**This chapter will be NB and have a big mark question asked.**

**In text book, split between 3 chapters.**

**Discussions and questions in class are important for tests**

**Software Testing Strategies**

Road map that describes the steps conducted as part of testing. Including how much effort, time and resources that will be required.

This is normally done before project coded (future estimations), so based on previous projects statistics.  
Based more on qualitative than quantitative data. As Normally don’t have the needed metrics to get 100% accurate.  
-Gives info on team, product. Depends where we use values, changes where it applies.

**A strategic approach to software testing**

A set of activities that can be planned in advance and conducted systematically. A template for software template should be defined for the software process.

The testing goes hand in hand with the software process.  
-eg Agile has continuous testing as developed

Must know the characteristics of testing to do this well.

**Characteristics**

Error Defect Model (Error elimination): Should conduct effective technical reviews.  
-benefit of conduction reviews to eliminate errors.  
-Reviews catch errors before they cause issues.

Begins at component level and works “outwards”.  
-You don’t want to approach testing strategy as nothing to testing everything.  
-Do small amounts, over time (incremental testing).  
-This is also to catch any propagating, cascading errors.

Different testing techniques are appropriate for different software engineering approaches and at a different point in time.  
-Testing strategies aren’t cookie cutter.  
-A team will develop their own testing style.  
-There are different tests, for different stages of project.  
-Goal: Increase efficiency, reduce errors  
-Testing starts at the design phase

Testing is conducted by developer and independent testing groups.  
-These independent groups will **test both code** and **if requirements** are met. They sit with normally design document + client.

Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.  
-Testing **identifies** **errors**  
-Debugging **identifies** **cause** of error itself  
-Limited budget + time scale will result in not being able to fix every error. The strategy, if implemented properly reduces errors at end of day.

**Strategies must include**

Low level tests…

…

**Verification and Validation**

Verification refers to the set of tasks that ensure software correctly implements a specific function.  
-Are we building the product right? What we delivered, is functioning correctly.

Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.  
-Are we building the right product? What we delivered, matches what the customer wants.

V and V include a wide variety of SQA activities. While testing plays an important role in V&V, to is not the only activity.

Quality can be assessed from testing and errors being uncovered. However testing should not be viewed as a safety net.  
-Testing cannot show something is Quality.  
- No Errors != Quality

Quality, you can evaluate on a scale, but not give a mark from that scale.

**Organising for software testing**

There is an inherent bias for testing your own code.

Developer tests if the program is error free for the requirements customer sets, vs independent tests that test for everything.

From **psychological** viewpoint:

Software engineering produces software products: You are proud of what you build. --Would you use it or not? This is a good test. As you “try” to build a good product.

Testing: Attempt to break the product – can be considered as (psychologically) destructive.   
-You are testing to try break system, not if system is good  
-You must make sure that you point out where needs more work to not break down team spirit, and to fix errors.

Builder designs tests to demonstrate that the product works rather than to find errors.  
-As builder you test whether it works, not to find errors.

3 main **misconceptions**:

Developers should not do any testing.

Software will be “tossed over the wall” to strangers who will test it mercilessly.

Testers get involved only when testing is about to begin.  
-They are involved when product is being developed and the requirements extraction discussion so as to assist in setting up testing strategies and to know what to evaluate.

**Criteria for completion of testing**

When are you done testing? Think about it…

-You are never done testing; the burden simply shifts from you to end user.

-You’re done testing when you run out of time or money.

--Product no longer supported

--Maintenance, post implementation, is where most money is at.

**Test strategies for Conventional Software – also big for exam and ST2 probs**

There are 2 extremes:

Code everything, then test everything at once at end.

OR

Code something, then test it (daily basis).

**You want a middle ground:**

**Unit testing**

Focusses verification efforts on the smallest unit of software design. You are testing a singular thing, eg A function

The testing uncovers errors within the boundary of the module.

**5 main thing you look at – Unit test considerations**

**Module interface:** tested to ensure info properly flows in and out of the program.  
-**Black box**: only consider input and output

**Local data structures:** Examined to ensure the data stored temporarily maintains its integrity during all sept in an algorithms execution. (Ensure you use protected, private classes are used and stored properly)

**Boundary conditions:** Test the upper and lower boundaries of what you are expecting from boundary to make sure it operates within the bounds. (Plus processing requirements). This is so you can handle the min and max.

**All independent paths: White box testing**: Looks at every single path of module and ensures every path can be executed and is executed correctly.

**All Error handling paths:** Are handled

These test units not only test for functionality, but for where errors occur.

**Unit test procedures**

UTis normally considered as an adjunct to the coding step.

Can be designed before coding begins or after source code has been generated.

Each test case should ne complied with a set of expected results.

**Disadvantage:** Has one flaw, where is the info going, and where is it coming from.

You have a:

**Driver:** Adds input data to module being tested

**Stub:** Receives output from function

**Integration Testing**

When you start combining modules that work individually, to test if they work together.

Systematic technique: Constructing the modules and building them up into different clusters/collections and then testing them

**Incremental integration vs non-incremental**

Incremental: Test a module, Add another test it, ect

Non-Incremental: test all at once.

Two approaches

**Top down approach:** Start with main program, and work you way down.  
-Algorithm: Breadth/depth first search.

5 steps:

1. Main control module needs a driver, and the stub
2. When you test module 1, you have eg 3 stubs, where the stubs are replaced **1 at a time** with the following modules (eg module 2,3,4)
3. Tests are conducted as each component is integrated. Driver remains at top
4. W
5. Regression testing, when you add a new module, you test all its predecessors as well.

Repeat 2-5 until complete

Verifies major control or decision points early on in the test process.

**Bottom up integration:**

Opposite of top down approach.

Group into clusters to test.

Start at bottom.

You cluster modules that have an integration with each other.

Start with drivers and stub, and you move up as you have success.  
-combining these into a cluster.

Steps:

1. Low level components are combined into clusters to perform a specific function
2. A driver is written to coordinate test case input and output
3. The cluster is tested (not individual modules)
4. The drivers are removed and clusters are combined moving up the structure

Can be combined with top down to reduce number of drivers needed and to simplify integration of clusters.

**Regression testing**

Re-execution of modules that have already been tested, but with the new module added.

Can have capture tools that can be played back to see how the results change as they progress in the subset.

**Has 3 main test cases we look at:**

Representative samples: Run input and test output

Additional test: That are likely to be affected by change.

Tests that focus on software components that have been changed.

**Smoke testing (where there is smoke, there is a fire)  
-Does not test for functionality, but errors**

Commonly used when product software is developed.

Looks for system critical errors/system breaking errors (“show stopper errors”).  
-These errors have the highest likely hood of making the project go behind schedule.  
-Low budget/time projects use this.

You will have a series of tests that try break the system.   
-You try test out of the boundaries.

-The build is integrated with other builds, and entire project is smoke tested daily.  
--Can be top down or bottom down  
--This daily testing gives a realistic assessment of integration testing progress

Benefits:

Integration risk is minimized: As you can identify show stoppers and fix immediately

The quality pf the end product is improved: As you start getting to a **good enough** state.

Error diagnosis and correction are simplified: As you know where the errors are coming from.

Progress is easier to assess: As you can easily where the progress comes from.

**Integration test work products**

Description of tests are documented in a test specification.

Testing is divided into phases that build and address specific functional and behavioural characteristics.  
-So that its ready for client to review/test/provide feedback