



Software Verification and Validation

Course Material for SWEN 647 Software V&V

Abstract

SWEN 647 – A study of methods for evaluating software for correctness, efficiency, performance, and reliability. Skills covered include program proving, code inspection, unit-level testing, and system-level analysis. The difficulty and cost of some types of analysis and the need for automation of tedious tasks are examined. Emphasis is on problem-solving skills, especially in analyzing code.

Date Modified: 3/26/2020

Note from the Editor:

All material in this book is written by UMGC staff or is creative commons. You have the right to print and reproduce this books.

For some topics in the book better material was located in the UMGC library. For those topics references are provided to link to where the material is located in the library. Due to copyright law the material could not legally be copied into this book. But the UMGC library provides the material free of charge. If you would like to keep a copy of the material for reference once you level UMGC this material can be purchased throughout book stores like Amazon and EBay.

Table of Contents

1. Introduction.....	8
1.1. Why this is important.	8
Built-In Quality	8
The Testing Mindset	9
1.2. Different Techniques (how to pick the right kinds)	10
Test Early, Test Often	10
1.3. Verification vs Validation	11
1.4. Quality systems Cmmi and Iso.....	11
CMMI	11
ISO	12
2. Documents	13
2.1. Test Plan.....	13
The Purpose of Test Planning	13
Risk in Software.....	13
Software in Many Dimensions.....	14
Test Identification	16
Test Estimation	16
Refining the plan.....	16
2.2. Test Case	17
Elements of a Traditional Test Case	17
Need for Specific Input and Output Values in Test Cases.....	19
2.3. Test Suite.....	20
2.4. Test Report	20
3. Category of tests.....	21
3.1. Static vs dynamic analysis	21

Static Analysis	21
Dynamic Analysis	21
3.2. White-box vs Black-box testing.....	22
Black-box Testing	22
White-box Testing	22
3.3. Functional tests.....	22
3.4. Structural tests	23
3.5. Unit, Integration and System Testing.....	23
Unit Testing	23
Integration Testing	23
System Testing.....	24
3.6. Installation testing.....	24
3.7. Regression testing.....	24
3.8. Examples	25
Example 1	25
Example 2	25
Example 3	26
4. Common Testing Techniques	27
4.1. Code/Statement coverage	27
4.2. Boundary test.....	27
4.3. Code inspect.....	27
4.4. Code review.....	27
4.5. Examples	27
5. Unit Testing Automation	29
5.1. Introduction to JUnit	29
When a defect is found	30

5.2.	Additional Readings on JUnit.....	30
5.3.	JUnit Website	30
6.	User Interface Testing Automation.....	31
7.	Human-Based Testing Techniques.....	32
7.1.	Act Like A Customer	32
7.2.	User Acceptance testing.....	32
7.3.	Alpha and Beta Testing	33
	Alpha Testing.....	33
	Beta Testing	33
8.	Static Analysis – FindBug	34
9.	Formal methods (proofs).....	35
	Introduction.....	35
	Background.....	37
	Pre and Post Conditions	39
	Simple Proof	39
	Loops.....	42
	Formal Method Dispute	43
	Formal Methods and Software.....	44
10.	Symbolic execution.....	46
	Introduction.....	46
	Advantages and Disadvantages.....	46
	Execution Table with Specific Values.....	46
	Execution Table with Symbolic Values.....	50
	Incorporating Conditions into Symbolic Execution	52
	Incorporating Loops into Symbolic Execution	57
	Software Used for Symbolic Execution.....	64

11. Model checking.....	66
Microwave Example	68
Division by Zero Example	71
Software Used for Model Checking	73
12. Throughout the SDLC	74
12.1. Verification and Validation.....	74
Verification	74
Validation.....	74
Examples.....	74
12.2. Complexity of code.....	75
Measuring Complexity.....	77
13. Defect management.....	80
13.1. Defect Report.....	80
Importance of Good Defect Reporting	80
Characteristics of a Good Defect Report	80
Isolation and Generalisation	81
Severity	81
Status.....	82
Elements of a Defect Report	83
13.2. Reporting and Metrics	84
Defect Reporting	84
Metrics of Quality and Efficiency.....	85
Defect Injection Rate	86
Defect Discovery Rate	86
14. Testing the test suite.....	88
14.1. Mutation Testing	88

Example	88
---------------	----

1. Introduction

The following comes from Jenkins, N., 2017. A Software Testing Primer. *An Introduction to Software Testing v2*.

