SCHULICH School of Engineering

UNIVERSITY OF CALGARY

SENG 637
Dependability and Reliability of Software Systems

Chapter 2: Foundations of Testing

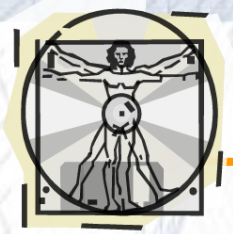
Department of Electrical & Software Engineering, University of Calgary

B.H. Far (far@ucalgary.ca)

http://people.ucalgary.ca/~far

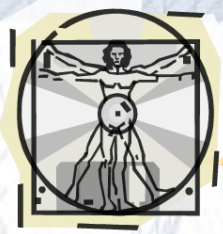
Contents

- Introduction to testing
  - What is software testing?
  - Manual (human-based) software testing
    - Ad-hoc, exploratory, scripted testing
  - Manual regression testing
  - Bug report
- Software development life-cycle
- Software testing life-cycle

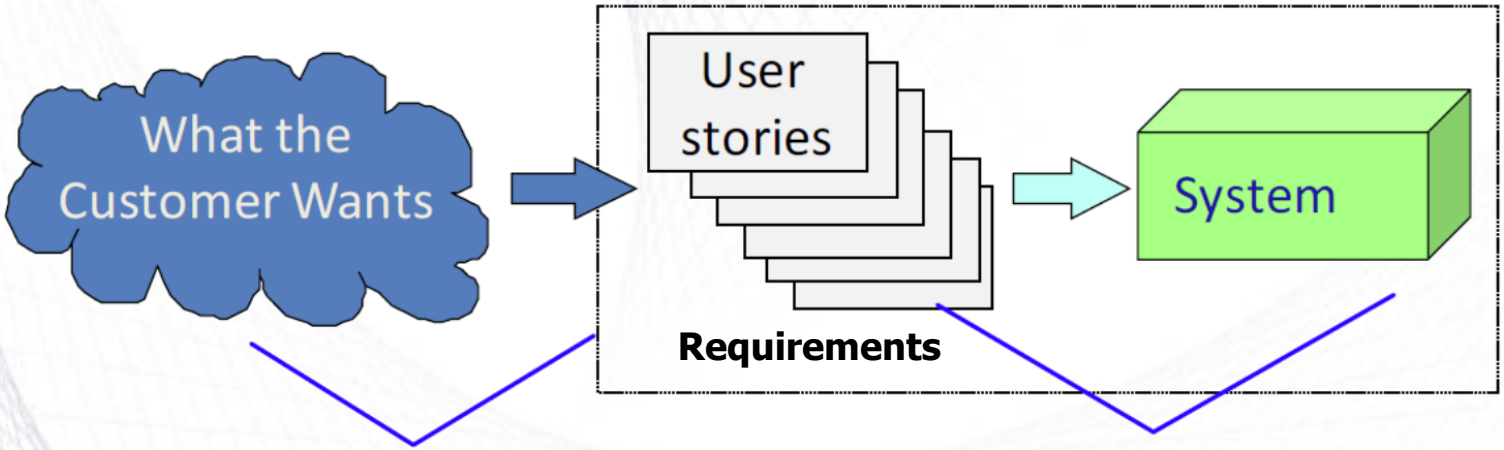


What Do We Know So Far?

- Software errors/faults are inevitable
- Software failure are expensive and sometimes life threatening
- Testing is a practical approach for finding faults to avoid failure
- The need for software testers and test engineers is growing fast
- We learn the basics of testing in this class



Validation & Verification

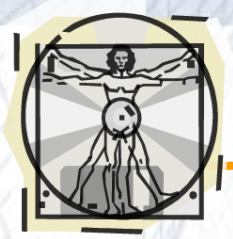


Validation: checking whether the system meets customer's actual needs

Building the right product

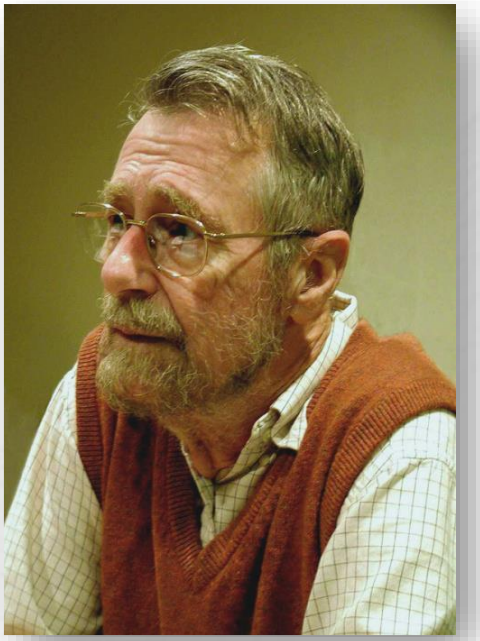
Verification: whether the system is well-engineered, bug free, etc.

Building the product right

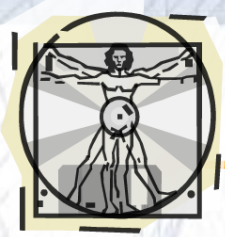


Testing

- Software Testing: Techniques to execute programs with the intent of finding as many defects as possible and/or gaining sufficient confidence in the software system under test.



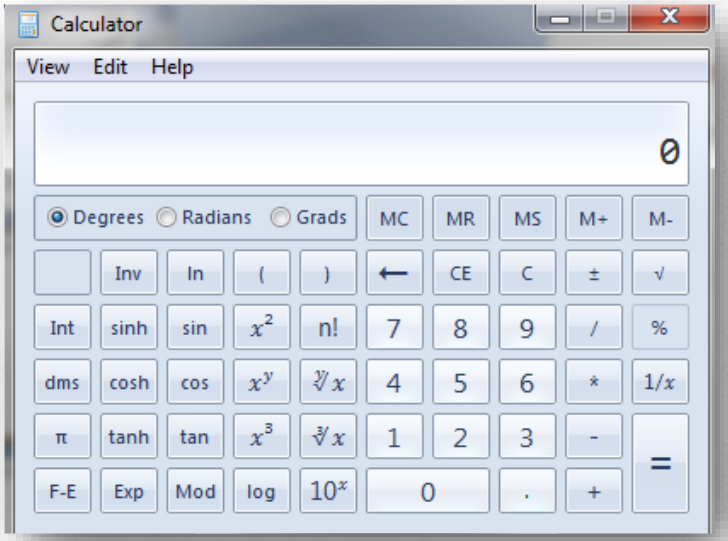
“Program testing can show the presence of bugs, never their absence” (Dijkstra, 1972)

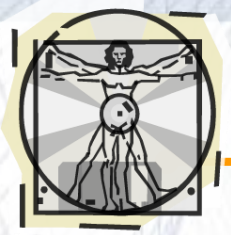


Brainstorming

- How would you test a Calculator app?

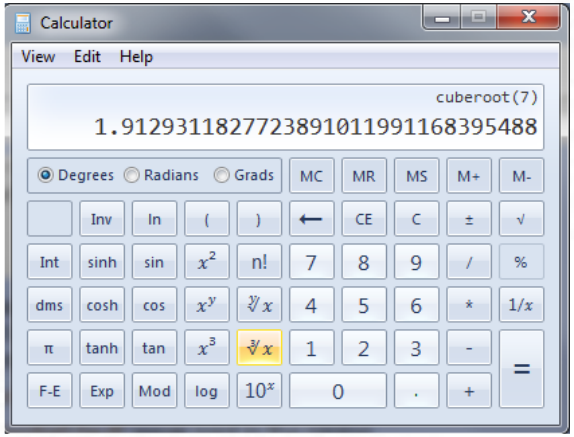
- Contextual knowledge
- Test plan
- How many inputs (test cases)?
  - So MANY possibilities
  - Tedious!
- How to know the outputs are correct?
  - Compare actual with correct outputs



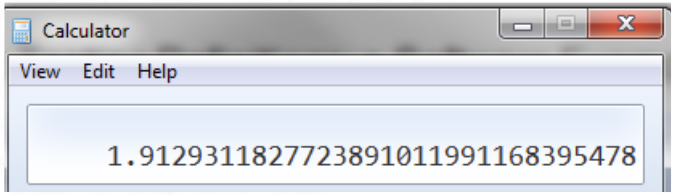


Brainstorming

- How to know the outputs are correct?
  - Cube root of 7 in Calculator
  - This the expected value:



  - But a manual tester runs and gets:

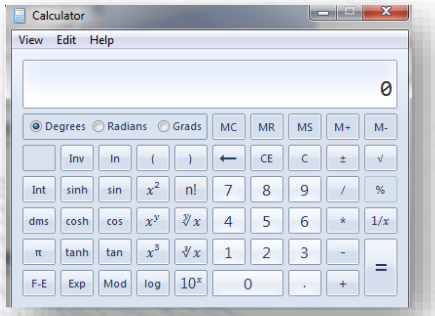


  - We need to set the correct expected output!

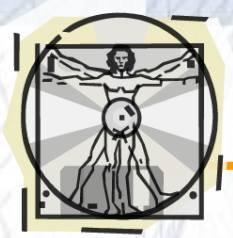
Test Case & Test Suite

- A **test case** is set of inputs and the expected outputs (oracle) for a unit/module/system under test
- A **test suite** (test set) is set of test cases
- Without the expected outputs, a test case is NOT complete

  e.g., for Calculator:

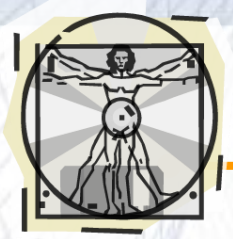


TC#	Unit under test	Input 1	Input 2	Expected output
1	+	5	6	11
2	+	16	0	16
3	/	5	2	2.5
4	/	4	0	Cannot divide
	...			
	So many more test cases!			



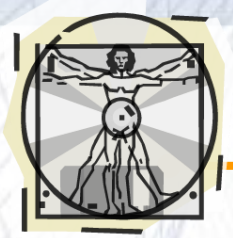
Test Case & Test Suite

- **Direct input variable:** a variables that controls the operation directly
- **Example:** arguments, selection menu, entered data field
  - Important during exploratory/unit test
- **Indirect input variable:** a variable that only influences the operations or its effects are propagated to the operation
  - **Example:** traffic load, environmental variable
  - Important during integration testing



Test Case

- A “test case” for a given functionality is specifying its direct input variables and the expected outcome
- Specification of the indirect input variable gives a test case the necessary ***context***
- During initial test stages (e.g. unit testing and regression test), the influence of indirect input variables should be kept to minimum to ensure that the function being tested is reliable
- Indirect input variables are effective during integration and system test



Example

- Example of direct and indirect input variables for a functionality to be tested
- Telephone dialing system

Direct	Indirect
Originator = 201 908 5577	Operational mode = prime hours
Forwardee = 908 555 1212	Database state: signified by time
Billing type = per call	Resource state: signified by time
Dialing type = standard	
Screening = yes	

Example from Musa's Book

Example

- Test case is specified with its direct input variables and expected outcome
- In theory, it is possible (but may not be practical) to record all the input variables needed to initiate the runs that make up the whole execution space of the software

Direct
Originator = 201 908 5577
Forwardee = 908 555 1212
Billing type = per call
Dialing type = standard
Screening = yes
Expected outcome: pass

Example from Musa's Book

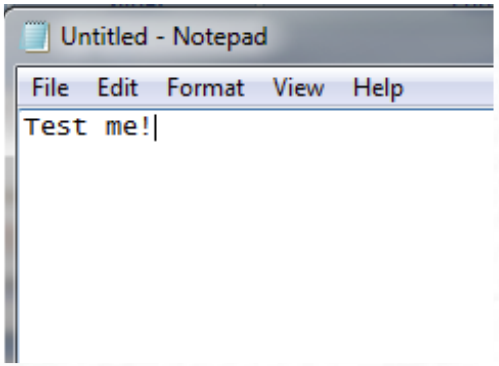
Brain Storming

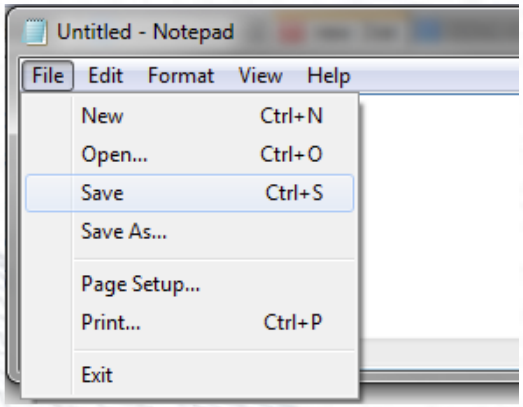
- How will you test the Notepad or Word?

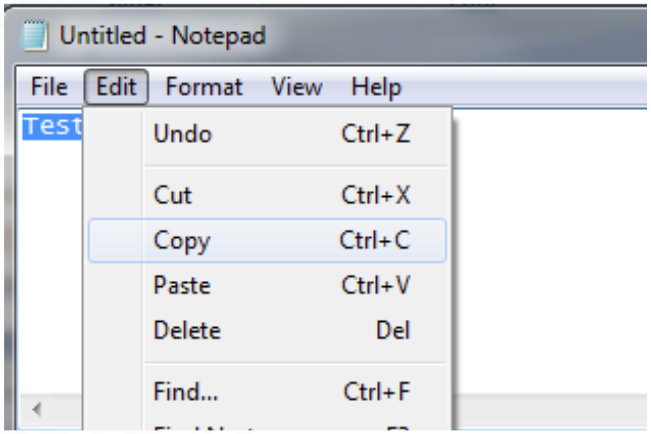
Planning:

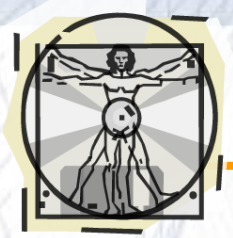
– similarities – differences

- Oracles
- Level of testing (i.e. small or large)
- Test case and test suite





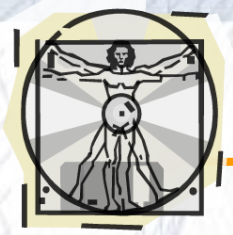




Test Case & Test Suite

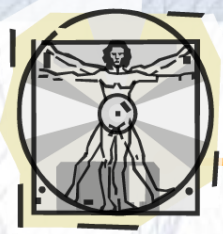
- The definitions hold for test cases in any level, unit, integration or system testing
- Example, for Microsoft Word:

TC#	Feature under test	Inputs	Expected output
1	Open	Valid DOCX file	File opened properly
2	Open	Invalid DOCX file	Error message
3	Open	RTF file	File opened properly
4	Open-Edit-Close	Open an empty DOCX file - Append "this is test" Close	The save message should appear.
	So many more test cases!		

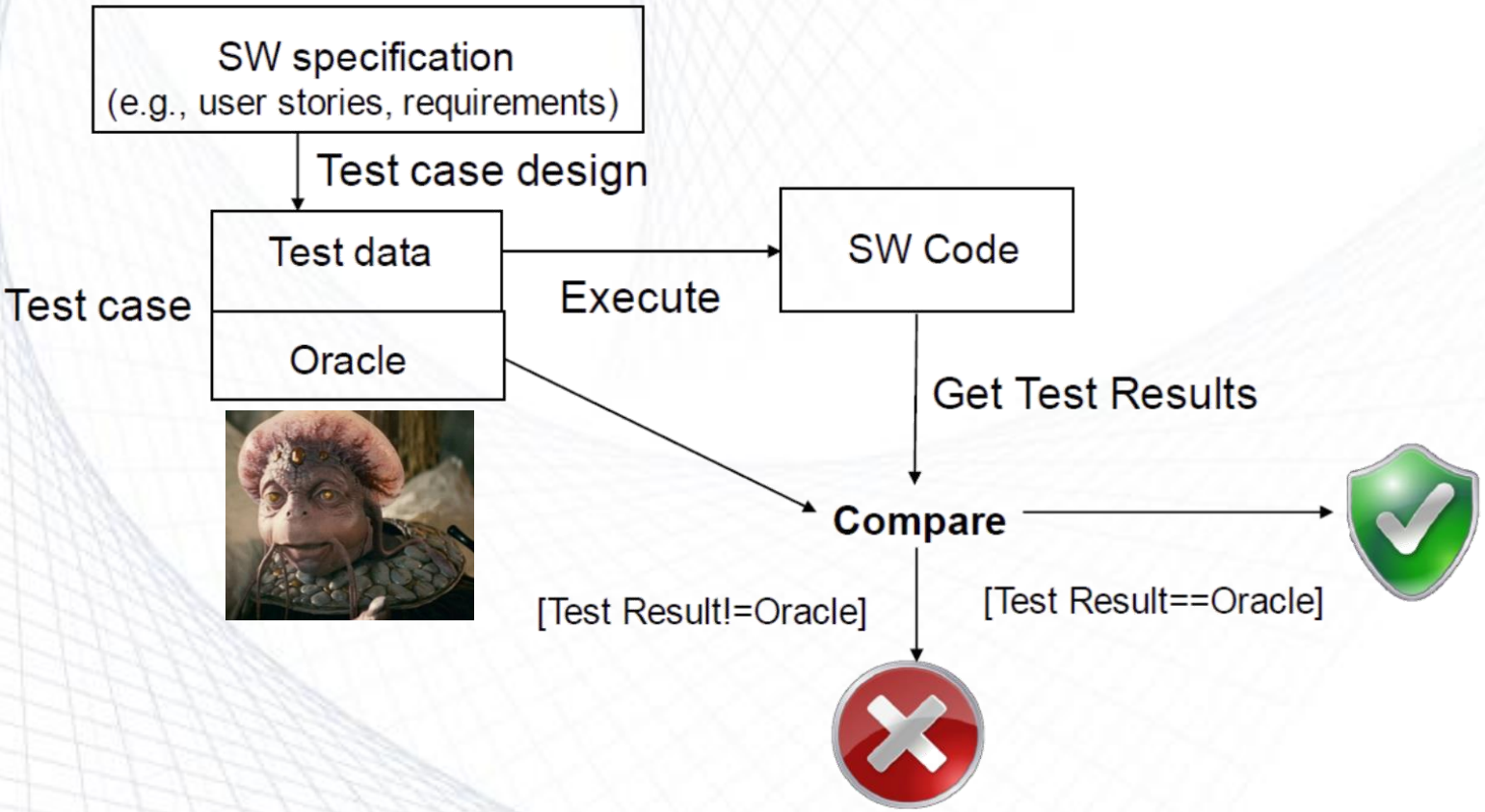


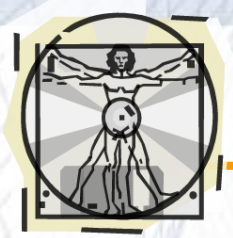
Test Case Design

- The art and science of deriving efficient but effective number of test cases is called "test case design"
- "Less is more!"
  - Less test cases, but maximizing the chance of finding faults



