CS 1632 Exam 1 Study Guide

Equivalence Class Partitioning

**Equivalence class** – a natural grouping of values with similar behavior or belonging to the same category

Sqrt() – would be negative numbers, 0, positive numbers

You could further separate into pos/0/neg and also whole and not whole numbers

Interior and Boundary values – TEST BOTH

“HIDDEN” boundary values – MAXINT, MININT, max precision of float, allocation limitation of hardware, undefined values

**Base Case** – an element in an equivalence class that is not around a boundary, AKA expected use case

**Edge Case** – An element in an equivalence class that is next to a boundary value, AKA an unexpected use case

**Corner Case** – A case which can only occur outside of normal operating parameters, or a combination of multiple edge cases.

Black White and Greybox testing

**Black-box testing -**  Testing with no knowledge of the interior structure or code of the application. Tests are often performed from the users perspective, looking at the system as a whole.

Examples – accessing a website using a browser, to look for flaws

Running a script against an API endpoint

Changing fonts in Word produces correct Font

**White-box testing** – Testing with explicit knowledge of the interior structurre and codebase, and directly testing that code. Tests are often at a lower level. (testing individual methods or classes)

Examples – Testing a function returns correct result

Testing instantiating an object creates valid object

Checking that there are no unused variables in a method

**Grey-box testing** - Testing with knowledge of the interior structure and codebase of the system under test, but not directly testing the code. Tests are similar to black-box tests, but are informed by the testers knowledge of the codebase.

Examples – Reviewing code, and noticing that bubble sort it used, then write a user facing test involving a large input size

Reviewing code and noticing an off-by-one error. Write a user facing test which checks that boundary value.

Static vs Dynamic Testing

**Static testing**  - code is not executed

Code walkthroughs and reviews

Requirements analysis

Source code analysis

Linting – using variable before value assigned to it

Model checking

Complexity analysis

Code coverage

Finite state analysis

**Dynamic testing** – code is executed

Code executed with input values/environment variables

Observed results compared with expected results

Requirements Analysis

**Requirements**  - Specifications of the software. They are how developers know what code to write, and how testers know what to test.

Requirements describe expected behavior!

Examples – Bird cage must be 120cm tall…….. what about 120.000001cm

Cage has no door…OK?

Case has 17 doors all opened by elaborate puzzles…OK?

**Verification –** did we build the software right? Requirements met, no unexpected fails

**Validation ­**– Did we build the right software? Ensure that the software does what customer actually wants

**Good Requirements**

Complete – cover all aspects of a system, be comprehensive

Consistent – Requirements shouldn’t contradict each other internally and externally

Internally – requirements don’t contradict each other

Externally – requirements don’t contradict world outside system

Unambiguous – using undefined term, like default value

Quantitative - system should be responsive to user, vs result should be displayed to user in less than 1 second for 99% of queries

Feasible to test – System shall process 100TB data in 4000 years vs 1mb data in 4 hours.

**Functional Requirements** – Specify the functional behavior of the system (the system shall do X under condition Y.

System shall return String NONE if no elements match query

**Quality Atributes –** (Non-Functional) – Specify the overall qualities of the system, not a specific behavior.

System shall be protected against unauthorized access

System shall be portable to other processor architectures

System shall have 99.999% uptime and be available at that time

Categories:

Reliability - converts to mean time between failures

Usability - convers to amount of training time

Accessibility

Performance - transactions per second

Safety – accidents/year

Supportability

Security

Converting Qualitative to Quantitative

The system must be highly usable vs over 90% users have no ?s after 1hr training

Test Plans

**Test Plan -**  A collection of test cases that lay out an entire plan for testing a system under test

**Test Case** - the individual tests that make up a test plan.

**Test suite** - Test plans are usually not as large as all of the requirements for a program; once they get big enough, test plans are grouped under larger test suites.

Test Case mainly consists of:

Preconditions

Execution Steps

Post conditions

**Test Case Template:**

**IDENTIFIER**

**DESCRIPTION**

**PRECONDITIONS**

**EXECUTION STEPS**

**POST CONDITIONS**

The test plan consists of all of the test cases

Test run – an actual run through of a test plan or test suite.