1.1. Why this is important.

Software development involves ambiguity, assumptions and flawed human communication. Each change made to a piece of software, each new piece of functionality, each attempt to fix a defect, introduces the possibility of error. With each error, the risk that of failure increases. In software of any reasonable complexity the risk quickly reaches unacceptable levels.

How do you assess the level of risk?

How do you know if your software works before you inflict it upon your customers?

Testing is a search for the truth. As Kem Caner, Brett Pettichord and James Bach said, testing is ‘applied epistemology’ – the applied search for justified belief. In philosophical terms, you can think you know something, but without reason or evidence, that belief is not justified; it is akin to false hope. You need evidence, logic or a reason to transform your belief into what a philosopher would call ‘knowledge’.

You can hope your software works or you can know it works – through testing. So testing is also a mindset and an attitude. This book is for anyone that wants to develop a testing mindset: a critical, evaluating mental process that mirrors the principles of the scientific method. But this book extends beyond that to a consideration of all kinds of quality assurance in software development. This book is for anyone that wants to make good software.

Built-In Quality

In manufacturing they have long known you can’t ‘inspect-in’ quality, it must be ‘built-in’. The goal is not to fix the defect – it is to diagnose how the defect was introduced and to engineer ways to avoid similar defects in future. Fixing the defect once delivers a tiny benefit. Fixing the process or removing the source of the defect delivers a compounding return that pays for itself many times over.

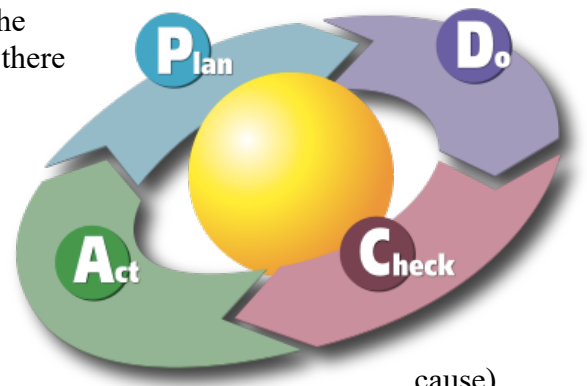
Suppose you are making cars, and a new car arrives at the end of the production line with three wheel-nuts on one wheel. One way to ‘fix’ the problem is to send off someone to find a ‘lug wrench’ and an extra wheel-nut to. But that will tie someone up in a pointless task, and the car will have to sit somewhere while it is being fixed. If it happens a lot you might need someone on standby with a wrench and a box of nuts!

By why not stop it from happening? Why not figure out why cars make it down the line with only three wheel nuts, and find a way to prevent that from happening in future? But how do you fix something at source? The temptation is always to rush to an immediate fix. First you have to stop and find the source,

you have to find the root cause. Stop and ask yourself “why” the defect occurred? What caused it? Was it a simple error or was there confusion or misunderstanding? Why did that occur?

Then you have to have a process for implementing a countermeasure or improvement, for evaluating if it worked and finally you have to have a way of standardising the change if it proves successful.

In ‘Lean’ this is known as the PDCA cycle or Plan-Do-Check-Act. When you detect a problem (and diagnose the cause) you plan a simple countermeasure to avoid the problem and you predict what will happen. Then you do the change. You then check the results and see if you get the outcome you expected, if not you can try again. If your change gave you the outcome you wanted, then you can act to roll out the change across your production line.



Another way of understanding this is to think about feedback loops – the shorter a feedback loop the faster you learn. The longer the time between when a defect is introduced and when it is found, the more rework you incur and the less you learn. If you shorten these loops then the contextual knowledge is fresher and leads to greater insight. You can then apply that knowledge to ‘error-proofing’ the process.