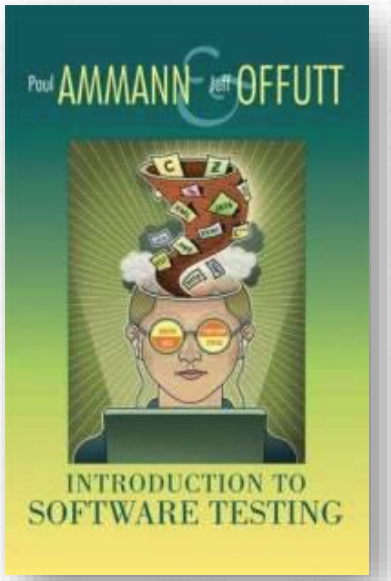
Testing Process Overview

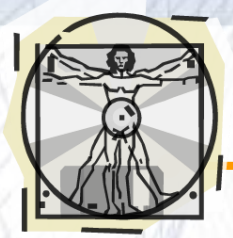




Types of Test Activities

- Test-case Design: Exploratory (human-based)
  - Design test values based on domain knowledge of the program and human knowledge of testing, exploratory testing
- Test-case Design: Criteria-based
  - Design test values to satisfy coverage criteria or other engineering goal
- Test Automation
  - Embed test values into executable scripts
- Test Execution
  - Run tests on the software and record the results
- Test Evaluation
  - Evaluate results of testing, report to developers



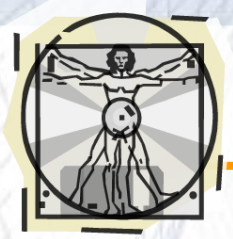


Types of Test Activities

- Background Knowledge Needed

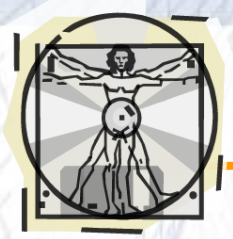
	Math	Programming	SW Engineering	Domain
Test Design – Criteria-based				
Test Design – Human-based				
Test Automation	-		-	
Test Execution	-	-	-	
Test Evaluation	-	-	-	

High Medium Low



Test Case Design

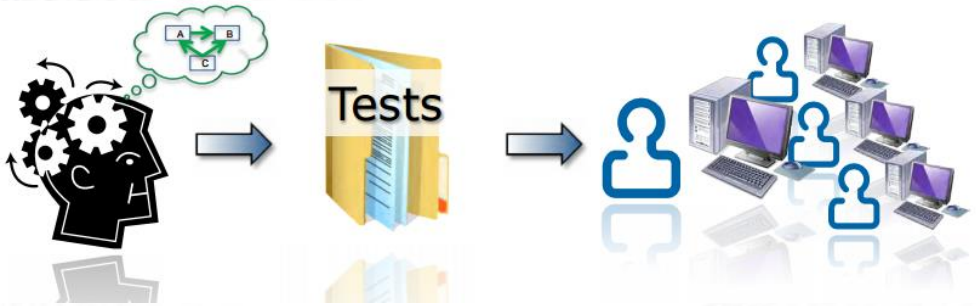
- Human-based
  - Ad-hoc
  - Exploratory
  - Scripted
- Criteria-based (later → BB and WB, etc.)



Testing Approaches

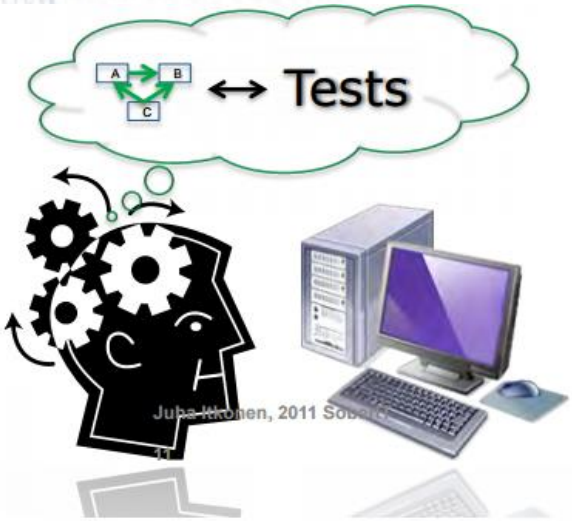
- **Exploratory testing**
  - In exploratory testing, **tests are designed and executed at the same time**
  - Unscripted doesn't mean unprepared
  - It's about enabling choice not constraining it
- **Scripted testing**
  - In scripted testing, tests are first designed and recorded Then they may be executed at some later time or by same or a different tester
  - With a script, you miss the same things every time!

Scripted vs. Exploratory



Scripted Testing

Most testing techniques (i.e. boundary value analysis, decision tables, etc.) can be used in both scripted and exploratory ways

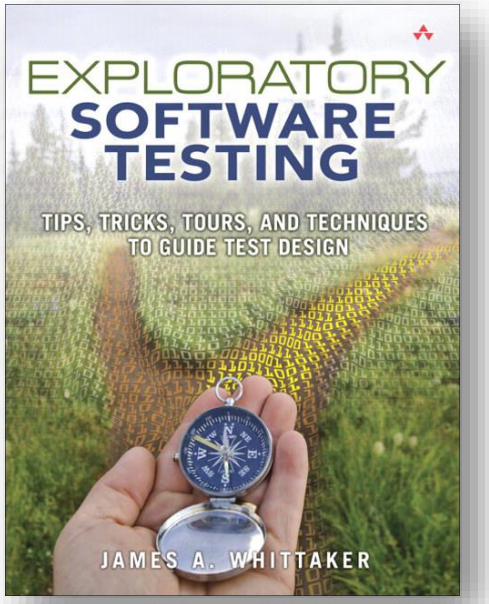


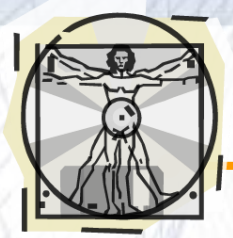
Exploratory Testing

Exploratory Approach

- Human tester
  - Using brain, and fingers
  - To create "realistic" scenarios that
    will cause software either to fail or to fulfill its mission
  - No strict plan (formal instructions)

In Exploratory Testing tester should increases their knowledge by exploring the application/software. Testers design and execute the next test based on the result of current step.





Exploratory Approach

- Human-based testing is all about varying things

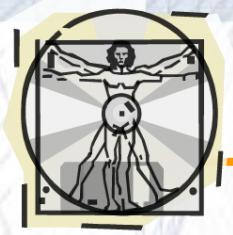
Input:
- Input combinations
- Order of inputs
- Legal versus illegal inputs
- Normal versus special
- Default / user supplied

State:
- History of stimuli paths
- Routes through environment
- Simulate the real world use of the SUT

Test Case Design

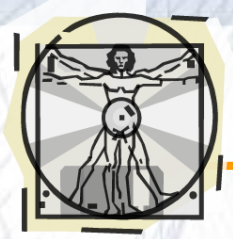
- Equivalent classes
- Boundary conditions
- Visible state transition

More about these comes later



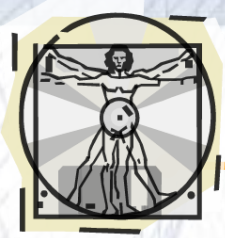
Exploratory Testing Level

- In the small
  - Focus on a simple functionality and excluding everything else
  - Easy to localize faults, if any
- In the large
  - Focus on a set or sequence of functionalities
  - Difficult to localize faults



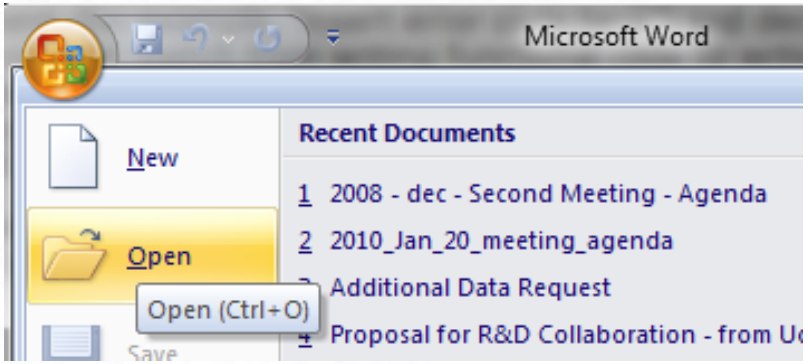
Exploratory Testing in the Small

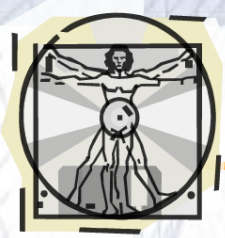
- Testing a small feature by choosing
  - which inputs to apply
  - what pages or screens to visit
  - which menu items to select
  - and the exact values to type into each input field they see.
- There are literally hundreds of such decisions to make with every test case we run
  - Exploratory testing can help a tester make these decisions, by varying things inputs, states, environment, user data, etc.



Exploratory Testing in the Small

- Example: in MS Word…
- Legal versus illegal input
  - User Input
- State
  - A state of software is a coordinate in the state space that contains exactly one value for every internal data structure
- User Data
  - Database contents, files, etc.
- Environment
  - E.g., OS and its configurations

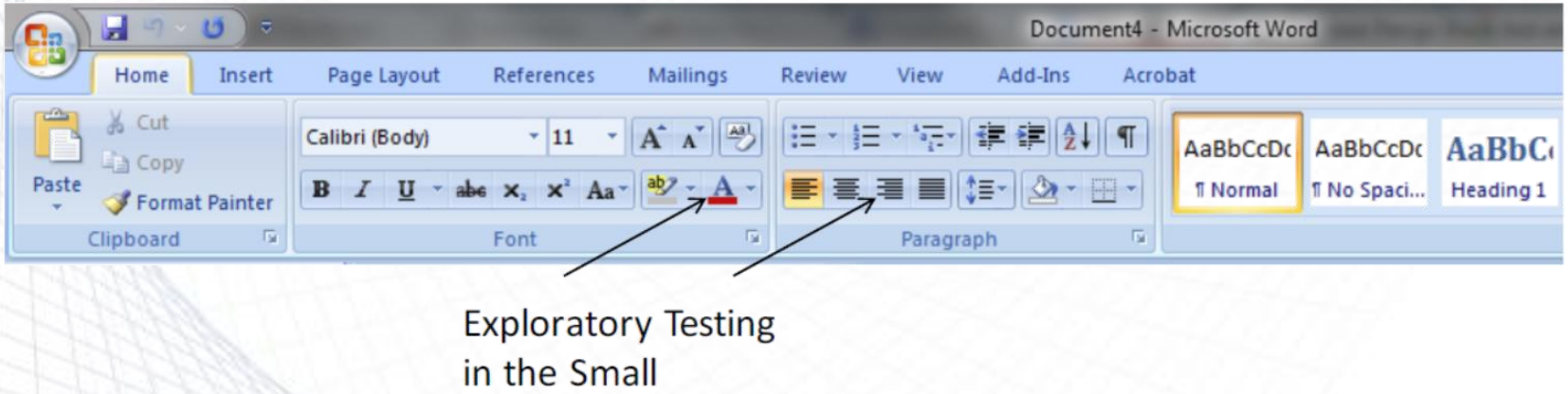


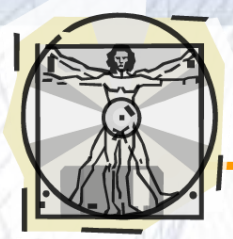


Exploratory Testing in the Small

- Example: in MS Word…

Each unit/feature is considered in isolation



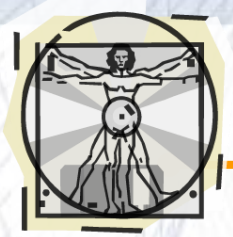


Exploratory Testing in the Large

- Testing units/features together
- Concerning feature **interaction**, **data flows**, and choosing **paths**
- Have an strategy in advance and let the goal guide you during testing sessions
- Exploratory testing in the Large: sequence of several small test cases

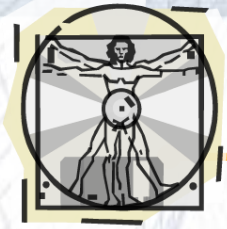
e.g.,
open file → change text → change font → save → close →
open it again → check if the changes were correctly saved

Sequence of several small functionalities: operation

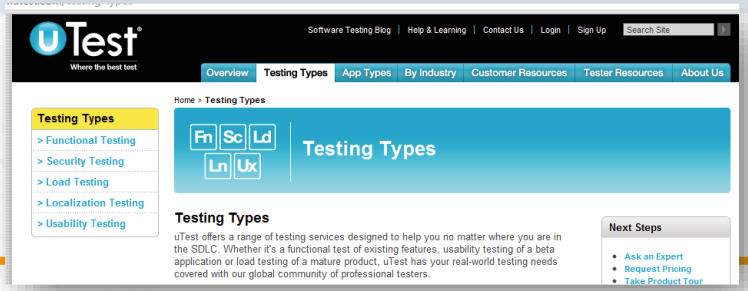


Hybrid Exploratory Testing

- Scenarios and exploration
  - Use scenarios like maps
- Scenarios
  - User story, requirement, feature, an integration, setup, things that go wrong, etc.
- Create variation in scenarios
  - Add steps: different paths
  - Modify steps: Change data, environment, etc.
  - Remove and repeat steps



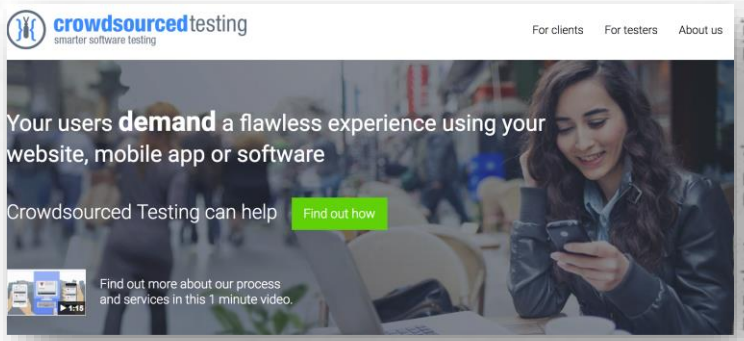
Crowd Testing



- A natural match for exploratory testing

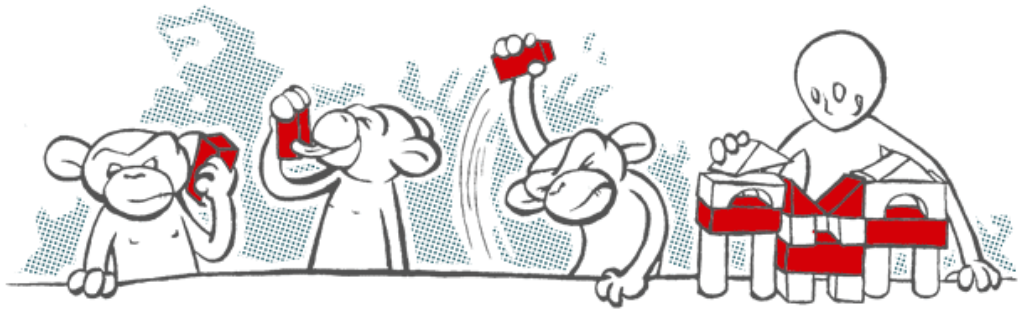
How It Works?

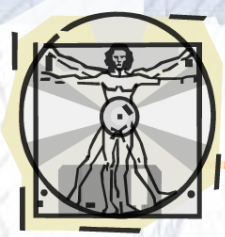
- Tell us your testing requirements
- Optionally create or import test cases
- We'll select testers who best match your requirements
- You'll receive a list of bugs and feedback from real world testing
- You can ask testers for additional details & information
- You can also export your bugs to popular bug tracking systems



Manual Scripted Testing

- Scripted testing follows a path that is written by the tester themselves (or someone else)
- The script includes test cases and test steps that are documented
- No deviation from the path laid out in the script

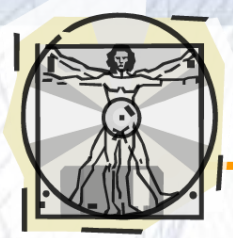




Manual Regression Testing

- Regression testing is testing done to check that a **system update** does not **reintroduce faults** that **have been corrected earlier**

- Usually performed after a bug fix code is checked in to the code repository



Human-based Testing Limitation

- Scalability, and bias ← **focus on a subset of features**
- Does exploratory testing scale up to very complex systems? **Usually no!**
- Can a human tester keep track of what he has tested and what not? **Usually no!**
- Can we do a complete testing job on MS Word only by exploratory approach?! **Perhaps no!**
- Is exploratory testing repeatable? **Usually no!**

Additional Videos

- Lecture Video (16 mins)
- How to Become a Software Tester? (5 min)
- Software testing tutorial (one hour)
- etc.