Statuses:

PASSED

FAILED

PAUSED – Not yet finished, going to start again

RUNNING

BLOCKED- not yet ready to be tested, waiting on something

ERROR

Types of errors

Logic Errors

Off-by-one errors

Floating point errors

Integration Errors

Errors of assumption

Missing Data errors

Bad data errors

Display Errors

Null pointer errors

I/O error

Configuration Errors

Verification vs Validation

**Verification** – ensuring that you’re building the software right

System doesn’t crash

meets the requirements

handles failures gracefully

**Validation** - - ensuring that you’re building the right software

Ensuring requirements meet actual customer needs

Does the software do what user wants

Are there gaps in requirements

Defects

Defect Life Cycle

Discovery

Reporting

Triage / Assignment

Fixing

Verification

***Defect Template –***

**SUMMARY**

**DESCRIPTION**

**REPRODUCTION STEPS**

**EXPECTED BEHAVIOR**

**OBSERVED BEHAVIOR**

**Severity/Impact\***

**Notes\***

Coding mistakes vs defect

From a definitional perspective, a defect is something that impacts the functionality of a program

**List of Severity –**

BLOCKER

CRITICAL

MAJOR

NORMAL

MINOR

TRIVIAL

Traceability Matrix

Traceability Matrix – A list of requirements and the associated test case

LOGIN\_REQ: GOOD-LOGIN-TEST, BAD-LOGIN-TEST, THREE-X-ERROR-TEST

DATABASE-REQ: VALID-QUERY-TEST, INVALID-QUERY-TEST, DV-DOWN-TEST

LOGIN-SCREEN-REQ: LOW-BANDWIDTH-TEST, HIGH-BANDWIDTH-TEST

CALC-REQ: - ADD1-TEST, ADD2-TEST, ADD2-TEST, SUBTRACT1-TEST

Smoke, Exploratory, and Path-Based Testing

**Exploratory testing –** Testing without a specific test plan, in which the goals are to both learn more about the system and inform the development of system by finding defects and possible enhancement

Benefits – fast, flexible, easy to update

Drawbacks – unregulated, unrepeatable, coverage, cant automate

**Smoke testing –** Some minimal testing to ensure that the system is, in fact, testable or ready to be released.

**Scripted –** a few small but important test cases run before software is ready to be tested. Auto or manual

**Unscripted­** – Experienced tester doing exploratory testing quickly

To ensure minimum standards met

**Path-based testing** – testing all possible paths

Example – Pick red or blue car, one or other always wins

Add easy/hard modes = more paths to test

Higher complexity == more chance of defects

McCabe Cyclomatic Complexity –

< 10 is very simple, low risk

> 50 is very complex, high risk

Automated Testing vs Manual Testing

Manual Testing

Pros – simple, cheap, easy to set up, no additional software to learn, flexible, humans catch issues programs may not notice

Cons - Boring, can be unrepeatable, some things are hard to test manually, like timing, individual methods, low level interfaces

Automated Testing

Pros – No chance of Human error, fast execution, easy to execute, repeatable, less resource intensive

Cons- Extra time up front to set up, may not catch user facing bugs, learning tools and frameworks, skilled staff, only tests what it is looking for!

Unit tests vs System Tests

System test – checks that the whole system works.

Unit Test – check that very small pieces of a functionality work, not that the system as a whole works together

Unit testing involves smallest “units” of code, such as functions, methods, or classes

Examples:

.sort() method sorts elements

passing nil/null throws exception

formatNumber formats properly

.send and .receive exist for a class

Why Unit tests?

Problems found earlier

Faster turnaround time

Living Documentation

Tells if changes caused issues elsewhere running entire test suite

Complexity – higher or lower complexity may be easier or harder to test

Unit Testing with Minitest

Unit tests consist of:

Set up code

Preconditions

Execution Steps

Postconditions/**Assertions**

Tear down code\*

Assertions are the expected behavior of a Unit test, when you execute, you find the observed behavior.

