1. Why software testing?

* Save money
* Security
* Increase quality of product
* Satisfaction of customer
* Enhance the development process
* Easy while adding a new feature
* Determine the performance of software

1. What’s the relationship between “testing” and “quality assurance”?

Quality assurance and testing are two integral parts of software development process.

**Quality assurance** ensures that a product meets its **intended requirements**

**Testing** verifies that a product’s **functionality** works as expected

1. What are some types of software “quality”?

Portability, Usability, Reusability, Correctness, Maintainability, Reliability, Efficiency

1. What are the similarities and differences in the way that various software process models incorporate testing and quality assurance?

* Waterfall models: it’s sequential and linear, test is in the **very end** step, so it’s insufficient testing and time-consuming to fix
* V-model: it’s a rigid and uncompromising testing model. it’s a modified version of waterfall model, place emphasis verification and validation, test design is incorporated **earlier** in the process and testing **various phase** as it is completed before moving to next one
* Spiral model: The spiral model combines elements of both the waterfall and iterative models, emphasizing risking analysis throughout. Additional functionality can be added later if any circumstances change in the middle of the process

1. Validation and Verification

* Validation: What we design meeting customer’ requirements or not?
* Verification: What we build meeting the what we design or not?

1. Mistake, Fault, Error, Failure

* Mistake: A human action results in an incorrect result
* Fault(**bug**): An incorrect code that may lead to an error
* Error: **Runtime** effect of executing a bug, may result in a failure
* Failure: The **manifestation** of an error

1. Test case, Test suite, Test oracle, Test effectiveness, Test plan

* Test case: **A defined set of input values** that cause a program to call some defined action, with an **expected output**
* Test suite: A collection of **test cases**
* Test oracle: A mechanism to determine whether the actual results match the expected behaviour
* Test effectiveness: The **extent** to which testing reveals faults or other objectives
* Test plan: A **document** describing scope, approach, resources and schedule of **intended** testing **activities**

1. Why not exhaustively test, or fully prove for correctness?

* Testing can only prove the **presence** of bugs not their absence
* Testing allows only **sampling** of enormous large program input space
* It’s difficult for tester to construct **effective samples**

1. What differences between black box and white box

* Black box testing: tester doesn’t know the **internal** structure/design/implementation of the project, just test the **functionality** and **external** structure and design
* White box testing: tester test the implementation and impact of **inner** code of a program.

1. What is the basic power of testing to find bugs, Why do we then test at all?

* Software can **never** be assumed to be correct
* Any software must be verified, Important to control the quality of the product

1. Principles of quality assurance?

* Partitioning: divide and conquer
* Visibility: making information **accessible**
* Feedback: tuning the development process
* Restriction: making the problem **easier**
* Redundancy: making intentions explicit, allow for **consistency** checks
* Sensitivity: better to fail every time than somethings

1. What is the exploratory testing? Strengths? Limitations? Who want to do? How? When?

* Exploratory testing is a type of test we **inherently** already do **without script**
* Strength: use **human intelligence** to guide testing; allows testers to observe and evaluate the unspecified behaviors; encourages testers to take notes to guide future testing
* Limitations: unscripted, unrepeatable, unautomated
* Who: developers, testers, designers or other project stakeholders
* When: requirements are **unclear**, time is **limited** or traditional test cases may **not** cover all potential issues
* How:
* Understand the software context and objective
* Plan rough test sessions
* Execute tests **without** predefined scripts
* Documenting finds and unexpected behaviors
* Adapt and refine approach based on insights
* Reviewing an share results with stakeholders

1. What is an exploratory testing “test charter”

The **mission** of testing, and perhaps also some **tactics** to be employed. Are often **vague** on purpose

1. What is acceptance testing? What form can it take?

Assessment of whether agree-upon **requirements** are met. Often employ checking the **business contract** between **client** and **contractor**

* Smoke testing
* User acceptance testing
* a type of testing performed by the end user or the client to verify the software system before moving the software application to the production environment
* A form of testing that bridge the gap between the verification and validation
* Intended to be performed by **people**
* Check to see if the requirements match the behavior, and match needs
* End-user testing: Aloha and beta testing