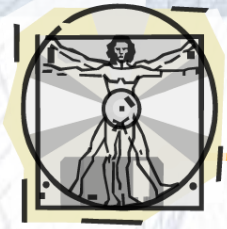
SCHULICH School of Engineering

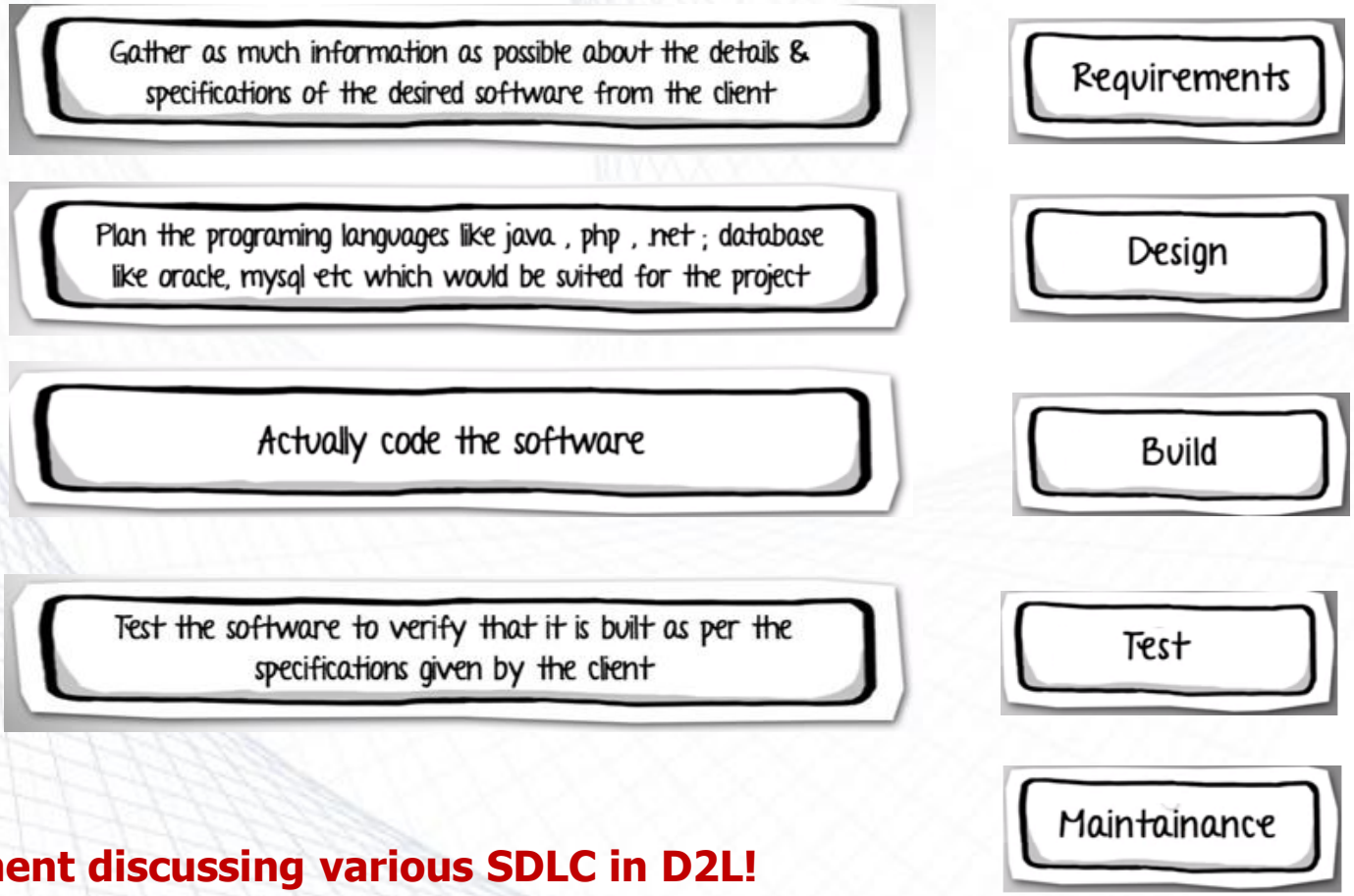
UNIVERSITY OF CALGARY

SENG 637
Dependability and Reliability of Software Systems

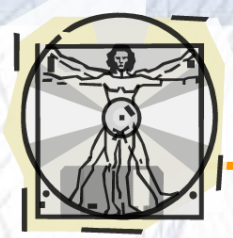
Software Development Life Cycle (SDLC)

Software Development Life Cycle (SDLC)



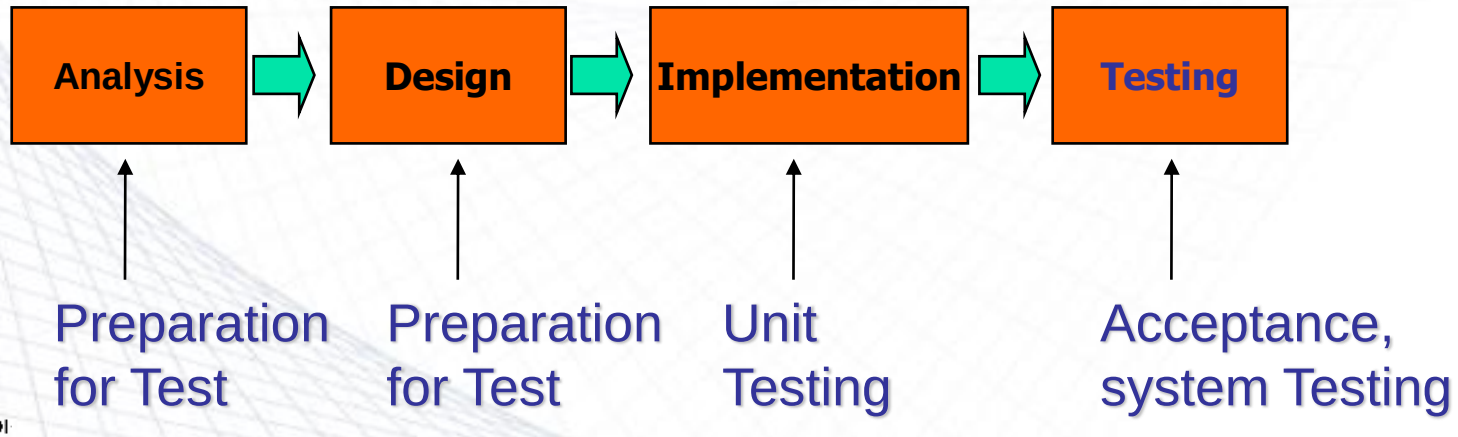


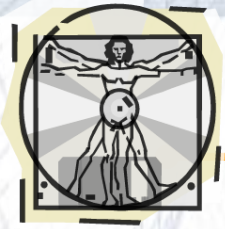
Document discussing various SDLC in D2L!



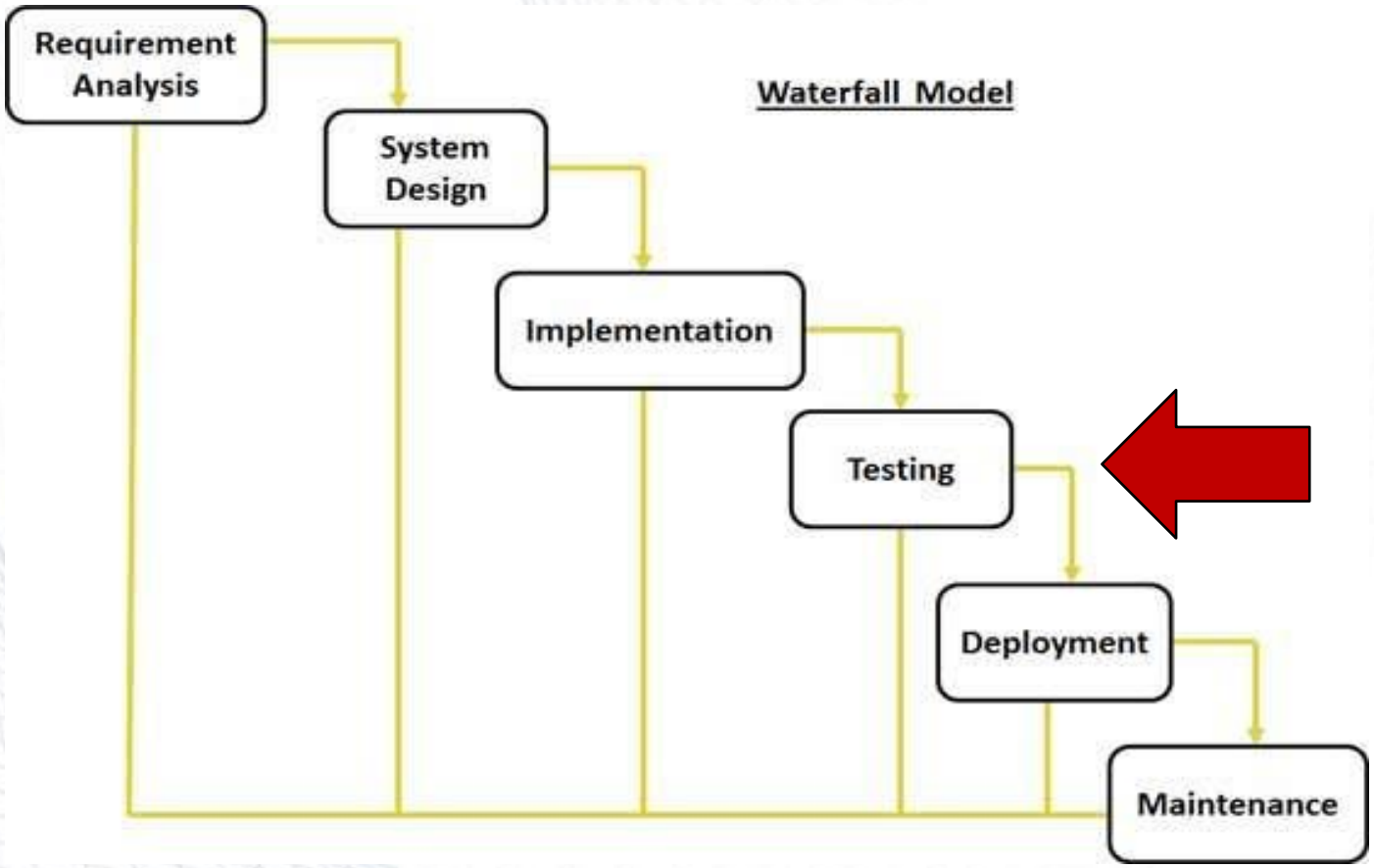
Testing throughout the SDLC – in Waterfall

- SDLC: Software Development Life Cycle
- Life-cycle development artifacts provides a rich source of test data
- Identifying test requirements and test cases early helps shorten the development time
- They may help reveal faults
- It may also help identify early low testable specifications or design





Testing in Waterfall

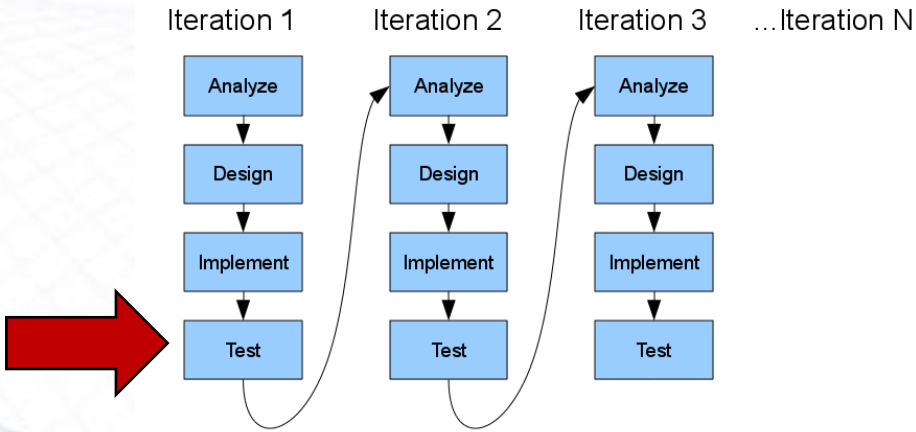


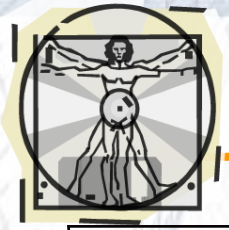
Waterfall Model Pros & Cons

Pros	Cons
• Simple and easy to understand and use • Easy to manage due to the rigidity of the model because ach phase has specific deliverables and a review process • Phases are processed and completed one at a time • Works well for smaller projects where requirements are very well understood • Clearly defined stages • Easy to arrange tasks • Process and results are well documented	• No working software is produced until late during the life cycle • High amounts of risk and uncertainty • Not a good model for complex and object-oriented projects • Poor model for long and ongoing projects • It does not allow for much reflection or revision, i.e. once an application is in the testing stage, it is very difficult to go back and change something that was not well-documented

Testing in Iterative Model

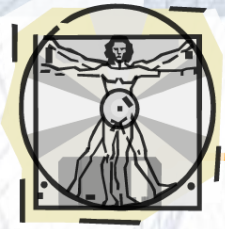
- Iterative process starts with a simple implementation of a subset of the software requirements
- At each iteration, design modifications are made and new functional capabilities are added
- The basic idea behind this method is to develop a system through repeated cycles (iterative) and in smaller portions at a time (incremental)



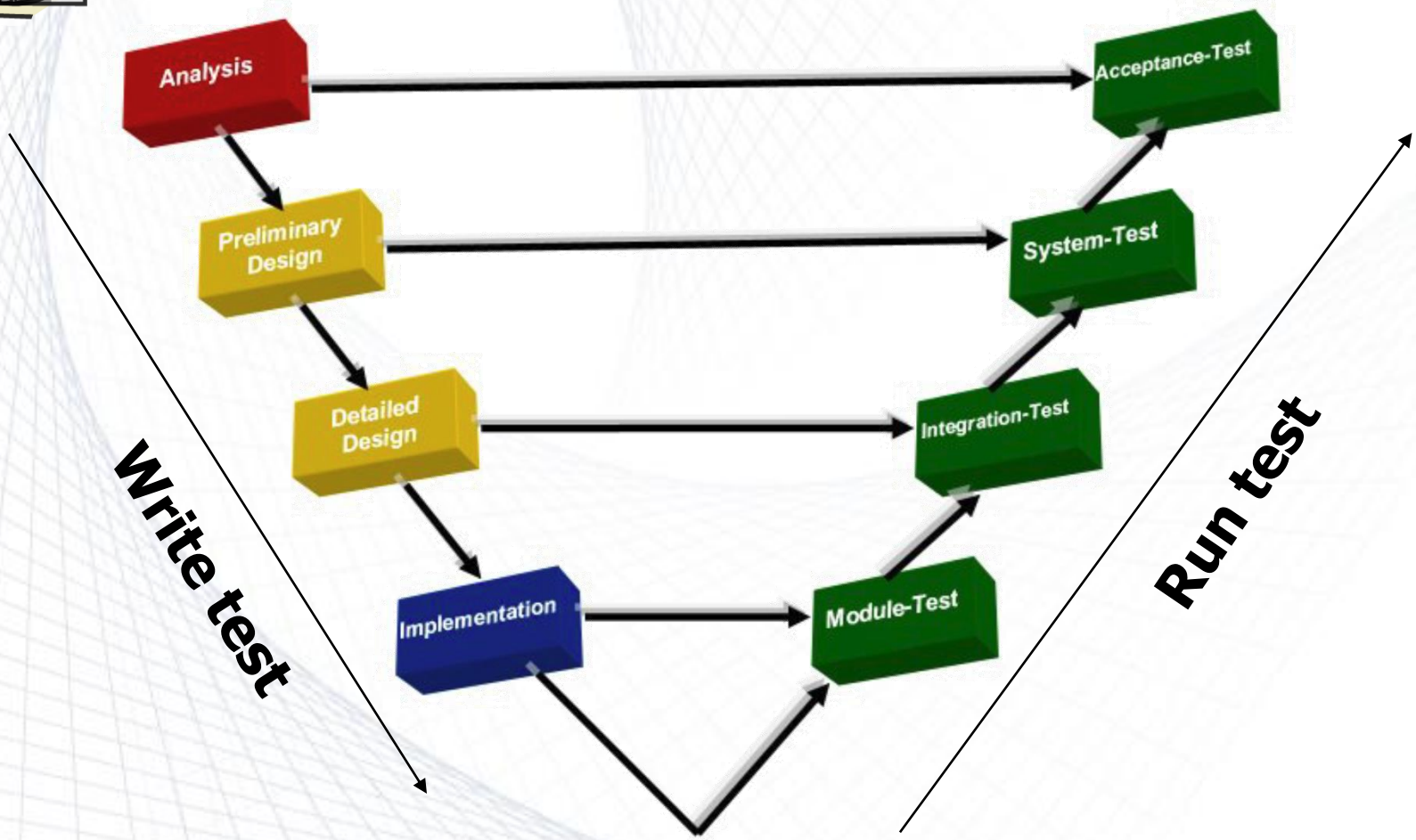


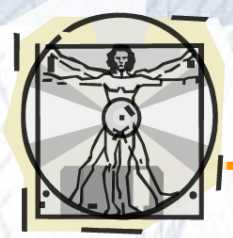
Iterative Model Pros & Cons

Pros	Cons
• Some working functionality can be developed quickly and early in the life cycle • Results are obtained early and periodically • Parallel development can be planned • Less costly to change the scope/requirements • Testing and debugging during smaller iteration is easy • Risks are identified and resolved during iteration and each iteration is an easily managed milestone • Easier to manage risk - High risk part is done first • With every increment operational product is delivered • Issues, challenges and risks identified from each increment can be utilized/applied to the next increment • During life cycle software is produced early which facilitates customer evaluation and feedback	• More resources may be required • More management attention is required • System architecture or design issues may arise because not all requirements are gathered in the beginning of the entire life cycle • Management complexity is more • End of project may not be known which is a risk • Highly skilled resources are required for risk analysis