The Testing Mindset

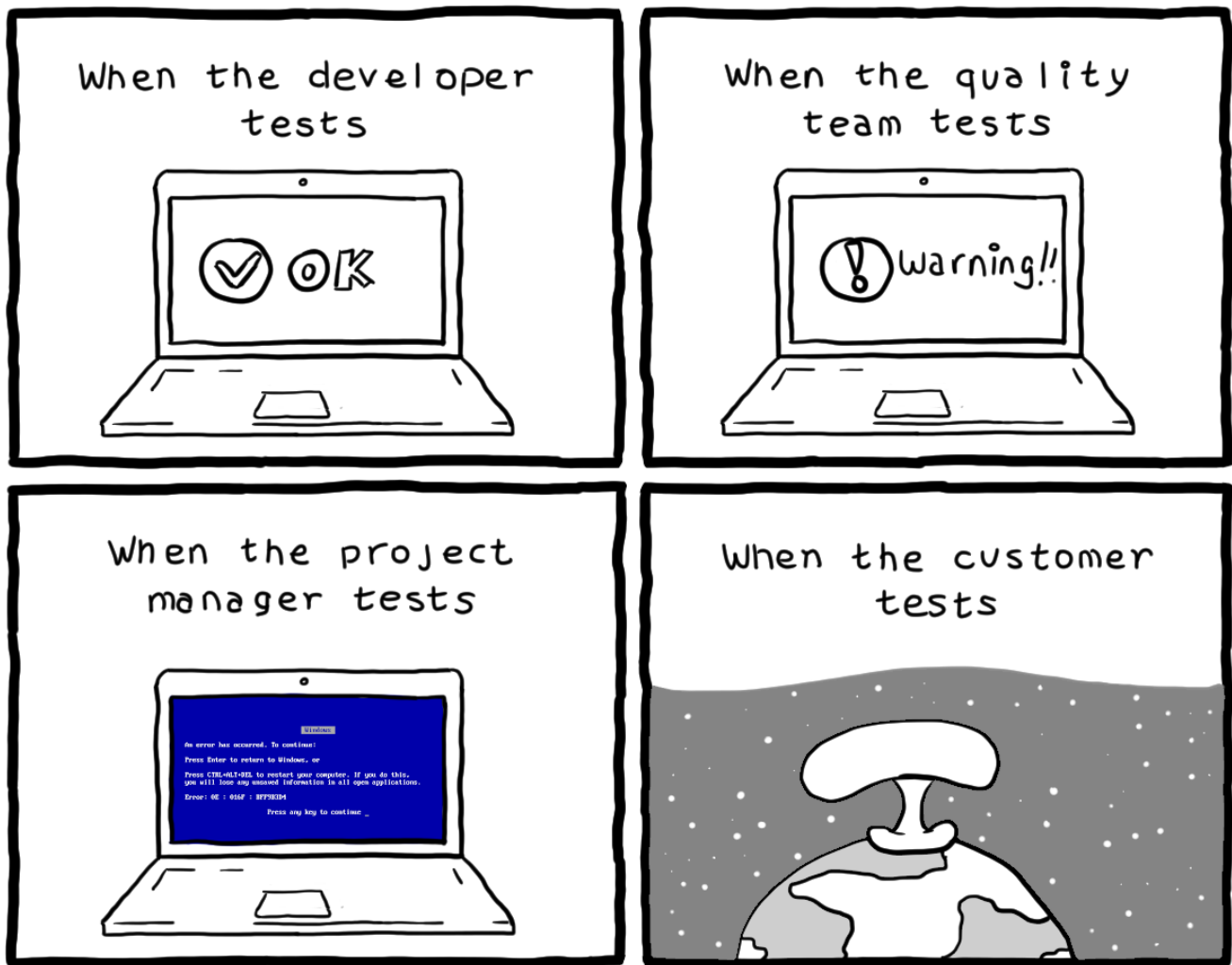
There is particular mindset that accompanies “good testing” – the assumption that product is already broken and it is your job to discover it. You assume the product or system is inherently flawed and it is your job to ‘illuminate’ the flaws.