1. What is smoke testing(**build verification testing**)

* Ensure that **most critical functions** of a software application are working correctly
* Not unlike exploratory testing
* Exercise the **major** functionality, ensure that everything still looks OK
* Often performed **before** even starting the **main** testing process

1. What is behavior driven development and how does it relate to acceptance testing?

* Write test cases that follow the **action model** (given, when, then)
* More **verification-focused** form of **acceptance testing**
* Focus on the **behavior** of the program rather than the tester
* Imposes the **structure** to test specification
* More **high-level** than TDD

1. How does BDD related TDD?

* BDD’s motivation is programmer need help with TDD. Knowing where to start, what to test, what not to test, how much to eat...
* BDD often considered more **high-level** than TDD

1. Random testing versus systematic?

* Random testing
  + Pick possible inputs **uniformly**, avoid designers’ bias
  + Treats all inputs as **equally** valuable
* Systematic
  + Try to select inputs that are **especially** valuable
  + Usually choosing representatives of class that are **apt** to fail often or not at all

1. Partition

* Breaking down something into smaller, do-able steps
* Separates the input space into classes whose **union** is the entire space

1. Why do we do functional testing?

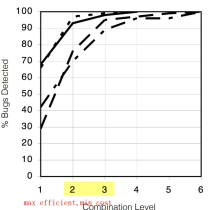
* Exploiting the **specification**
* Uses the specification to partition the input space
* Test each category and boundaries between categories

1. What is combinatorial testing?

Partitions for **multiple** categories presents the opportunities for each partition to be tested

1. What is the purpose pairwise testing, n-wise testing?

* Giving useful **combinations** but **taking less time** than trying all the combinations.
* Detecting more bugs.



1. How are different levels related, in terms of testing strength and size?

The more levels, the more bugs can be detected, but it also will increase costs.

Two and three pairs can increase efficiency rapidly, but it need less cost.

1. What are three purpose for having such finite functional models?

* Capture **specs**
* Inform what tests cases can and should be written
* Inform a “criterion” for what is “**enough**”

1. CFG concepts (including "peculiarities": for loops, switch, break, continue, ternary operator)

The graphical representation of **control flow** or **computation** during the execution of programs

1. Short-circuiting: What is it? How can it affect control flow

The compiler skips the execution or evaluation of some sub-expressions in logical expression.

1. Interprocedural? Intraprocedural?

* Interprocedural: **Across** procedural boundaries, procedural call, shared globals, etc
* Intraprocedural: **Within** a single procedure, function or method( intramethod)

1. What are three purpose of Control Flow Graph?

* Inform what tests cases can and should be written
* Inform a “criterion” for what is “**enough**”
* Allow for analyses and automated assistance

1. What would one want to perform structural testing?

* One way of answering the question of “**what is missing in our test suite**”?
* Occurs **after** functional testing
* Any statements that have never been executed could have bugs
* To increase confidence

1. Is structural testing enough? If we had full structural coverage, can we guarantee that the program is correct? If we don’t have full structural coverage, we confidence do we have in the uncovered parts of program?

* Structure testing will **never** focus on code that **isn’t** there
* Structural testing **complements** functional testing
* Executing all control flow elements does **not guarantee** finding all faults
* **Structure coverage** increases confidence in thoroughness of testing

1. What is the typical process that would utilize functional and structural testing?

* Create functional test suite first, then measure structural coverage to identify what is missing
* Interpret unexecuted elements
* Attractive because **automated**

1. What are the types of structural coverage that we discussed? What are each of their strengths? Are some subsumed by others?

* Statement testing
  + Each line of code must run **at least** once during testing.
  + **100% node coverage=100% statement coverage**, but **levels** will likely differ in between
  + Traversing all edges causes all nodes to be visited, converse not true
* Branch testing
  + **Stronger** than **statement coverage**
  + **100% branch → 100% statement, reverse is not sure**
* Condition testing
  + **Branch coverage** exposes faults in how a computation has been decomposed into cases
  + Consider case analysis in more details
  + Exponential complexity
* Path testing(beyond individual branches)
  + Many more paths than branches
* Method-level testing
  + Method coverage
    - Each method is executed at least one
    - Comes “**for free**” with unit testing
* Method-call coverage
* Each method can be called from multiple call sites, and each call site can call many methods
* Method-exit coverage
* Method may have multiple exit points