Testing in the "V" Model





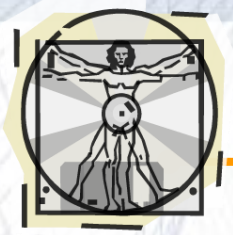
V-Model Testing Stages

- Unit testing
  - Testing of individual components.
- Integration testing
  - Testing to expose problems arising from the combination of components.
- System testing
  - Testing the complete system prior to delivery
- Acceptance testing
  - Testing by users to check that the system satisfies requirements. Sometimes called alpha testing

https://www.youtube.com/watch?v=N8-qNMHOVyw&index=5&list=PLDC2A0C8D2EC934C7

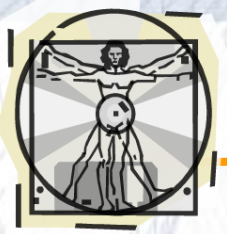
V- Model Pros and Cons

Pros	Cons
• This is a highly disciplined model and Phases are completed one at a time • Works well for smaller projects where requirements are very well understood • Simple and easy to understand and use • Easy to manage due to the rigidity of the model, that is, each phase has specific deliverables and a review process	• High risk and uncertainty • Not a good model for complex and object-oriented projects • Poor model for long and ongoing projects • Not suitable for the projects where requirements are at a moderate to high risk of changing • Once an application is in the testing stage, it is difficult to go back and change a functionality • No working software is produced until late during the life cycle

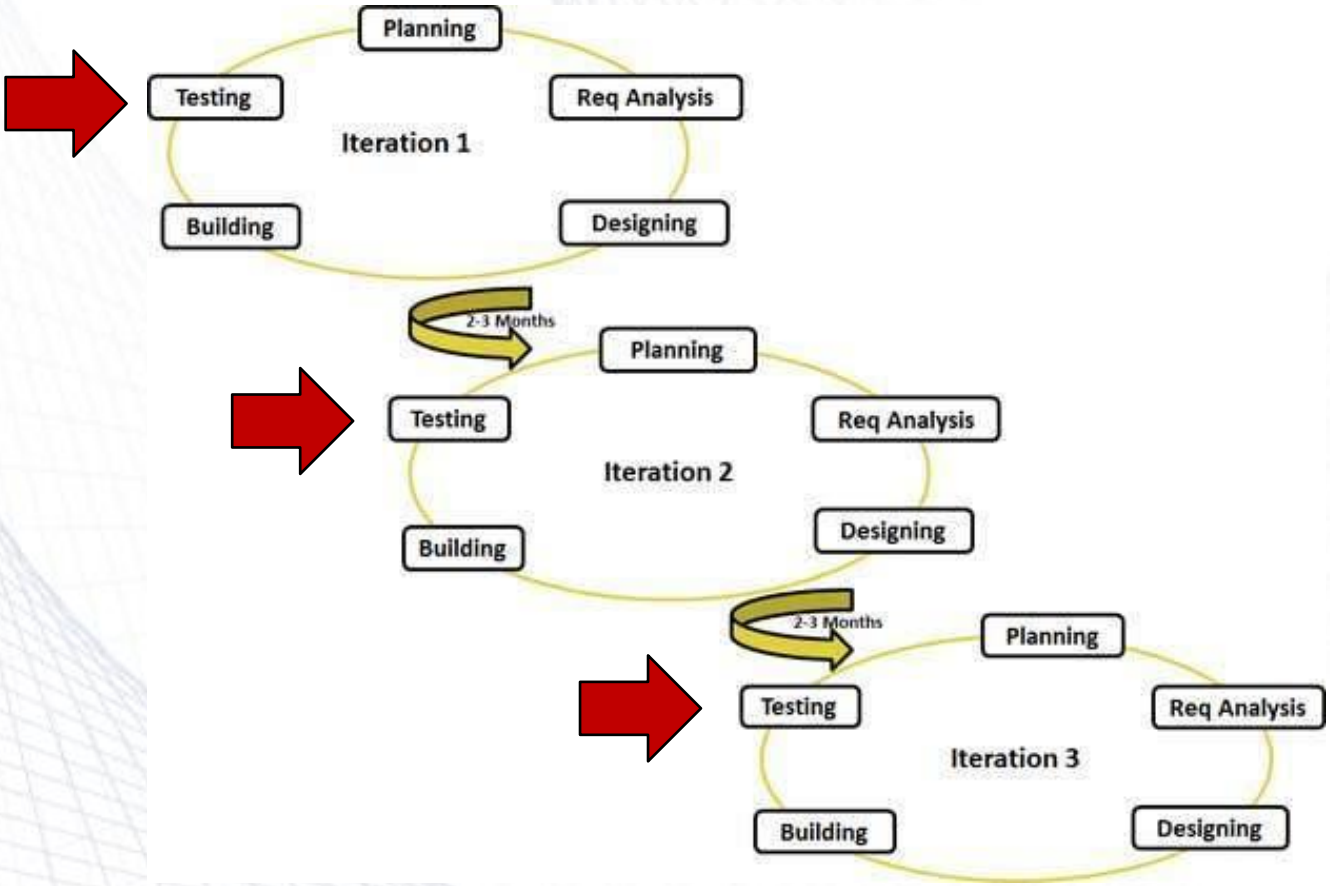


Testing in Agile Model

- Agile methods are being widely accepted in the software world recently
- In agile the tasks are divided to time boxes (small time frames) to deliver specific features for a release
- Working software build is delivered after each iteration

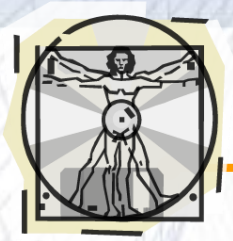


Testing in the Agile



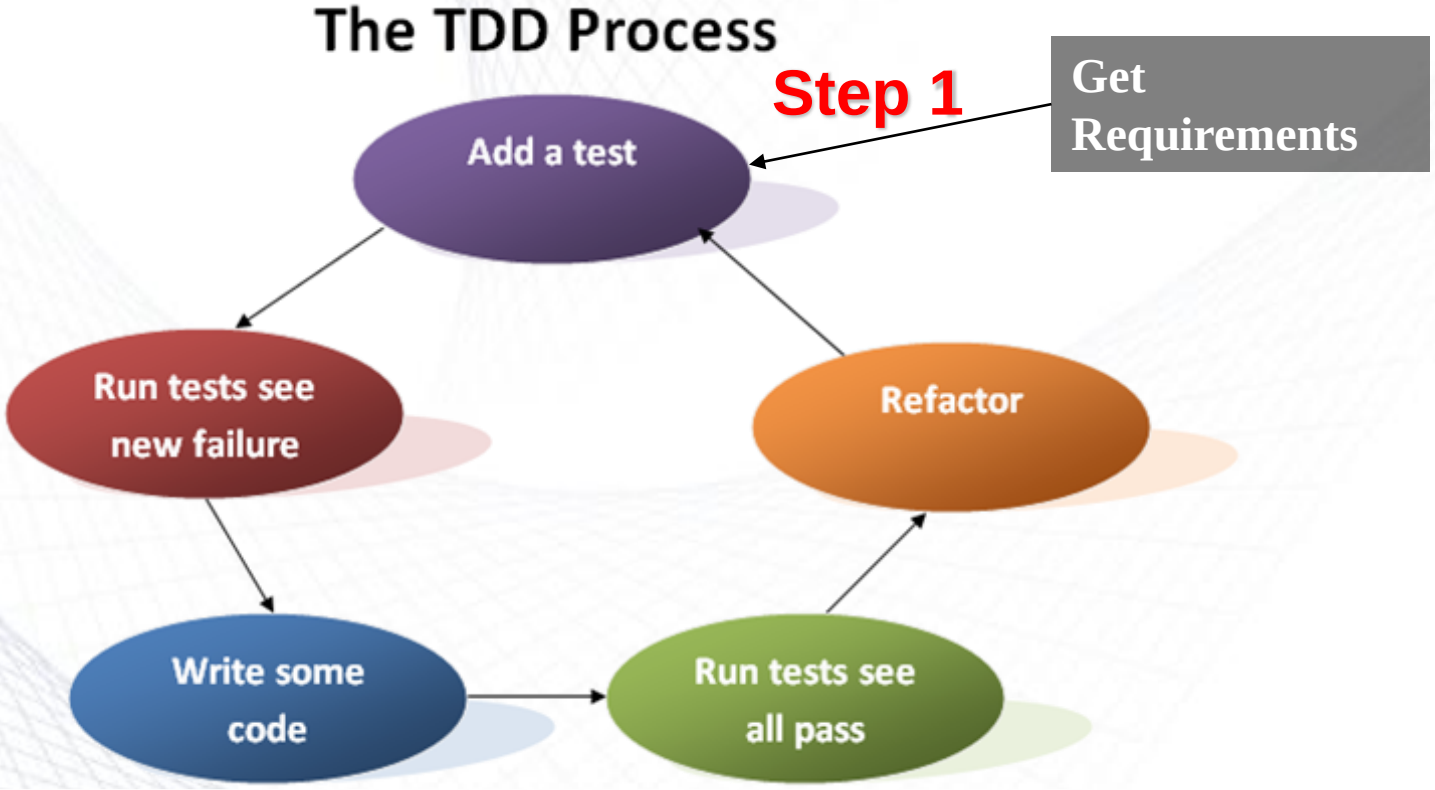
Agile Model Pros & Cons

Pros	Cons
• Is a realistic approach to software development • Functionality can be developed rapidly and demonstrated • Resource requirements are minimum • Suitable for fixed or changing requirements • Delivers early partial working solutions • Good model for environments that change steadily • Minimal rules, documentation easily employed • Little or no planning required • Gives flexibility to developers	• An overall plan, an agile leader and agile project manager practice is a must without which it will not work • Strict delivery management dictates the scope, functionality to be delivered, and adjustments to meet the deadlines • There is very high individual dependency since there is minimum documentation generated • Transfer of technology to new team members may be quite challenging due to lack of documentation



Testing in TDD

(TDD = Test-Driven Development)

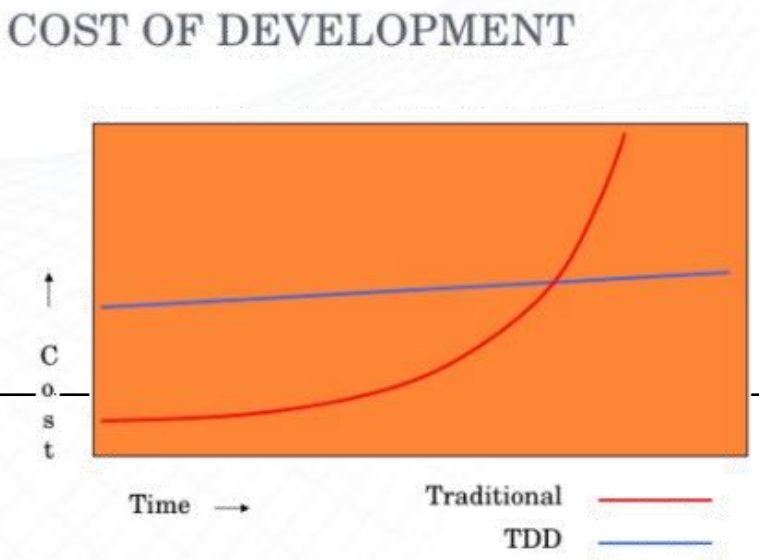


In TDD, test cases are actually the requirements of the system

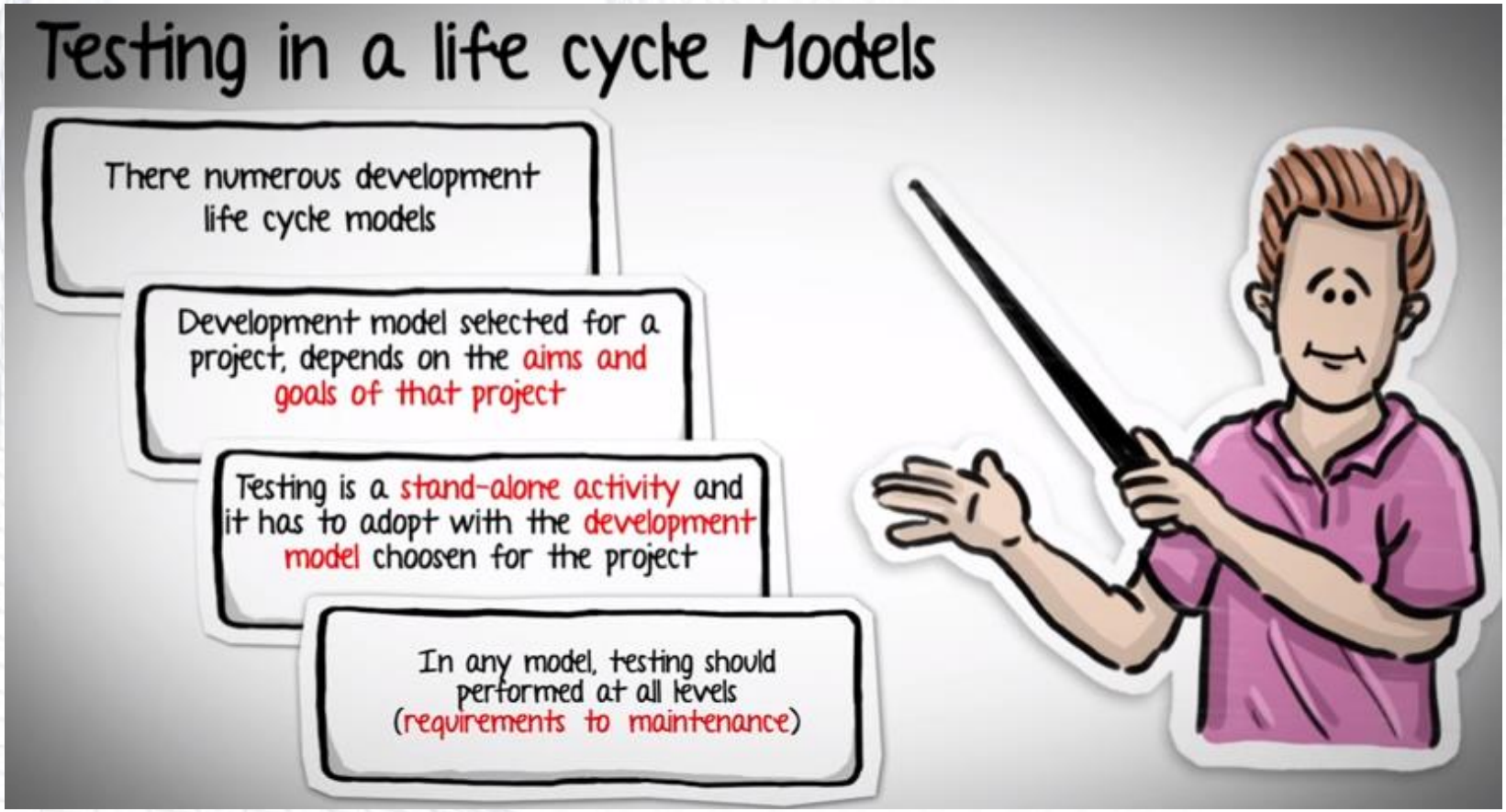
Document introducing TDD in D2L!

