hashed/encrypted
      * Token
        + Access token granted from basic or digest methods
* W10
  + MVC
    - Model view controller
    - Web app architecture pattern
    - Model
      * Data
    - View
      * GUI and bits, gets from model
    - Controller
      * Interface model and view
* PATTERNS
  + Enterprise and GOF
  + DETAILS/ESSAY Q
    - Strategy
    - Flyweight
    - State
      * Basically like strategy, but dealt with internally, eg money runs out and state is switched to broke
      * Also changes the whole thing, not just a single algo that switches
    - Mediator
      * Central class in between objects that handles all communication
      * Mediator stores instances of all colleagues, but colleagues only store mediator
      * Will have abstract/interface for mediator, concrete, colleague classes
    - Template
      * breaks an algorithm into multiple steps in an interface/abstract, and therefore different concrete implementations can vary the parts as needed
      * eg generatereport() calls fetchdata(), analyse(), put to file(), send()
      * each of these is its own method too
        + different for commercial vs solo clients, monthly vs weekly
    - Abstract factory
      * Extension of factory DP
      * Abstract class for factories, each of which is a factory for a family of kinda related objects
    - Visitor
      * When there are a few similar operations for different classes
      * Analogy
        + Zoo with abstract zookeepers, concrete zookeepers that actually visit the animals
        + You can train more zookeepers without having to touch the animals
        + In the animal classes, they will go zookeeper.visit\_lion or specific
      * Interface for visitor, has all the needed methods
        + But each concrete visitor might only actually implement one of them
        + The client object will have an accept(ConcreteVisitor) method
        + Then in the client method it will do visitor.method() for all of the different methods in the concrete implementation
        + Most will be void, but one will actually be implemented

This is kinda a crap pattern

* + - Chain of responsibility
      * Handler, concrete handler, client
      * Client only has to interface with one concrete handler
      * Good for hierarchies/processes
    - Observer
      * One to many
      * Subject and observer interface/abstract, and concretes
      * Concrete subject holds a list of concrete observers, and will notify() all in that list
      * So the observers don’t know about the other observers
    - Command
      * An instruction is encapsulated in an object so that a client can send instructions to different objects without knowing details of the receiver
      * Command interface declares execution method
      * Concrete command for different commands. Contains reference to receiver obj
      * Receiver performs the commands
      * Invoker receives the command and calls command.execute(). Doesn’t know details
        + Abstracts instructions from client, reduces coupling
        + Enables queuing
    - Bridge
      * Abstraction interface defines client methods
        + Contains reference to implementor
      * Refined abstraction extends abstraction
        + Optional subclass of abstraction, extends or customises methods
      * Implementor interface
        + Declares methods for implementor
      * Concrete implementations implement implementor
      * Client
    - Bridge demo
      * Concept: Remote that connects to devices
      * Abstract remote has button presses for 9 buttons
        + Can have concrete methods for stuff that is consistent
      * Concrete remotes implement the abstract methods
        + Stores the device, and in button presses will do device.volumeUp()
      * Abstract device
      * Devices
      * Client
        + Will create remotes and device, pass device to remote
        + Call button presses