1. Why might is be impossible to cover all statements in your program? How might we handle these situations?

* Sometimes criteria may not be **satisfiable**.
* Large amounts of **dead code** may indicate serious maintainability problems
* Solutions:
  + Make **allowances** by setting a coverage goal **less than 100%**
  + Require justification of elements left uncovered
  + Refactor and eliminate the **dead code**

1. What is program instrumentation? Why would any one want to instrument a program?

* The process of adding code to a program(called **probes**) when the program is executed, it **records information** about its execution

1. What is the difference between coverage, profiling and tracing?

* Execution trace: some sequence of **events** occur
* Execution profiling: some **quantity** that an event occur
* Execution Coverage: **whether** an event occur

1. **Can coverage profiling and tracing be derived from the others? If so, which can be derived form which?**
2. What are the runtime costs of each coverage, profiling and tracing?

* Trace is the slowest, profile is not slow(data structure), most easy is coverage(every sentence are zero)

1. What is an example of an instrumentation tool?

GCC/Gcov, Cobertuna, Jacoco, EclEmma

1. Does instrumentation change the behavior of the program?

* No
* **Should** preserve the behaviour of program (but may not succeed)
* Should only add extra features which inadvertently makes changes

1. What are some strategies for integration during development?

* Daily build
* Smoke test
* Continuous integration

1. What are some of the challenges of integration?

* Combined units can fail in more places and in more complicated ways
* How to test a partial system where not all parts exist?
* How to “rig” the behavior of unit A so as to produce a given behavior from unit B?

1. What is a stub? Why do we need them? What are they useful for?

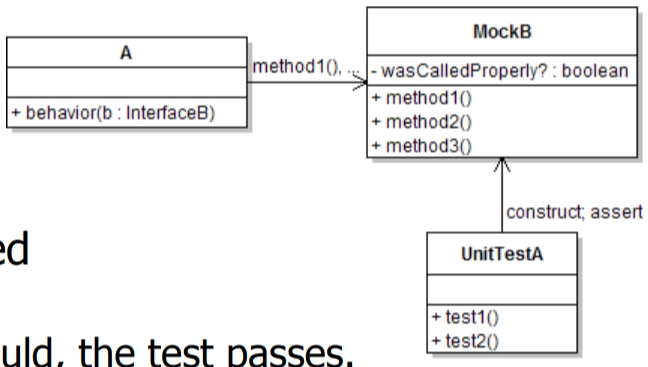
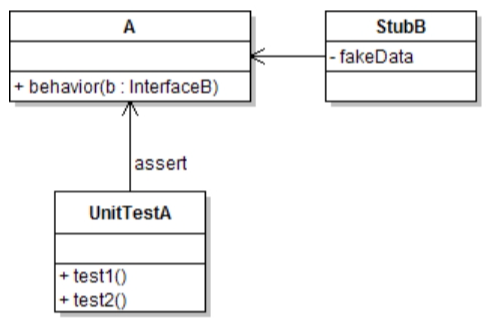
* A **fake/partial** implementation just to allow test cases to run ex.
* Useful for simulating **difficult-to-control** elements, useful to avoid side effects of real systems
* Difficult-to-control elements: network, database, time-sensitive code, files, threads, memory

1. What is a mock, Why do we need them, What are they useful for?

* A fake object that decides whether a unit test has passed or failed by watching **interactions** between objects
* Similar to stub except provides extra **recorded information** than stub
* Useful for **interaction testing** (as opposed to **state testing**)

1. How are stubs and mocks different?

* A stub gives out **data** that goes to the object/class under test, the unit test directly asserts against class under test, to make sure it gives the right result when fed this data
* A mock waits to be called by the class under test(A), it makes sure that it was contacted in exactly the right way



1. What are some strategies for waiting your code that make it easier to test, particularly when it comes to integration?

* Avoid complex private method
* Avoid static method
* Be careful hardcoding in “new”
* Avoid logic in constructors
* Avoid singleton pattern

1. What is continuous integration?

Adding new units immediately as they are written

1. What is continuous deployment/delivery?

* an extension of continuous integration since it automatically **deploys** all code changes to a **testing or production** environment **after** the building stage.