TDD Model Pros & Cons

Pros	Cons
• TDD shortens the programming feedback loop • Guarantees the development of high quality software • Provides concrete evidence that software is working • Less debug time • Reduced software failure rate	• Programmers like to code, not to test • Test writing is time consuming



SDLC - Conclusion



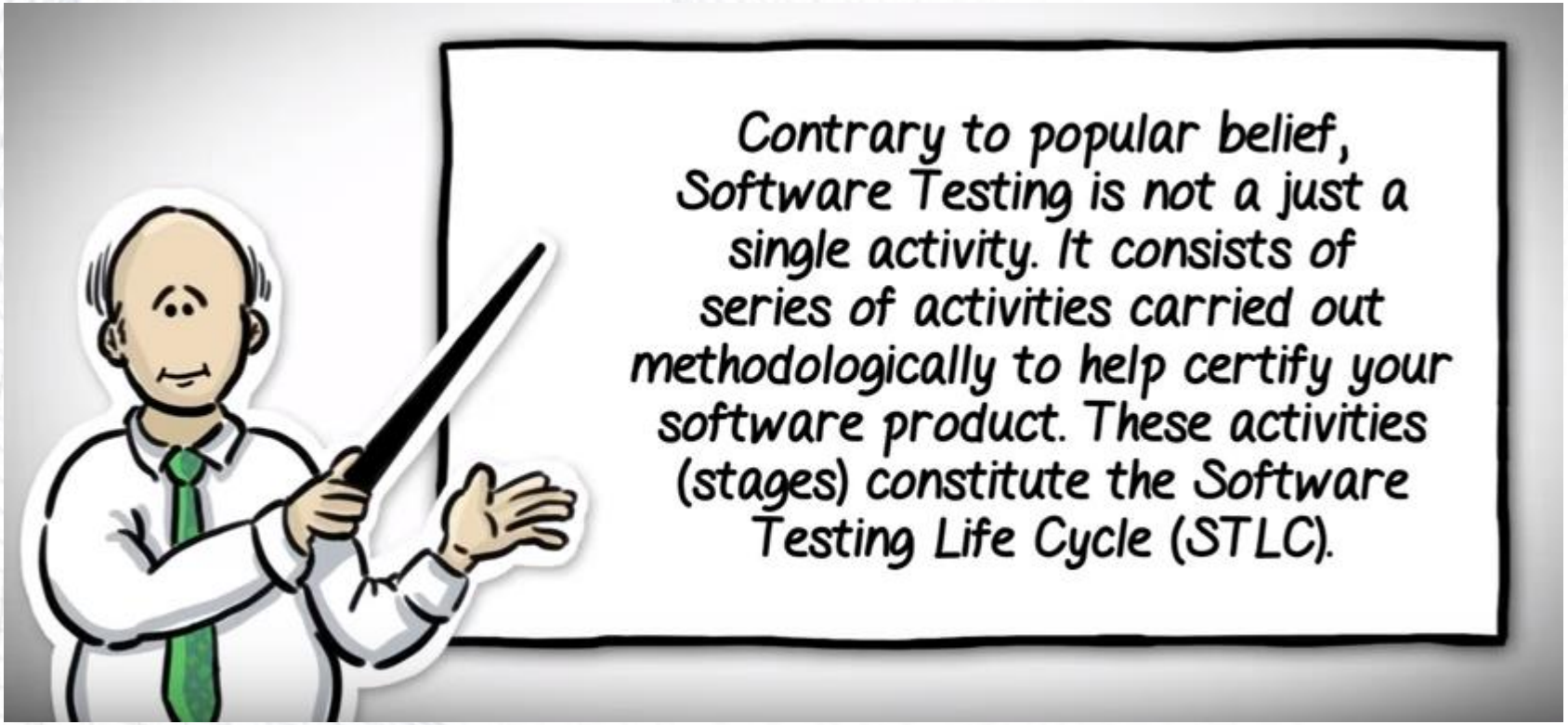
SCHULICH School of Engineering

UNIVERSITY OF CALGARY

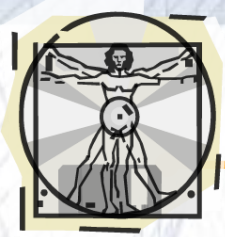
SENG 637
Dependability and Reliability of Software Systems

Software Testing Life Cycle (STLC)

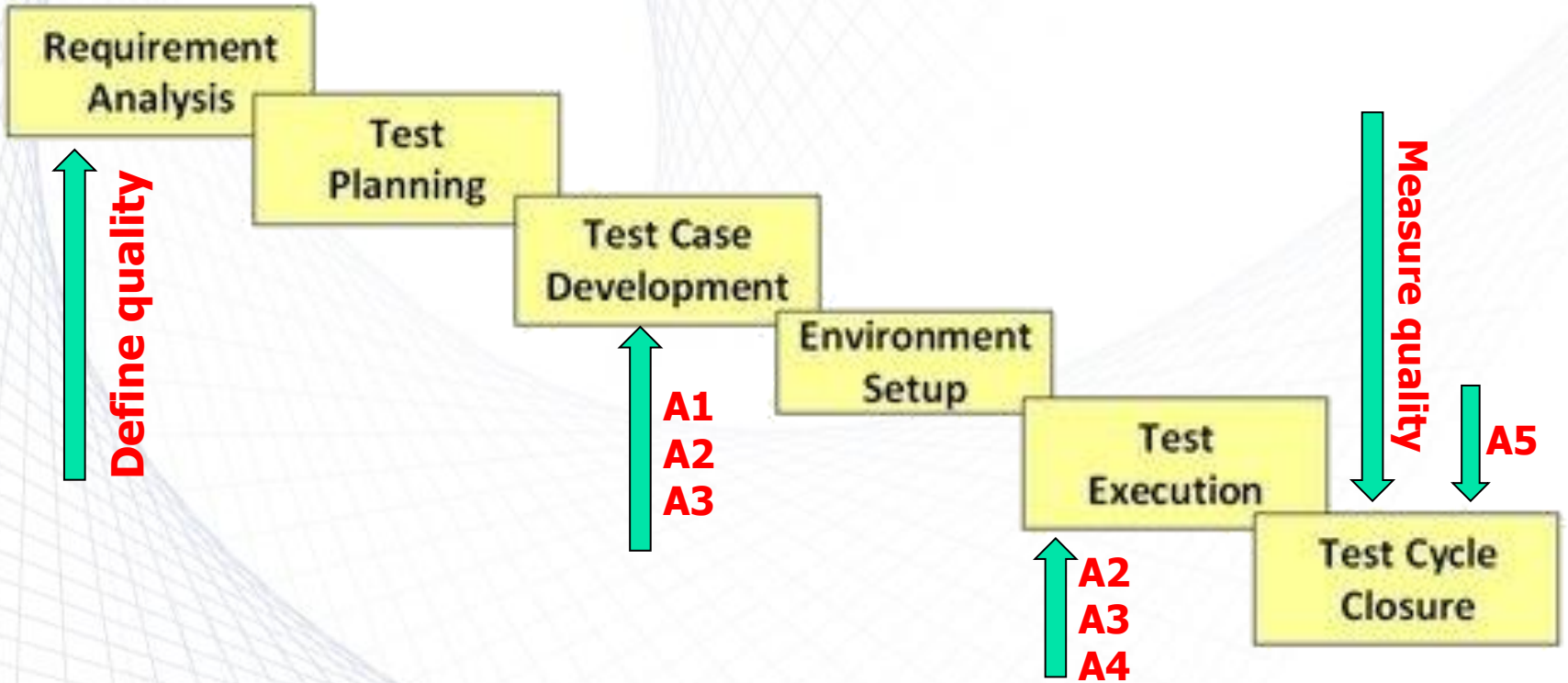
Software Testing Life Cycle (STLC)



Software Testing Life Cycle (STLC)



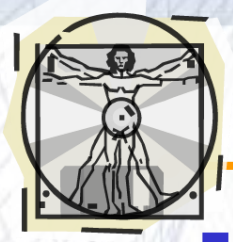
A1-A5: Course Assignments



https://www.youtube.com/watch?v=Dq5IYYqnnGQ&index=24&list=PLDC2A0C8D2EC934C7

Requirement Analysis

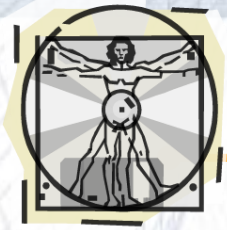
- Test team studies the requirements from a testing point of view to identify the ***testable requirements***
- The QA team may interact with various stakeholders (client, business analyst, technical leads, system architects, etc.) to understand the requirements in detail
- Requirements could be either **Functional** (defining what the software must do) or **Non Functional** (defining system performance, security, availability, etc.)
- Automation feasibility for the given testing project is also done in this stage



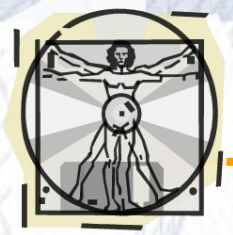
Requirement Analysis

- Activities
  - Identify types of tests to be performed
  - Gather details about testing priorities and focus
  - Prepare *Requirement Traceability Matrix (RTM)*
  - Identify test environment details where testing is supposed to be carried out
  - Automation feasibility analysis (if required)
- Deliverables
  - RTM
  - Automation feasibility report (if applicable)

Requirement Traceability Matrix (RTM)



Requirements		Req 1	Req 2	Req 3	Req 4	Req 5	Req 6	Req 7	Req 8	Req 9	Req 10
Test Cases	Totals	1	2	4	2	2	5	5	5	2	4
TC1	3	X		X							X
TC2	1		X								
TC3	1		X								
TC4	2				X	X					
TC5	2				X	X					
TC6	3						X	X	X		
TC7	3						X	X	X		
TC8	3						X	X	X		
TC9	3						X	X	X		
TC10	3						X	X	X		
TC11	1									X	
TC12	1									X	
TC13	2			X							X
TC14	2			X							X
TC15	2			X							X



Requirement Traceability Matrix (RTM)

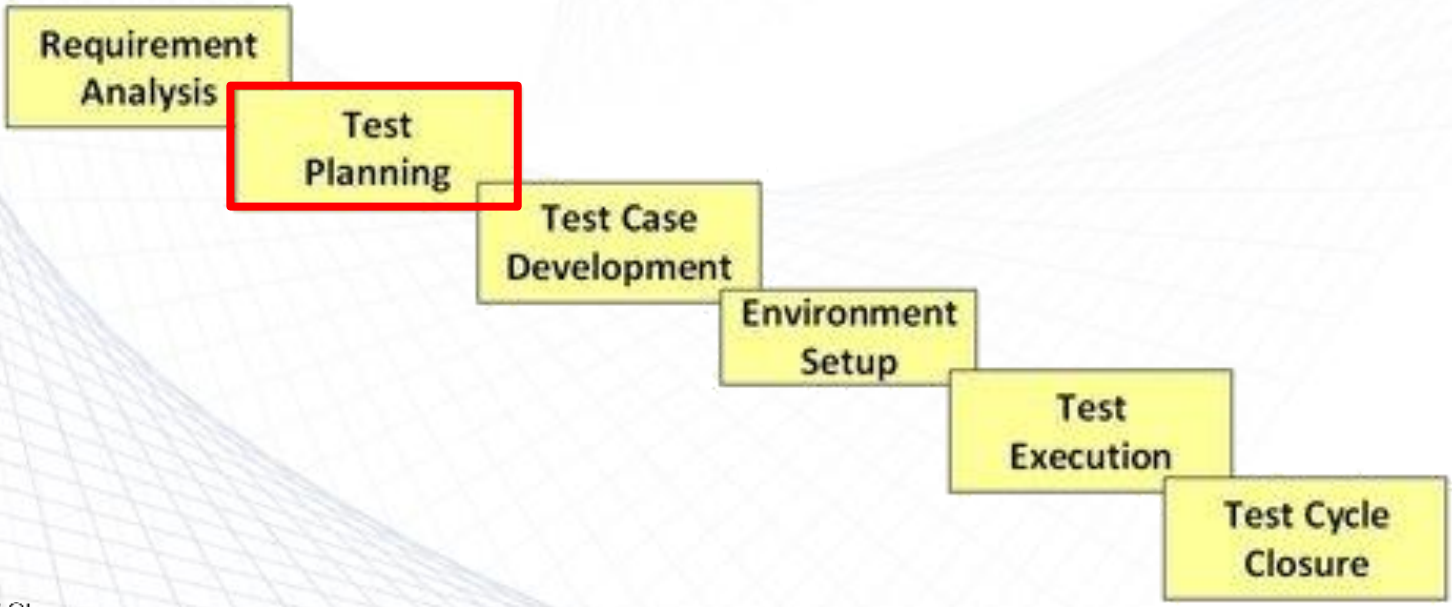
- Discussion
  - How RTM can help test engineers?

    Are all requirements tested?
  - How RTM can help project managers?

    Are we on schedule with development and testing?

Test Planning

- This phase is also called Test Strategy phase
- Typically, a senior QA manager will determine **effort** and **cost** estimates for the project and would prepare and finalize the **Test Plan**



Test Planning

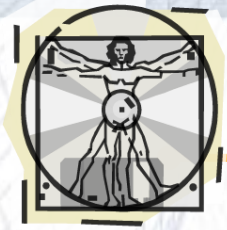
- Activities
  - Preparation of test plan/strategy document for various types of testing
  - Test tool selection
  - Test cost/Cost/effort estimation
  - Resource planning and determining roles and responsibilities.
  - Training requirement
- Deliverables
  - Test plan /strategy document
  - Effort estimation document

- There are templates for documents such as test plan, etc. For example: IEEE 829 Test Plan

What to write in a Test Plan Document

- Introduction: Overview, goals/objectives, Any constraints.
- References: List the related documents, Project Plan, Configuration Management Plan
- Test Items: List the test items (software/products) and their versions.
- Features to be Tested
- Features Not to Be Tested
- Approach: [Manual/Automated; White Box/Black Box/Gray Box]
- Pass/Fail Criteria
- Suspension Criteria and Resumption Requirements

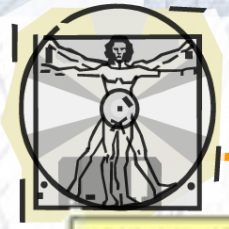
What to write in a Test Plan Document

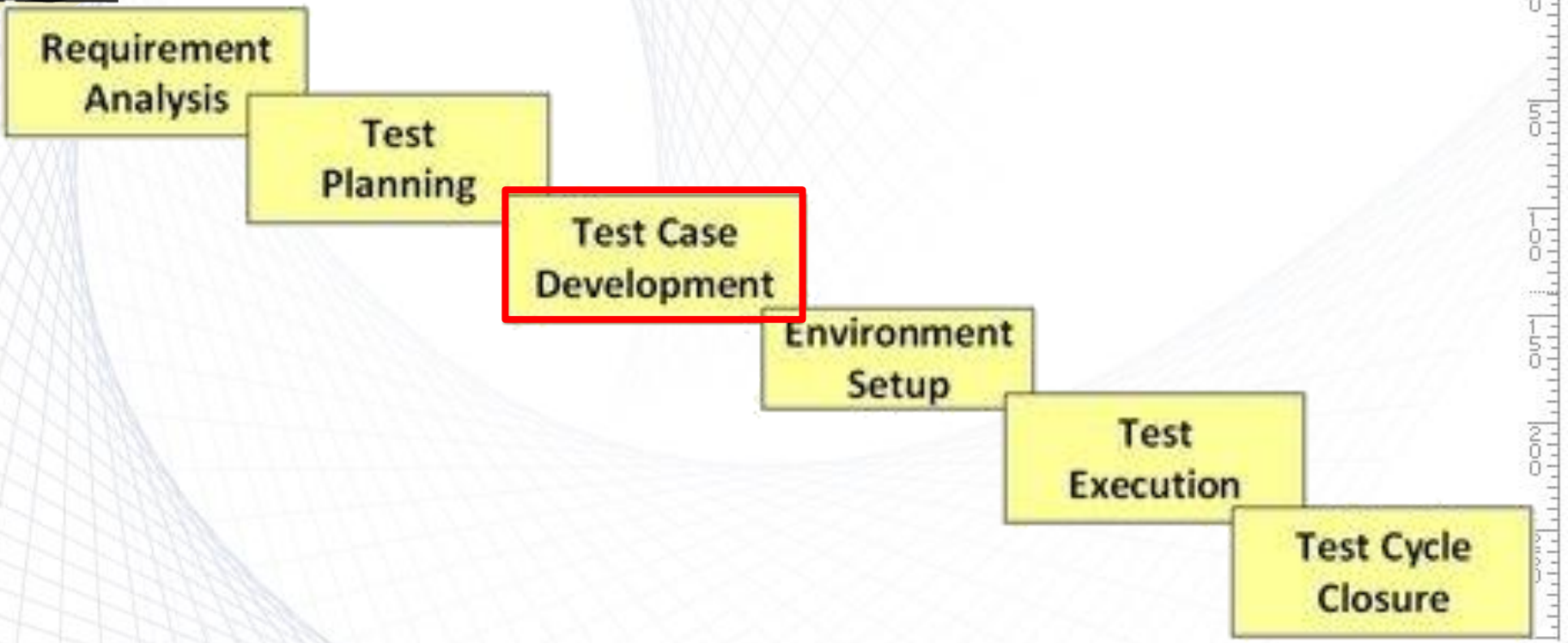


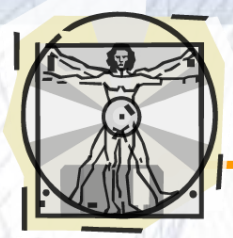
- Test Deliverables: Test Cases, Test Scripts, Test Logs, Test Reports
- Test Environment: hardware, software, network
- Estimate: test estimates (cost or effort) and/or provide a link to the detailed estimation
- Schedule
- Staffing and Training Needs
- Responsibilities
- Risks
- Assumptions and Dependencies
- Approvals

Sample test plan in D2L!

Software Testing Life Cycle (STLC)





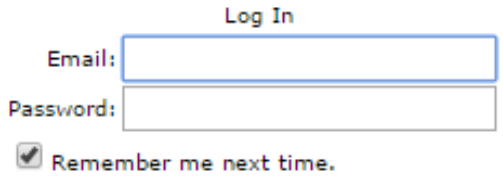


Test Case Development

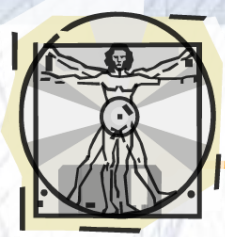
- This phase involves creation, verification and rework of test cases and test scripts. Test data, is identified/created and is reviewed and then reworked.
- Activities
  - Create test cases, automation scripts (if applicable)
  - Review and baseline test cases and scripts
  - Create test data (If Test Environment is available)
- Deliverables
  - Test cases/scripts
  - Test data

Sample test case worksheet in D2L!

What to Write in a Test Case?