Designers and developers often approach software with an optimism based on the assumption that the changes they make are the correct solution. How could they do anything else? But they are just that – assumptions. Someone with a testing mindset is aware of their assumptions, and the possible limitations. Without being proved assumptions are no more correct than guesses. Individuals often overlook fundamental ambiguities in requirements in order to deliver a solution; or they fail to recognise them when they see them. Those ambiguities are then built into the code and represent a defect when compared to the end-user's needs.

By taking a sceptical approach, we offer a balance.

Take nothing at face value. Always asks the question “why is it that way?” Seek to drive out certainty where there is none. Illuminate the darker part of the projects with the light of inquiry.

Sometimes this attitude can bring conflict to a team, but it can be healthy conflict. If people recognise the need for a dissenting voice, for a disparity of opinions, then the tension is constructive and productive. But often when the pressure bites and the deadline is looming the dissenting voice is drowned out, sometimes to the organisation's peril.



This image is license under Creative Commons for turnoff.us/geek/software-test

1.2.Different Techniques (how to pick the right kinds)

Test Early, Test Often

There is an oft-quoted truism of software engineering that states - “a bug found at design time costs ten times less to fix than one in coding and a hundred times less than one found after launch”. Barry Boehm, the originator of this idea, actually quotes ratios of 1:6:10:1000 for the costs of fixing bugs in requirements, design, coding and implementation phases (waterfall).

If you want to find bugs, start as early as is possible.

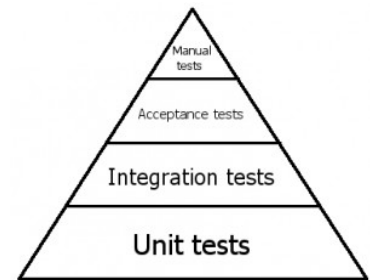
That means unit testing (qqv) for developers, integration testing during assembly and system testing. This is a well understood tenet of software development that is simply ignored by the majority of software development efforts.

This concept is epitomised by the (automated) testing triangle promoted in various circles. Like most aphorisms, there is truth in it but it oversimplifies things and it should be interpreted in your own context. Certainly you should have a well thought out structure to your testing.

In modern software development it is common to use continuous integration and delivery (qqv CI/CD). Every change a developer makes is tested (and possibly deployed) every time they commit it to the code repository.

Nor is a single pass of testing enough.

Your first pass at testing simply identifies where the defects are. At the very least, a second pass of (post-fix) testing is required to verify that defects have been resolved. The more passes of testing you conduct the more confident you become and the more you should see your project converge on its delivery date. As a rule of thumb, anything less than three passes of testing is inadequate.



1.3.Verification vs Validation

Sometime individuals lose sight of the end goal. They narrow their focus to the immediate phase of software development and lose sight of the bigger picture.

Verification tasks are designed to ensure that the product is internally consistent. They ensure that the product meets the specification, the specification meets the requirements... and so on. The majority of testing tasks are *verification* – with the final product being checked against some kind of reference to ensure the output is as expected.

For example, test plans can be written from requirements documents and from specifications. This verifies that the software delivers the requirements or meets the specifications. This however does not however address the ‘correctness’ of those documents!

Validation tasks are just as important as verification, but less common.

Validation is the use of external sources of reference to ensure that the internal design is valid, i.e. it meets user’s expectations. By using external references (such as the direct involvement of end-users) the test team can validate design decisions and ensure the project is heading in the correct direction (but sometimes all you have is validation, there is no ‘internal’ point of reference).

1.4.Quality systems Cmmi and Iso

CMMI

To get a good understanding of CMMI read Appendix A of Software Project Management: A Process-Driven Approach, by Ashfaque Ahmed, copyright 2012.

You should be able to access it through this link. If not, go to <http://www.umgc.edu/library> Click on E-Books, Click on What’s in this collection, click on Books 24x7.

<http://library.books24x7.com.ezproxy.umgc.edu/assetviewer.aspx?bookid=47182&chunkid=226550580&rowid=1424>

ISO

To get a good understanding of ISO read Appendix B of Software Project Management: A Process-Driven Approach, by Ashfaq Ahmed, copyright 2012.