Assert\_true Refute\_true

Assert\_equals Refute\_equals

Assert\_includes Refute\_includes

Assert\_nil Refute\_nil

Usually only one or two assertions in a Unit test, you are testing a small bit of code

Tests are run in random order

**Double –** fake objects you can use in your tests to “stand in” for other objects in the code

Hides piece of the codebase that may not work

Allows you to localize the source of errors

Tests for one method should fail only with issues in that method, not ones it depends upon

**Stub -**  fake method

**Mock -**  allow you to assert that a particular method was called on a mocked object

Assert less, so you test specific things, and name those tests specifically

Unit Tests should be independent. Order shouldn’t matter, one test wont effect another

Test Driven Development (TDD)

**Test Driven Development** – A strategy for developing highly tested, quality software

A software development methodology that comprises:

Writing tests BEFORE writing code

Writing ONLY code that is tested

Writing ONLY tests that test the code

A very short turnaround cycle

Refactoring early and often

Red-Green-Refactor loop

Red – write a test for new functionality – should fail immediately

Green – write only enough code to make the test pass

Refactor – Review code and make it better

Benefits

Automatically create tests

Makes writing tests easy because its done often

Tests are relevant

Developer is focused on end result, not code

Code is extensible

Large test suite automatically created

Confidence in codebase

Drawbacks

Focus on unit tests may mean other aspects of testing get short shrift

Extra up-front time

May not be appropriate for prototyping

Hard to do large architectural changes – some things not possible in small steps

Complex or mission/life critical systems need more robust test strategy

Test become part of project overhead

Overtesting risk

Can be difficult to implement on projects started under different paradigm

Writing Testable Code

**Testable Code** – Code for which it is easy to write and perform tests, automated and manual, at various levels of abstraction, and track down errors when tests fail.

Strategies – give yourself something to test, RETURN VALUES are useful

DRY principle – don’t repeat yourself, refactor, don’t have multiple similar methods Make generic methods that apply in as many places as possible

**Pure function -** Output depends only on input arguments

**Impure function -** Output depends on more than just inputs

Not deterministic, may depend on file read/write

**Seams -** Places where behavior can be modified without modifying code

**Dependency Injection** – Passing in any objects that a method relies on

In other words, favor arguments to methods as opposed to instance variables or internally generated objects

Allow you to easily double/stub/mock objects

SEAM – dependencies injected

def printDoc printer, document, arguments

printer.setArgs arguments

printer.print documents

end

NO SEAM – no dependency injection

Def print\_doc\_

Document = generate\_doc

P = Printer∷new [:doublesided]

p.print document

end

Pairwise and Combinatorial Testing

**Pairwise testing –** We are testing all possible pairs of interactions

Not-bold/ not-italic

Bold/not-italic

Not-bold/italic

Bold/italic

**Combinatorial Testing -**  testing all possible of n-large combinations

NIST only found errors when checking all six-way combinations

Performance Testing

**Service-Oriented** indicators measure how well a system is providing a service to the users. Measured from a specific users point of view

Availabiltiy

Response Time

**Efficiency-Oriented** indicators measure how well the system makes use of the computational resources available to it. This is looking at the performance from a more developmental perspective.

Throughput

Utilization

**Performance targets –** quantitative values that the performance indicators should reach

**Key Performance Indicators –** most important indicatorsperformance targets

**Performance Thresholds –** the absolute minimum performance level a system can reach and be considered production ready

Performance testing relies heavily on coding, automation, and statistics

**User time –** time user code executes

**System time –** time kernel code executes

**Total Time –** user + system

**Real time-** actual time taken

NINES –

one 9 = 90%

two 9 = 99%

three 9 = 99.9%

**Model system to test**

Load testing – how many concurrent users/work can system handle

Baseline test – bare minimum use for baseline

Soak/Stability test – run for extended time at low levels of usage

Stress test – high levels of activity