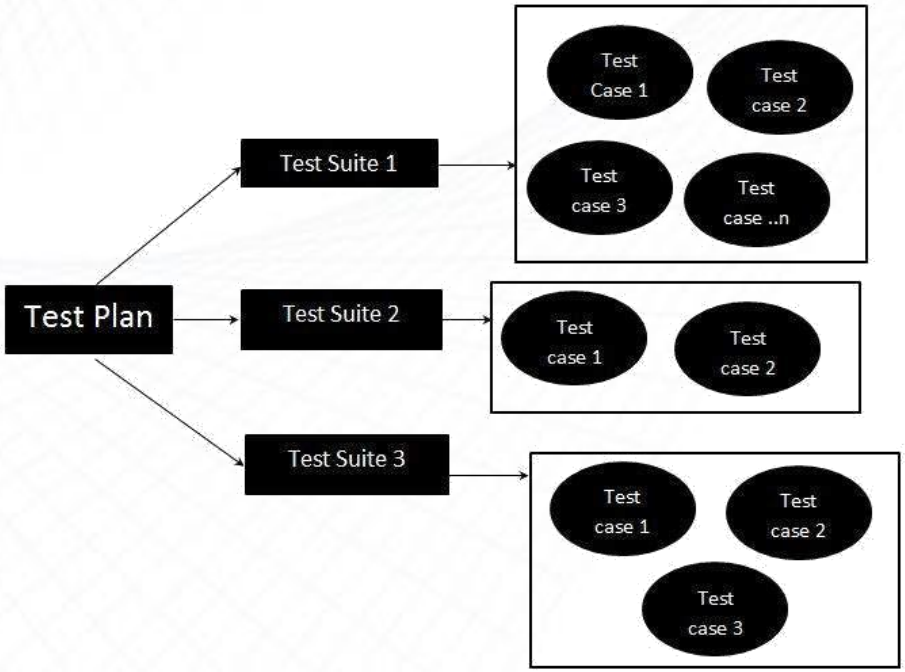
Test case design date: Test Designed By:				Test case Executed date: Test Executed by:			
Pre-Conditions							
TC #	**Scenario**	**Test case**	**Test Data**	**Test Steps**	**Expected Results**	**Actual Result**	**Pass/ Fail**
1	Check Login Functionality	Check response on entering valid login information	Username: SENG Password: 437	1. Launch application 2. Enter username 3. Enter password 4. Clock ok button	Login must be successful		
Post-Condition							



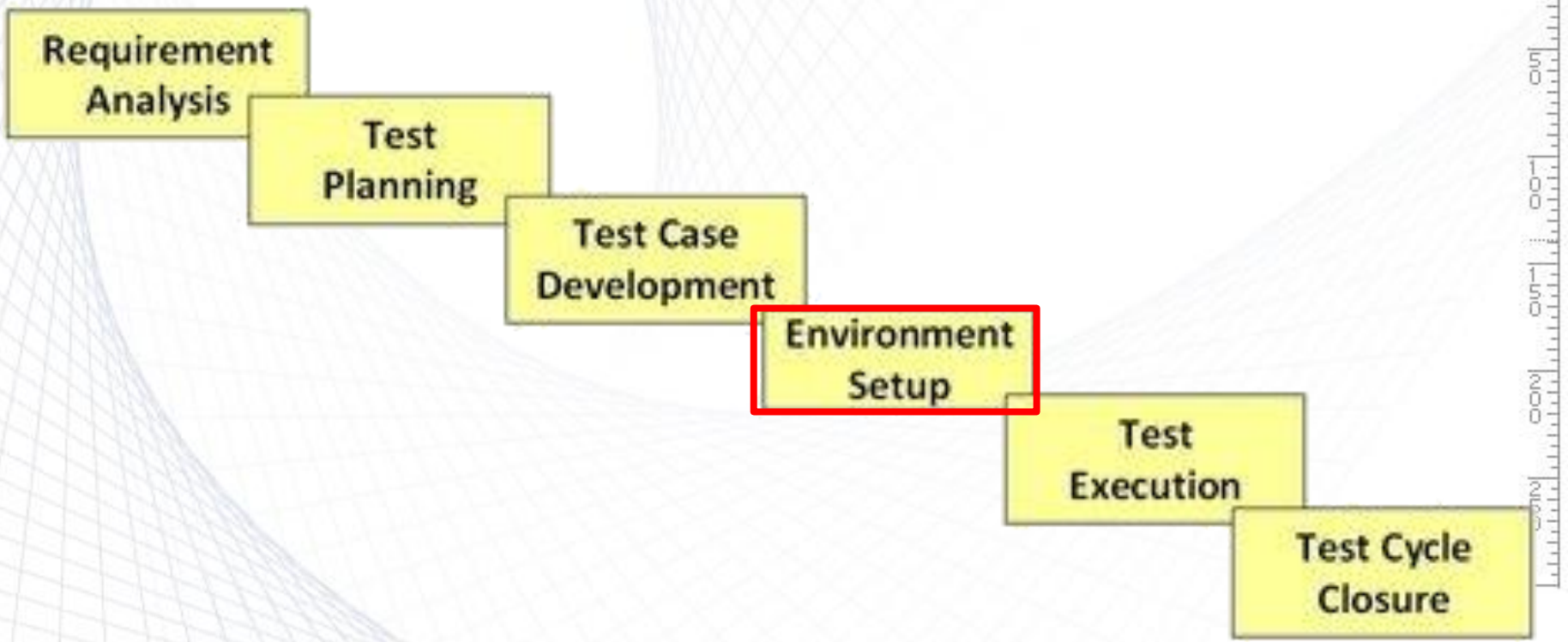
What is a Test Suite?

- Test suite is a set of test cases that needs to be run together with the system under test (SUT)

test fixture
unit test
test case
test suite
test runner

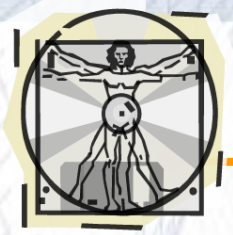


Software Testing Life Cycle (STLC)



Test Environment Setup

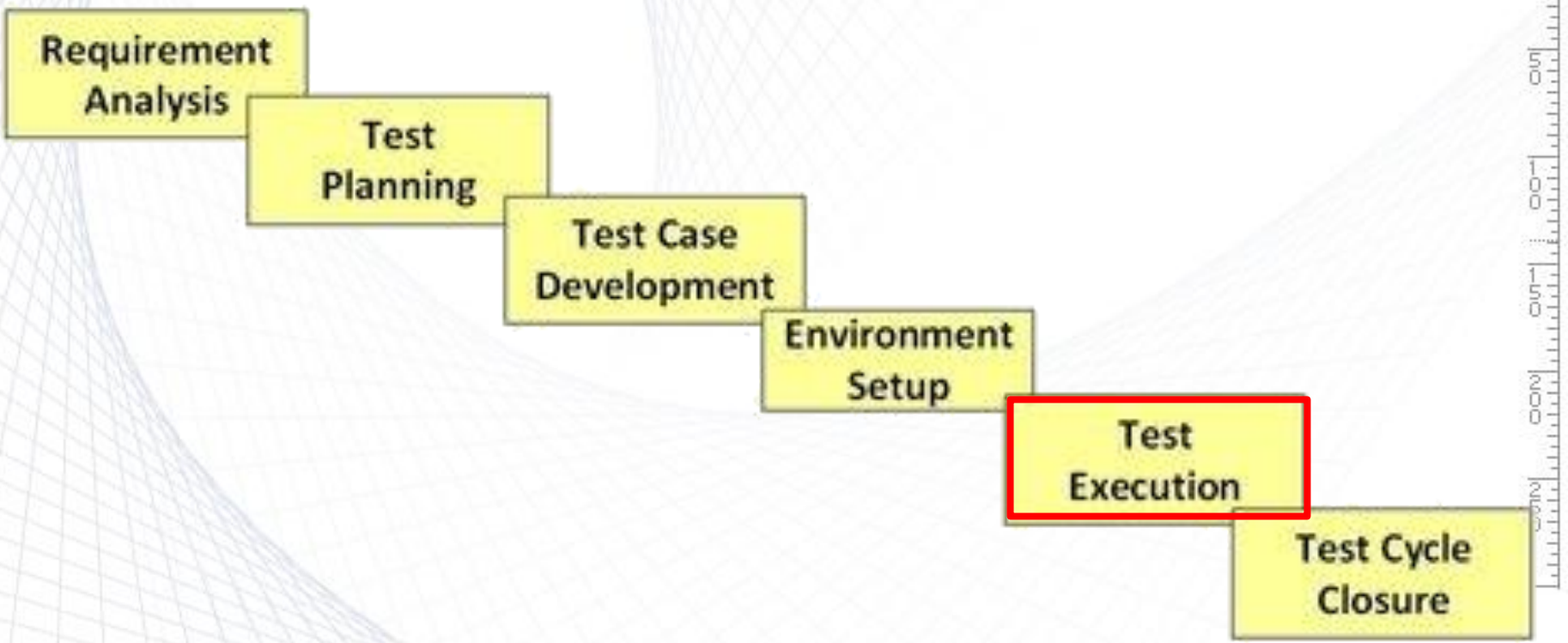
- Test environment decides the software and hardware conditions under which a work product is tested
- Test environment set-up is one of the critical aspects of testing process and *can be done in parallel with Test Case Development Stage*
- *Test team may not be involved in this activity* if the customer/development team provides the test environment in which case the test team is required to do a readiness check (smoke testing) of the given environment



Test Environment Setup

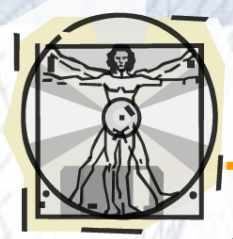
- Activities
  - Understand the required architecture, environment set-up and prepare hardware and software requirement list for the Test Environment
  - Setup test environment and test data
  - Perform smoke test on the build
- Deliverables
  - Environment ready with test data set up

Software Testing Life Cycle (STLC)



Test Execution

- During this phase test team will carry out the testing based on the test plans and the test cases prepared
- Bugs will be reported back to the development team for correction and retesting will be performed

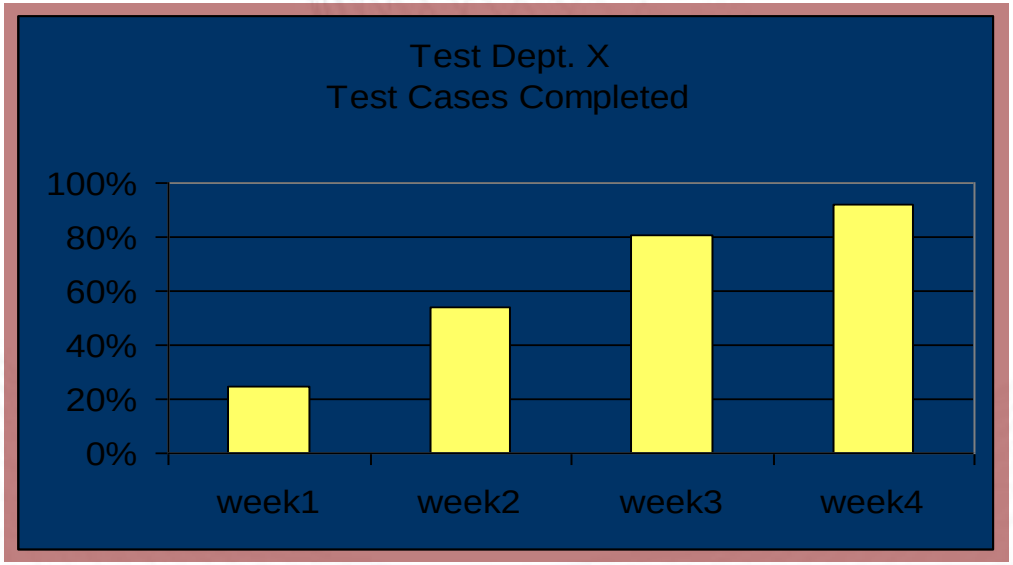


Test Execution

- Activities
  - Execute tests as per plan
  - Document test results, and log failures for failed cases
  - Map failures to test cases in RTM
  - Retest the failure fixes
  - Track the failures to closure
- Deliverables
  - Completed RTM with execution status
  - Test cases updated with results
  - **failure reports**

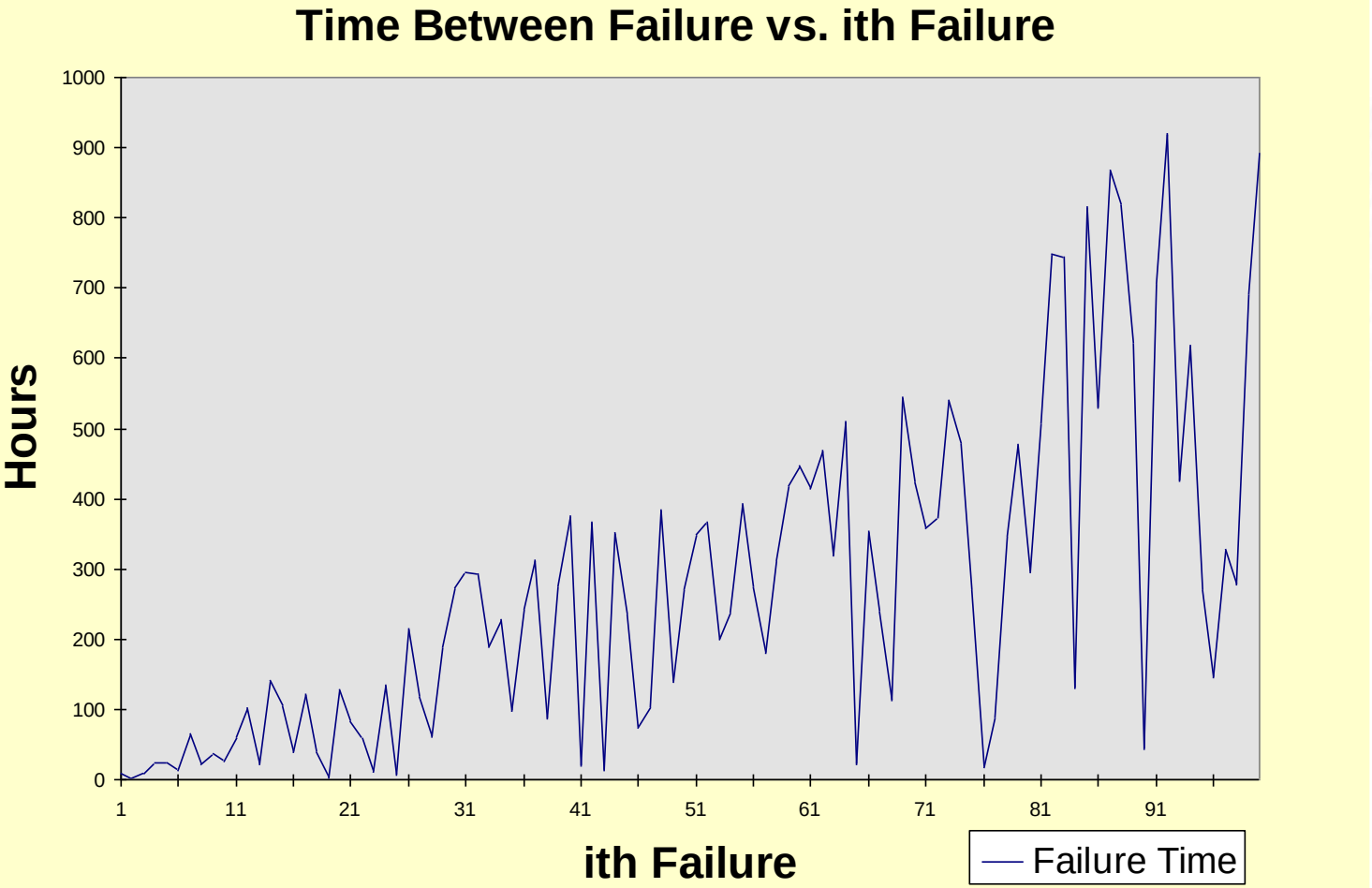
We will discuss this soon!