You should be able to access it through this link. If not, go to <http://www.umgc.edu/library> Click on E-Books, Click on What's in this collection, click on Books 24x7.

<http://library.books24x7.com.ezproxy.umgc.edu/assetviewer.aspx?bookid=47182&chunkid=945971966&rowid=1459>

2. Documents

The following comes from Jenkins, N., 2017. A Software Testing Primer. *An Introduction to Software Testing v2*.

2.1. Test Plan

The Purpose of Test Planning

As an organised activity, testing should be planned, to a reasonable extent.

But test planning represents a special challenge.

The aim of testing is to find bugs in the product and so the aim of planning is to plan how to find the bugs in the product. The paradox is of course, that if we knew where the bugs were then we could fix them without having to test for them.

Testing is the art of uncovering the unknown and therefore can be difficult to plan.

The usual, naïve retort is that you should simply test “all” of the product. Even the simplest program however will defy all efforts to achieve 100% coverage.

Even the term coverage itself is misleading since this represents a plethora of possibilities. Do you mean code coverage, branch coverage, or input/output coverage ? Each one is different and each one has different implications for the development and testing effort. The ultimate truth is that complete coverage, of any sort, is simply not possible (nor desirable).

So how do you plan your testing?

At the start of testing there will be a (relatively) large number of issues and these can be uncovered with little effort. As testing progress more and more effort is required to uncover subsequent issues.

The law of diminishing returns applies and at some point the investment to uncover that last 1% of issues is outweighed by the high cost of finding them. The cost of letting the customer or client find them will actually be less than the cost of finding them in testing (see A/B testing).

The purpose of test planning therefore is to put together a plan which will deliver the right tests, in the right order, to discover as many of the issues with the software as time.

Risk in Software

Risk is based on two factors – the likelihood of the problem occurring and the impact of the problem when it does occur. For example, if a particular piece of code is complex then it will introduce far more errors than a simple module of code. Or a particular module of code could be critical to the success of the product. Without it functioning perfectly, the product simply will not deliver its intended result.

Both of these areas should receive more attention and more testing than less 'risky' areas.

But how to identify those areas of risk?

As already mentioned, complexity is a good proxy for probability – the more complex the code the more likely it is that an error will be introduced. In existing code, the historical location of defects can be a good guide to likely areas of complexity. In new code, effort or duration can be good proxies for complexity – the more complex or difficult the code, the longer it will take to write.

But impact or criticality is much harder to measure. Business value can be a proxy for criticality, but not always. Business users might not regard the phone or the internet as highly valuable but they are critical pieces of infrastructure that they would struggle to operate without them.

What we need therefore is a recursive model of the software to help guide our testing.

Software in Many Dimensions

It is useful to think of software as a multi-dimensional entity with many different axes.

For example one axis is the code of the program, which will be broken down into modules and units. Another axis will be the input data and all the possible combinations. Still a third axis might be the hardware that the system can run on, or the other software the system will interface with.

Testing can then be seen as an attempt to evaluate as many of these axes as possible. Remember we are no longer seeking the impossible 100% coverage but merely 'best' coverage, based on risk and budget.

Outlining Taxonomy

To start the process of test planning a simple process of 'outlining' can be used :

1. List all of the 'axes' or areas of the software on a piece of paper (a list of possible areas can be found below, but there are certainly more than this).
2. Take each axis and break it down into its component elements. For example, with the axis of "code complexity" you would break the program down into the 'physical' component parts of code that make it up. Taking the axis of "hardware environment" (or platform) it would be broken down into all possible hardware and software combinations that the product will be expected to run on.
3. Repeat the process until you are confident you have enumerated as much of each axis as you possibly can (you can always add more later).
4. Recurse into each sub-area of each axis if necessary in order to build a taxonomy of your 'testing space'. For example if you have a large monolithic application it may warrant breaking it down into two to three levels of functionality until you reach the right 'granularity'. Each axis will have a different taxonomy, platform for example might map browser, O/S and hardware combinations.