1. What are some common ways that developers use continuous integration systems?

* Maintain a single source code repository
* Everyone commits to the main line **every day**
* Every commit should build the **main line**
* **Automate** the build
* **Automate** testing
* Keep the build and testing fast
* Clone the **production** environment in the integration machine
* Make it easy for anyone to get the **latest** executable
* Everyone can see what’s happening
* **Automated** deployment

1. What is automated GUI testing? Why do it?

* The usage of specific tools or frameworks for automating the manual test processes that verify UI functionality.
* Increasing efficiency, Improving test coverage, reducing human error, consistency, improving accuracy, scalability, cost-effective, continuous integration

1. What part of automated GUI testing that almost always needs extra human specification?

* Capture of pixels means **layout changes** can cause troubles
* All **environment** events can play a part in test behavior
* **Timing** can play a part in test behavior... allowing replay at user speed would take time

1. What is code review and how does it work? What are some of the difficulties with doing these, and what are some approaches to try to avoid such difficulties?

* Usually informal manual-human reviews of code, way of find **potential problems** early
* Reviewing **small** portions of code at a time, recording all feedback, review code independently **before** gathering to discuss, using checklist
* Code reviews can be unpleasant experiences for the reviewee
* Keep it **professional** and about the code, remember to notice and mention the **good choices** as well as the **critique**

1. What is a static analysis?

It is done **without** running the program

1. What are some best practices and etiquette for code reviews?

* Best practice
  + Reviewing **small** portions of code at a time
  + Recording all feedback
  + Review code independently **before** gathering to discuss
  + Using **checklist**
* Etiquette
  + Code reviews can be unpleasant experiences for the reviewee
  + Keep it professional and about the code
  + remember to notice and mention the good choices as well as the critique

1. What are “static analyzers” and how do they relate to code reviews?

* Tools that can perform some code review features
* Has prepackaged patterns of **bad coding**, will scan all of program

1. What is mutation testing? What is the purpose of mutation testing?

* Design new software tests and also evaluate the **quality** of already **existing software tests**
* Purpose:
  + Determine the **power** of your test suite
  + When weaknesses are found, identify ways in which test suite can be improved
  + **Who** will test the tests

1. What is the difference of strong and weak mutation testing?

* PIE model of failure: **execution, infection, propagation**
* **Strong**: the traditional way described thus far--all three situations are needed
* Weak: requires only execution and infection
* In the equivalent mutant, weak mutation is the strongest that can be expected

1. Be able to define the terms: “mutant”, “kill”, “mutant operator”

* **Mutant**: a single version of a **mutated program**
* **Kill**: at least one test cases fails in a way that it does not on the **un-mutated** program
* **Mutant operator**: one of a set of systematic recipes for mutating the program. Ideally, these are designed to simulate real **bugs** or to target particularly problematic types of bugs

1. What is the difference of testing and debugging?

* Testing will **reveal** bugs by seeing failure, debugging is the process of identifying what/where the bug is and then **fix** it

1. What are some techniques/tools/strategies that developers use to help them debug their programs?

* Incremental development
* Instrument program to log information**(print statements)**
* Instrument program with assertions
* Use debuggers**(break points)**
* Backtracking
* Binary search
* Form hypotheses and test them

1. Tool

* JUnit
  + @Test annotation, assert method
  + A set of input: @parameterizedTest and @ValueSource
* Instrumenters(coverage)
  + GCC/Gcov, Cobertuna, Jacoco, EclEmma, Clover, Java JVM TI
* BDD
  + JBehave, Cucumber, RSpec, CBehave, Jasmine, Spock
* Capture/ Repay
  + Ranorex Studio, Abbot, Selenium, Jubula, TestComplete
* Continuous Integration
  + Jenkins, Cruise Control, Continuum, Hudson, TravicCI, CircleCI, Github Action
* Static analyzers
  + SpotBugs/FindBugs, PMD, CheckStyle, Facebook Infer, SonarQube
* Mutant
  + PIT, muJava, muClipse, Jumble, Javalanche, JavaMut, Jester