Reporting: Execution Results



	week1	week2	week3	week4
Completed %	25%	54%	81%	92%
Test Cases Passed	200	175	150	50
Test Cases Failed	50	60	70	25
Test Cases Planned	1000	900	875	850
Test Cases Completed	250	485	705	780
Pass Rate	80%	74%	68%	67%

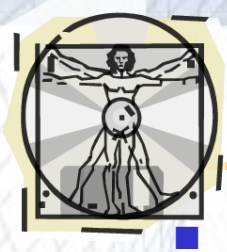
Reporting: TBF



Software Testing Life Cycle (STLC)

Test Cycle Closure

- Testing team will meet, discuss and analyze testing artifacts to identify strategies that have to be implemented in future, taking lessons from the current test cycle
- Main question to discuss: Did we succeed to reach the quality setup earlier? If not what to do?
- The idea is to remove the process bottlenecks for future test cycles and share best practices for any similar projects in future



Test Cycle Closure

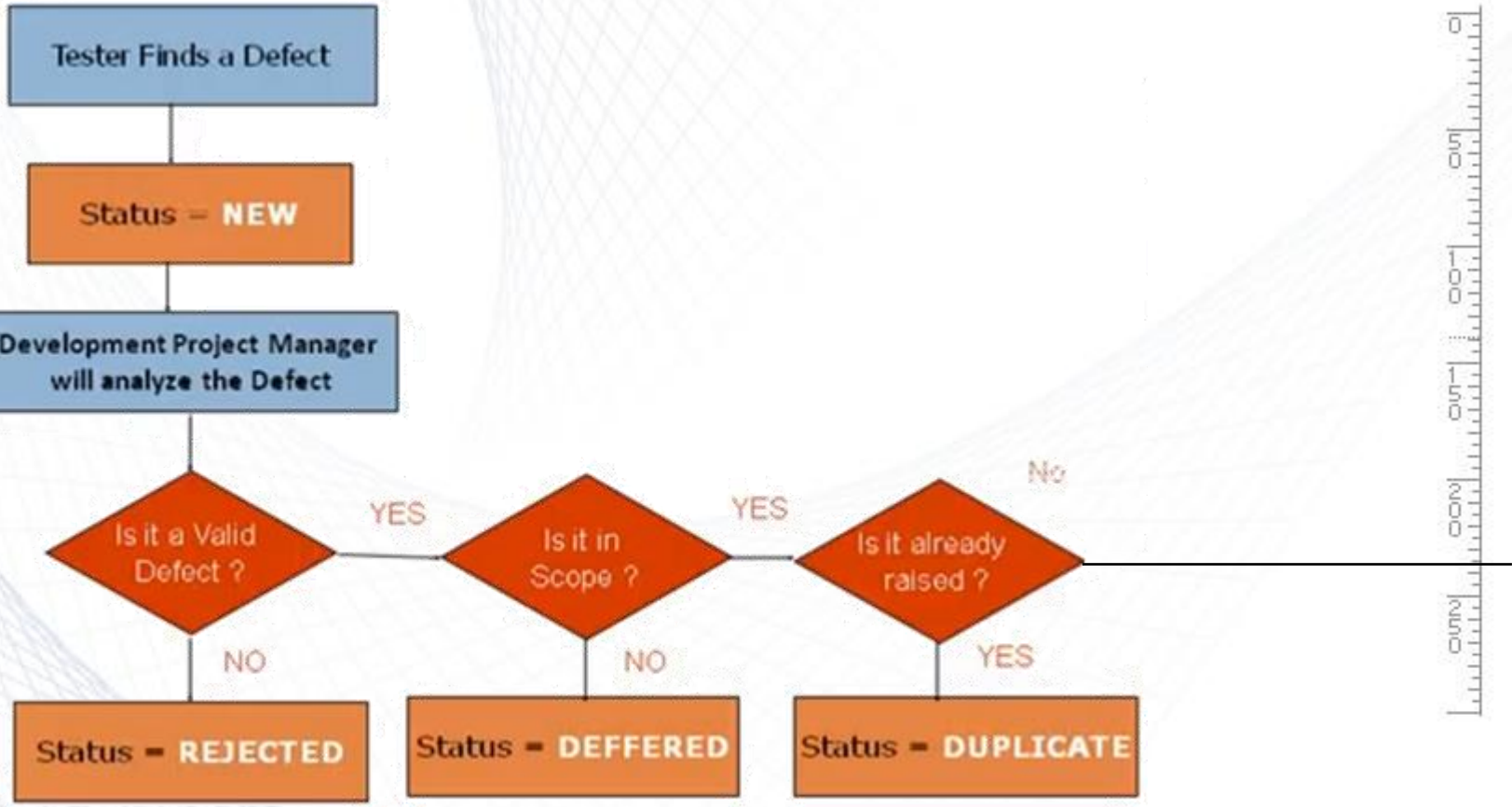
- Activities
  - Evaluate cycle completion criteria based on Time, Test coverage, Cost, Software, Critical Business Objectives, Quality
  - Prepare test metrics
  - Document the learning out of the project
  - Prepare Test closure report
  - Qualitative and quantitative reporting of quality of the work product to the customer
  - Test result analysis to find out the defect distribution by type and severity
- Deliverables
  - Test Closure report and Test metrics

Test Completion Criteria

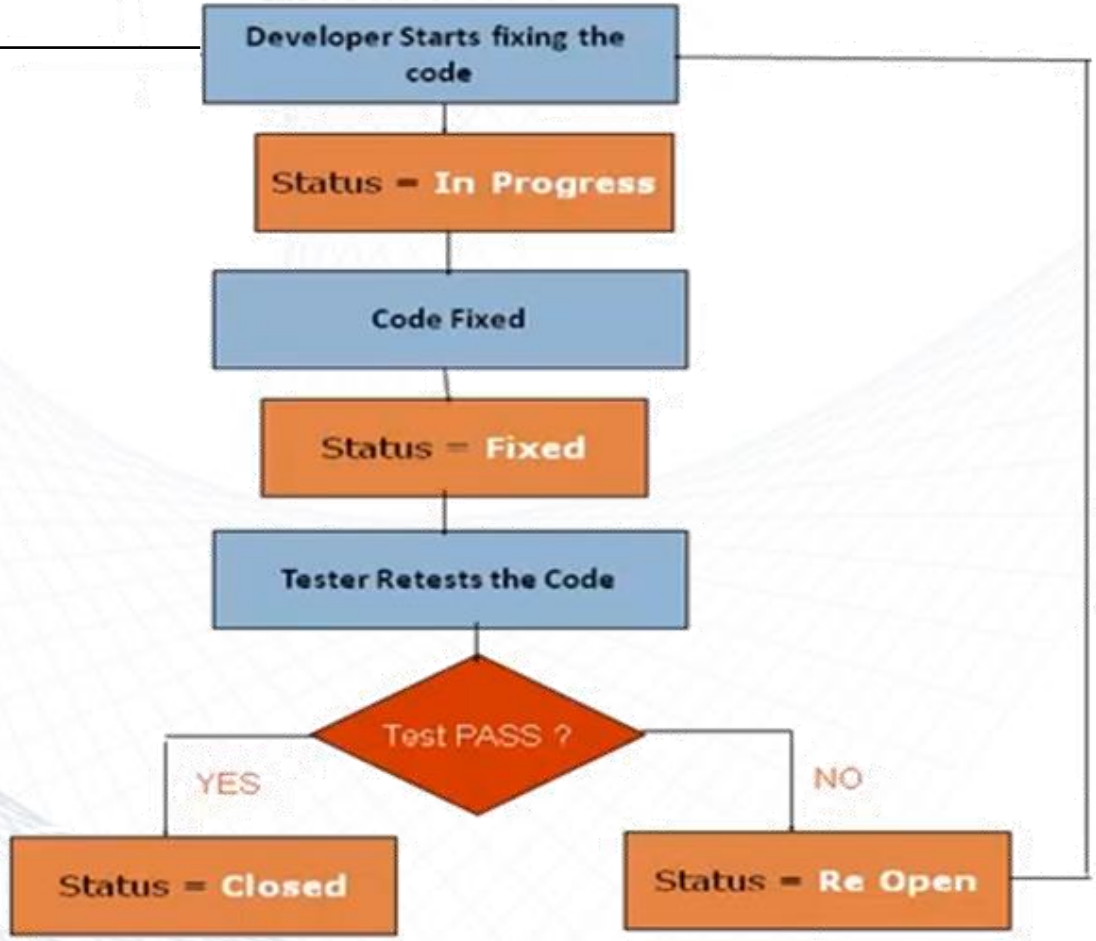
- Completion or exit criteria apply to all levels of testing to determine when to stop
  - Coverage, using a measurement technique,
    - e.g. branch coverage for unit testing
    - User requirements
    - Most frequently used transactions
  - Failures found (e.g. versus expected)
  - Cost or time
  - Quality

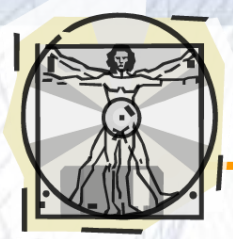
We will discuss this again later!

Failure Life Cycle



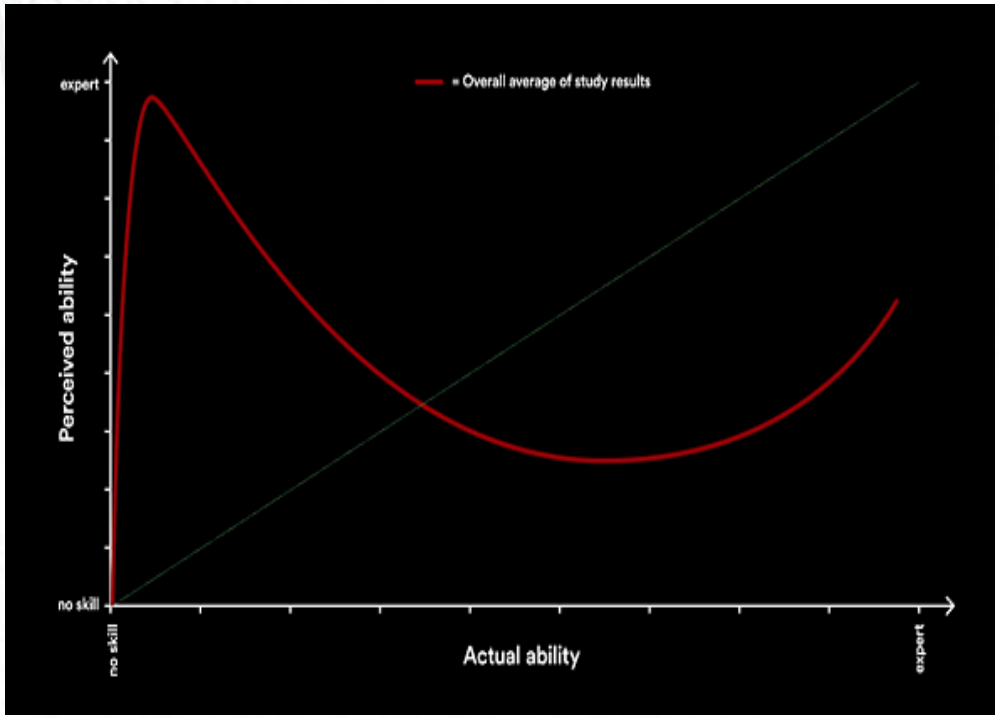
Failure Life Cycle



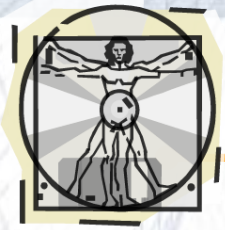


Let's See How Much We Know …

- After doing Assignment 1 we may think that we know everything about writing tests
- But …



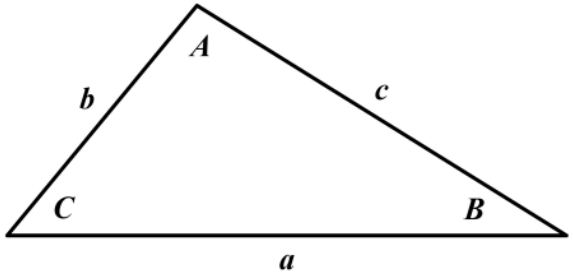
Dunning-Kruger Effect



Let's Write Some Test Cases

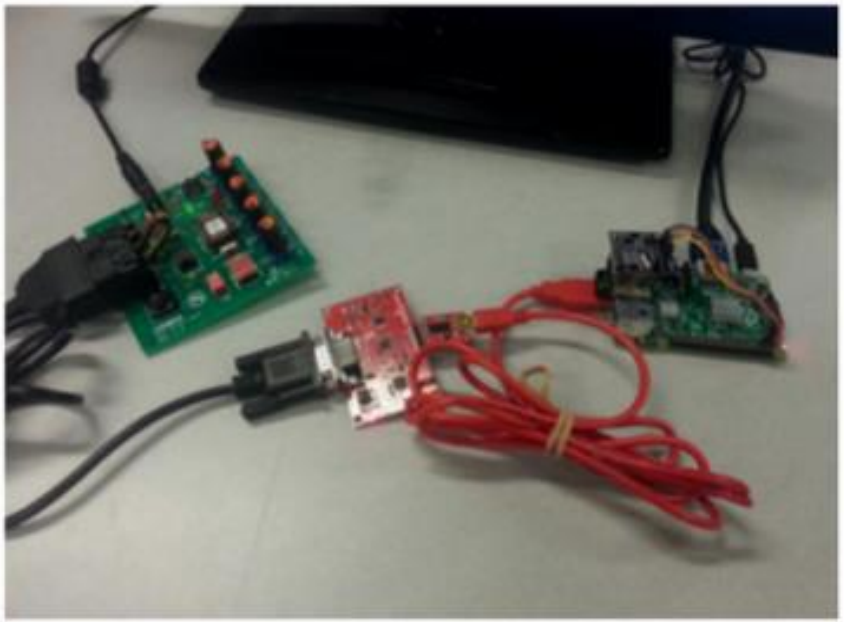
- Triangle program

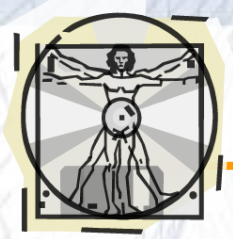
Toy



- IVMS project (automotive data acquisition and monitoring)

Real





Exercise: Triangle Program

- Triangle program

Requirements:

The triangle program accepts three integers, a, b, and c, as input. These are taken to be sides of a triangle. The integers a, b, and c must satisfy the following conditions:

C1) **1<= a <= 200**

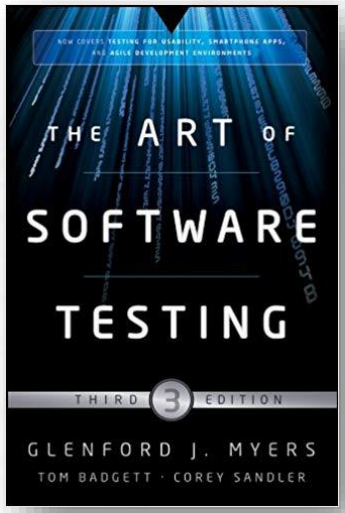
C2) **1<= b <= 200**

C3) **1<= c <= 200**

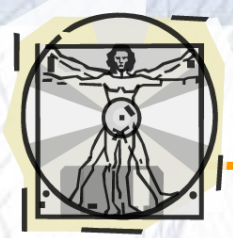
C4) **a < b + c**

C5) **b < a + c**

C6) **c < a + b**



Famous triangle program first Introduced in Myers' software testing book (1979)



Exercise: Triangle Program

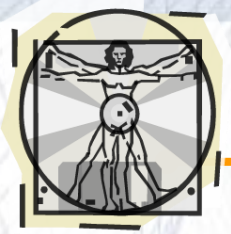
- Triangle program

Requirements: (cont'd)

If an input value fails any of conditions C1, C2, or C3, the program notes this with an output message, for example, "Value of b is not in the range of permitted values."

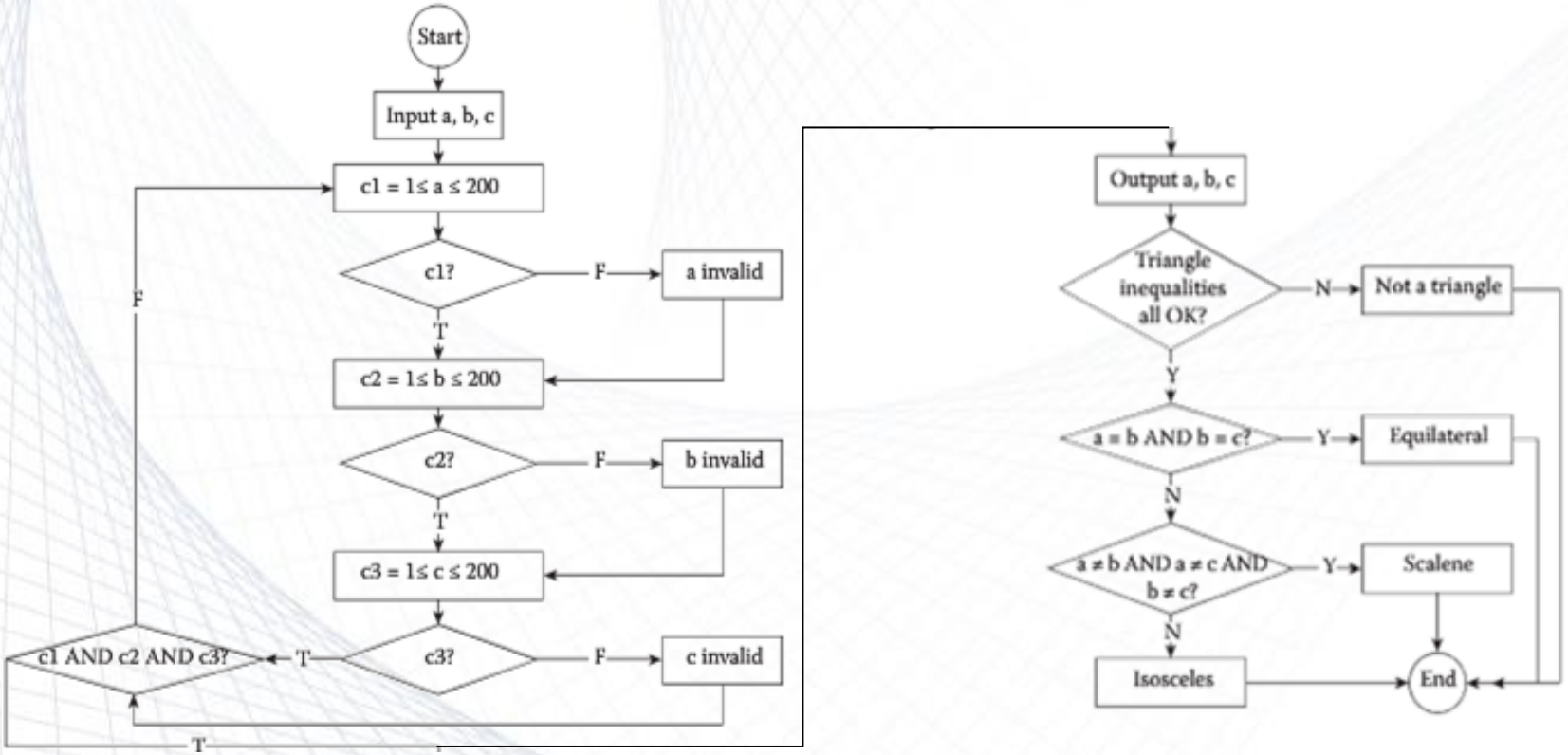
The output of the program is the type of triangle determined by the three sides:

- **If all three sides are equal, the program output is "Equilateral"**
- **If exactly one pair of sides is equal, the program output is "Isosceles"**
- **If no pair of sides is equal, the program output is "Scalene"**
- **If any of conditions C4, C5, and C6 is not met, the program output is "Not A Triangle"**

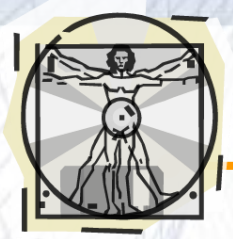


Exercise: Triangle Program

- Triangle program: Flow diagram



Only one way to implement this?