Common Axes

The most common starting point for test planning is a functional decomposition based on a technical specification or an examination of the product itself. This is an excellent starting point but should not be the sole 'axis' which is addressed – otherwise testing will be limited to 'verification' but not 'validation'.

Axis	Category Explanation
Functionality	As derived from some other reference
Code Structure	The organisation and break down of the source or object code
User interface elements	User interface controls and elements of the program
Internal interfaces	Interface between modules of code (traditionally high risk)
External interfaces	Interfaces between this program and other programs
Input space	All possible inputs
Output space	All possible outputs
Physical components	Physical organisation of software (media, manuals, etc.)
Data	Data storage and elements
Platform and environment	Operating system, hardware/software platforms, form factors
Configuration elements	Any modifiable configuration elements and their values
Performance	Different aspects of performance in different scenarios

Test Identification

The next step is to identify tests which 'exercise' each of the elements in your outline. This isn't a one-to-one relationship. Many tests might be required to validate a single element of your outline and a single test may validate more than one point on one axis. For example, a single test could simultaneously validate functionality, code structure, a UI element and error handling.

For now, don't worry too much about the detail of how you might test each point, simply decide 'what' you will test.

Test Estimation

Once you have prioritised the test cases you can estimate how long each case will take to execute. Take each test case and make a rough estimate of how long you think it will take to set up the appropriate input conditions, execute the software and check the output. No two test cases are alike but you can get a good enough estimate by assigning an average execution time and multiplying by the number of cases. Total up the hours involved and you have an estimate of testing for that piece of software.

You can then negotiate with your product manager for the appropriate budget to execute the testing. The final cost you arrive at will depend on the answers to a number of questions, including: how deep are your organisation's pockets? how mission critical is the system? how important is quality to the company? what is the organisation's risk appetite? how reliable is the development process?

The number will almost certainly be lower than you hoped for.

Refining the plan

As you progress through each cycle of testing you can refine your test plan. Typically, during the early stages of testing not many defects are found. Then, as testing hits its stride, defects start coming faster and faster until the development team gets on top of the problem and the curve begins to flatten out. As development moves ahead and as testing moves to retesting fixed defects, the number of new defects will decrease.

This is the point where your risk/reward ratio begins to bottom out and it may be that you have reached the limits of effectiveness with this particular form of testing. If you have more testing planned or more time available, now is the time to switch the focus of testing to a different point in your outline strategy.

Cem Kaner said it best, "The best test cases are the ones that find bugs." A test case which finds no issues is not worthless but it is obviously worth less than a test case which does find issues. The maximum amount of information is generated at a failure rate of about 50% - if your tests are failing less than 50% of the time then they are not adding maximum value.

If your testing is not finding any bugs then perhaps you should be looking somewhere else. Conversely, if your testing is finding a lot of issues you should pay more attention to it – but not to the exclusion of everything else, there's no point in fixing just one area of the software!

Your cycle of refinement should be geared towards discarding pointless or inefficient tests and diverting attention to more fertile areas for evaluation.

This is where automation can be your friend. If the incremental cost of running the test is small then you can continue to run it without loss (regression testing being the prime example). If the cost of the test is high (say with intensive manual testing) then you should consider diverting your effort elsewhere.

Also, referring each time to your original outline will help you avoid losing sight of the wood for the trees. While finding issues is important you can never be sure where you'll find them so you can't assume the issues that you are finding are the only ones that exist. You must keep a continuous level of broad coverage active while you focus the deep coverage on the trouble spots.

2.2. Test Case

A test case documents a test, intended to prove a requirement or feature.

The relationship is not always one-to-one, sometimes many test case are required to prove one requirement. Sometimes the same test case must be extrapolated over many screens or many workflows to completely verify a requirement (there should usually be at least one test case per requirement however).

Some methodologies (like RUP) specify there should be two test cases per requirement – a positive test case and a negative test case. A positive test case is intended to prove that the function-under-test behaves as required with correct input and a negative test is intended to prove that the function-under-test does not provoke an error with incorrect input (or responds gracefully to that error).