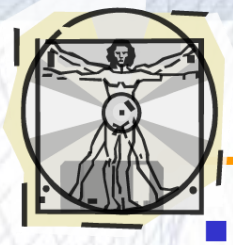


Exercise: Triangle Program

- Triangle program
- Write your exploratory test suite

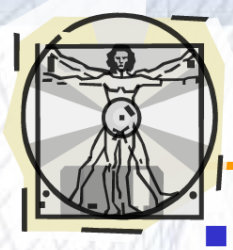
- How you write black-box tests?
- How you write white-box tests?
- How you write TDD tests?

Later



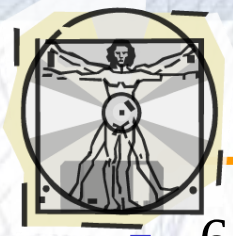
Triangle Program: Hints

- This program has at least three outputs: “scalene”, “isosceles”, “equilateral”…
  - I know that a set of test cases doesn’t have complete coverage if it doesn’t include at least one case for each program output.
  - I can easily write three test cases: a valid input for a scalene triangle, a valid input for an isosceles triangle, and a valid input for an equilateral triangle.
- Be careful!
  - “Do you have a test case that represents a *valid* scalene triangle? (Note that test cases such as 1,2,3 and 2,5,10 do not warrant a “yes” answer because there does not exist a triangle with these dimensions.)”
  - (Hmmm, I wrote “1,2,3”. I thought this was an interesting – because degenerate – case of a scalene triangle ;-)
  - Process question: how can I be sure my test cases are correct?



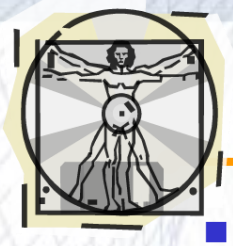
Triangle Program: Hints

- 1,2,3. do you have valid test cases for the three types of triangle?
  - My result: no I didn't in all cases, but I *thought* I did. And my mistake revealed an ambiguity in the program specification!
- 4. "Do you have at least three test cases that represent valid isosceles triangles such that you have tried all three permutations of three equal sides (such as, 3,3,4; 3,4,3; and 4,3,3)?"
  - My result: no.
  - Why this is important? Hmmm… now I get it! (Do you?)
- 5. "Do you have a test case in which one side has a zero value?"
  - My result: no, I didn't test for a degenerate (zero-area) isosceles triangle. This is an important boundary case which I should have tested. Ooops!



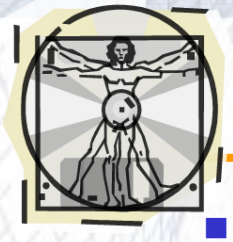
Triangle Program: Hints

- 6. Do you have a test case in which one side has a negative value?
  - Ouch, I really should have thought of that! I've been burned by that sort of latent bug before… !@#^&* unsigned ints …
- 7. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is equal to the third?
  - Well, no, but I'm finally remembering the "triangle inequality":
    - For any triangle, the sum of the length of any two sides is greater than or equal to the length of the third side.
    - For triangles drawn on a Euclidean plane, the inequality is strict: "For any non-degenerate triangle, the sum of the length of any two sides is greater than the length of the third side."
  - I finally know enough about the problem to write a good test suite!
    - If you don't discover the "boundary cases" you probably won't test them…



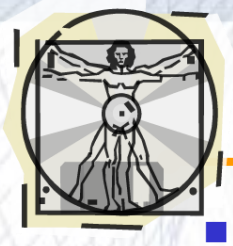
Triangle Program: Hints

- 7. ... "(That is, if the program said that 1,2,3 represents a scalene triangle, it would contain a bug)."
  - Oh… now you're telling me! I would have called it a "degenerate scalene triangle" – it's still scalene, but has zero area. But if you don't want to call it a triangle, that's fine by me: you write the spec, and I test it. I'll adjust my test cases now.
- 8. Do you have at least three test cases in category 7 (i.e. a degenerate triangle with zero area) such that you have tried all three permutations where the length of one side is equal to the sum of the lengths of the other two sides (for example, 1,2,3; 1,3,2; and 3,1,2)?
  - No… but I see what you're getting at. An input that is "strange" in one way (e.g. degenerate) may provoke strange behaviour in the program, so I should test such cases carefully (provoking all possible program outputs; different input permutations; …)



Triangle Program: Hints

- The initial set of test cases will, usually, trigger some questions about what the program is "designed to do" and what is "unintended".
  - Most program-specifiers don't think about what is "unintended".
  - Most program-specifiers assume that "everybody" will know what they mean by a "simple" word such as "triangle" or "input".
  - Test cases usually reveal ambiguities and gaps in the specification!
- But: you'll never get a job done if you keep asking questions.
  - Usually you have to "steam ahead". If you're methodical, you'll write down your unanswered questions, and prioritise them. (Should a tester be methodical?)
  - Which questions are important enough that they really *must* be answered *before* I deliver my first-draft test set? (My experience: none!!!!)

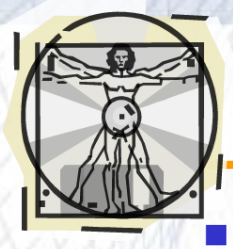


Triangle Program: Hints

- 9. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is less than the third (such as 1,2,4 or 12,15,30)?
  - Yes, I wrote this case. But your spec didn't say how the program is supposed to behave when it gets an input that isn't a triangle. I assumed the program should terminate normally, and output nothing. (Is this what you want it to do? I can't test for the "correct" production of an unspecified output.)
- 10. Do you have at least three test cases in category 9 such that you have tried all three permutations (for example, 1,2,4; 1,4,2; and 4,1,2)?
  - No, I didn't think about permuting that case. But it's very important, come to think of it! If the program doesn't check all three triangle inequalities (A+B<C, A+C<B, and B+C<A) then it'll accept some non-triangle inputs.

Triangle Program: Hints

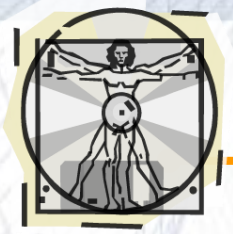
- 11. Do you have a test case in which all sides are zero (0,0,0)?
  - No. (Why is this important, I wonder? Ah… a "boundary" with the negative-length case!)
- 12. Do you have at least one test case specifying non-integer values (such as 2.5,3.5,5.5)?
  - Well, I did a little checking of the keyboard input, but my test was for integers that would overflow a 4-byte signed int: 10000000000, 20000000000, 40000000000. I also gave some thought to checking for semicolons, commas, and spaces as delimiters but I reckoned that this was my first-draft test set for a program with an unspecified input format. I'd prefer to see the first set of testing results before finalising my tests.
  - Is the program written in C, Java, Fortran, PHP? Different programming languages handle integer input quite differently, leading to different confusions over "unintended behaviour".



Triangle Program: Hints

- 13. "Do you have at least one test case specifying the wrong number of values (two rather than three integers, for example)?
  - Well, I certainly thought about writing one, then decided that was a test case for the "input dialog module". Now that I know I'm supposed to test a program that is handling its own keyboard input, I'll add more cases. I have been making a lot of incorrect assumptions about the input format you designed your program to handle. Do you want to tell me about your format now?
- 14. "For each test case did you specify the expected output from the program in addition to the input values."
  - Yes, But I incorrectly guessed what was "expected" for degenerate triangles. And I still don't know what the program was designed to output in cases where the input is not a triangle. So some of my output specifications are incorrect.

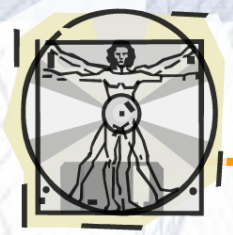
And this goes on and on and on …



Triangle Program Code

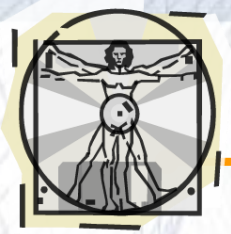
- What if we have access to the implemented code?
- What can go wrong here?

```
import java.util.*;

class Triangle {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        int a = sc.nextInt();
        int b = sc.nextInt();
        int c = sc.nextInt();


        if(a==b && b==c)
            System.out.println("Equilateral");

        else if(a >= (b+c) || c >= (b+a) || b >= (a+c) )
            System.out.println("Not a triangle");

        else if ((a==b && b!=c ) || (a!=b && c==a) || (c==b && c!=a))
            System.out.println("Isosceles");

        else if(a!=b && b!=c && c!=a)
            System.out.println("Scalene");
    }
}
```



IVMS Project: Problem & Goal

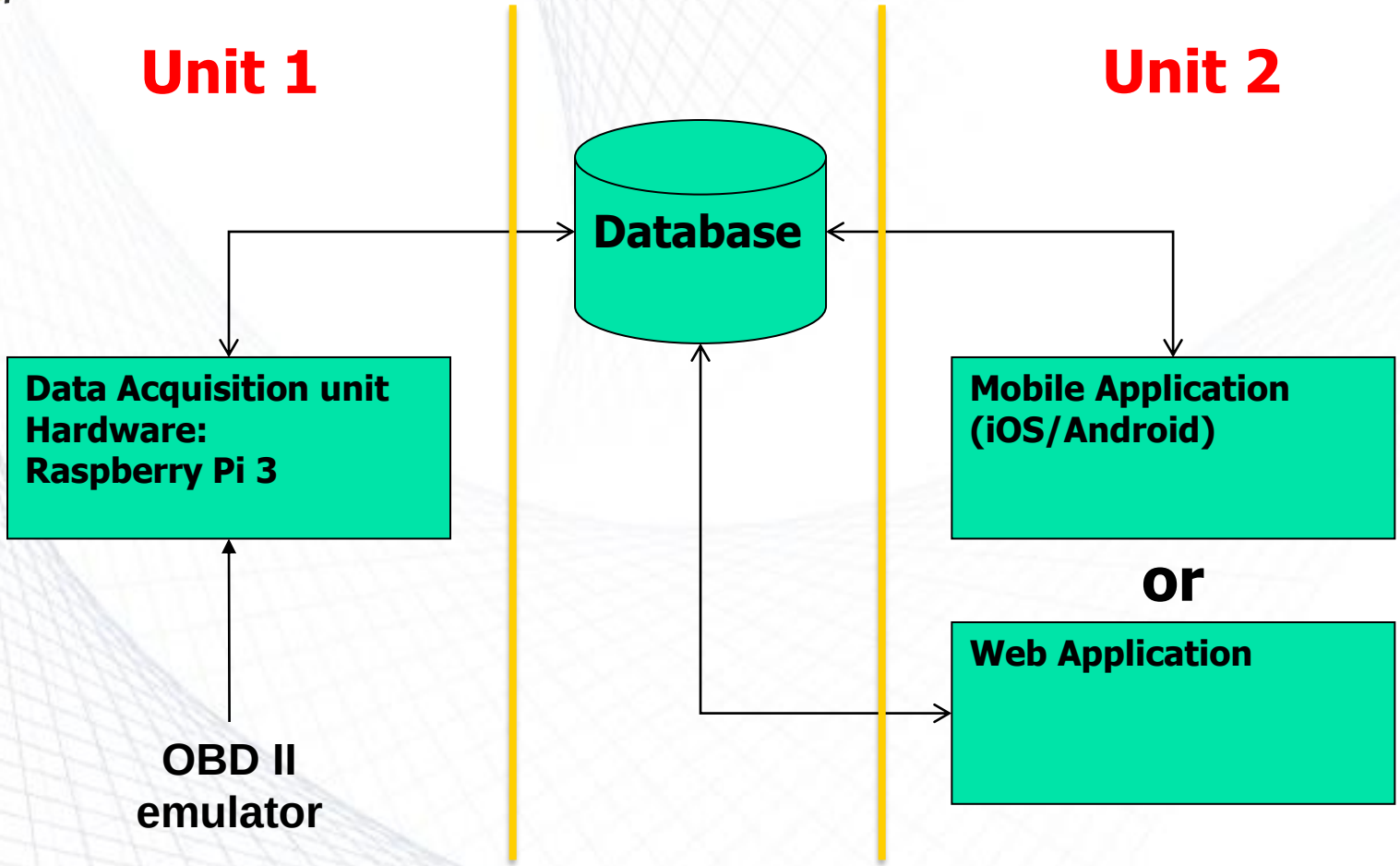
- An in-vehicle monitoring system to collect and transmit data from the OBDII (on-board diagnostic II system)
- Goal: design and develop a hardware and software solution for collecting, storing, and transmitting data from the OBDII of a car, to a backend server; and providing the remote users access to the collected data through a mobile or web application

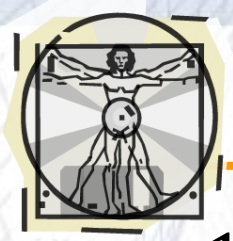


IVMS Project: Tasks

- The goal is accomplished by:
  - Connecting the RPi to the OBDII and gathering, filtering and saving data
  - RPi connecting to Wi-Fi when available
  - Sending saved data to the database located on a server
  - Providing access for remote users to retrieve data (through mobile or web applications)

IVMS: System Overview



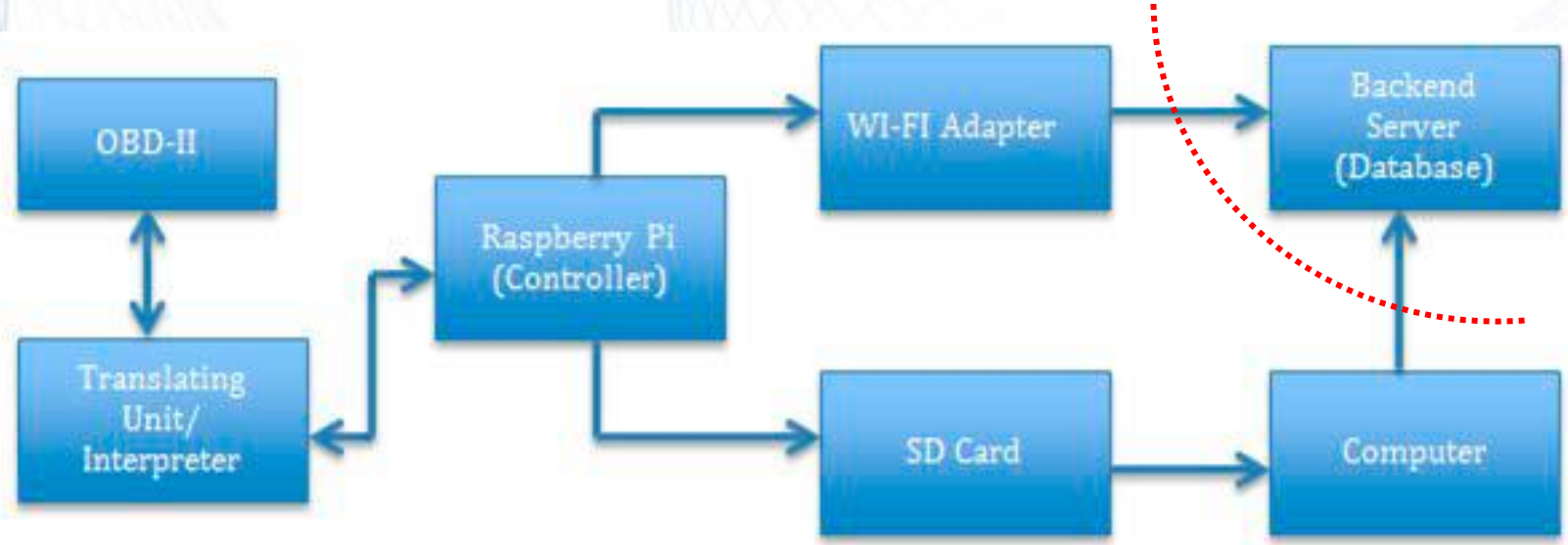


IVMS: System Specification

1) Data acquisition module
   - Given: Requirement document
   - To Do: Design of software residing on RPi
2) Database module
   - To Do: Design of a stand-alone database for storing the data collected from RPi unit
3) User Interface module
   - To Do: Web application or mobile application design

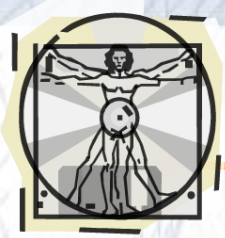
IVMS: System Hardware

- Data acquisition module



IVMS Project: Test Cases

- Write your exploratory test cases for the data acquisition unit
  - Do we need detailed design? code?
  - Select a few high level functions/features and check how they can be tested



Next …

- Unit testing using JUnit