Elements of a Traditional Test Case

The following table lists the suggested items a test case should include:

Item	Description
Title	A unique and descriptive title for the test case
Priority	The relative importance of the test case (critical, nice-to-have, etc.)
Status	<p>For live systems, an indicator of the state of the test case.</p> <p>Typical states could include :</p> <ul style="list-style-type: none">• Design – test case is still being designed• Ready – test case is complete, ready to run• Running – test case is being executed• Pass – test case passed successfully

	<ul style="list-style-type: none"> • Failed – test case failed • Error – test case is in error and needs to be rewritten
Initial Configuration	The state of the program before the actions in the “steps” are to be followed. All too often this is omitted and the reader must guess or intuit the correct pre-requisites for conducting the test case.
Software Configuration	The software configuration for which this test is valid. It could include the version and release of software-under-test as well as any relevant hardware or software platform details (e.g. WinXP vs Win95)
Steps	An ordered series of steps to conduct during the test case, these must be detailed and specific. How detailed depends on the level of scripting required and the experience of the tester involved.
Expected Behavior	What was expected of the software, upon completion of the steps? What is expected of the software. Allows the test case to be validated without recourse to the tester who wrote it.

This sub-section was written by Dr. Michael Brown of UMGC.

There are other formats for test cases and some organizations make customer test case formats to meet their needs.

For Unit Testing this is a common format.

Item	Description
Title (Test Case Name)	A unique and descriptive title for the test case
Methods being tested:	The method (and class in object oriented systems) that the test case will execute.
Short Description	Short description
Input Data to constructor and/or method you are testing.	Specific values for each input parameter for the test.

Expected Results	The output of the method for the test.
------------------	--

For testing functions or feature from the User Interface of the application, this is a common format.

Item	Description
Name	A unique and descriptive title for the test case
Description	Short description of the test case.
Requirements	The requirement(s) in the system that this test case is attempting to test.
Prerequisites	If there are any prerequisites for the test running. A common example is that the user is logged into the system. You might have a test case or two for logging in and then a prerequisite for all other test cases is that the user is logged in. If they are not prerequisites just put "N/A".
Steps	This is a numbered list of test steps between user activity and system responses.
Expected Output	The final expected output
Assumptions	Any assumptions that were made by the writer of the test case.

Need for Specific Input and Output Values in Test Cases

It is important that test cases have specific values for inputs and outputs. There are a number of reasons for this.

Reproducible: If a test case uncovers a defect it needs to be reported to the developers to fix. When there are not specific input and output of the test the developer might not be able to reproduce the defect.

Cannot Meet Testing Criteria: Many testing criteria (like code coverage or boundary) cannot be determined without specific values

Speed of testing: Without specific values testers are left to figure out values and expected values as they run the test. This consumes time. In some cases it may take a lot of time to independently determine the expected result.

Impossible to determine expected result: In some cases it might be nearly impossible for the tester to determine the expected result with their background.

Cannot use the system to determine expected result: Too often the test users the system to compute the expected result. That is not good, since we cannot rely on the system that we are testing.

2.3.Test Suite

The term Test Suite is simply a group of Test Cases. Typically each application has a single Test Suite, but very large applications could have many Test Suites. There could be a Test Suite for different parts of the application. Or there could be a small Test Suite to run after minor releases and a larger Test Suite to run after major releases.

2.4.Test Report

A Test Report is a document that describes the results of running a Test Suite. For each Test Suite of an application you will have multiple Test Reports; one Test Report each time that you execute the Test Suite.

A typical format for a Test Report is shown below.

Test Case Identifier (Title or #)	Outcome (Pass/Fail)	Comments
Login Valid UserID/Password	Passed	
Login Invalid UserID/Password	Failed	Application crashed with “Database invalid SQL” error.

3. Category of tests

There are many different categories of testing activities. Many tests may fall into multiple different categories. This chapter covers the most common categories of tests.

There is a difference between the term *Testing* and *Analysis*. A test is a case in which the system will pass or fail. Those are the only two outcomes to a test. There is no middle ground. Analysis is a process that produces knowledge about the system. In many cases the categories in this chapter can have both testing and analysis. For example, there can be Dynamic Testing and Dynamic Analysis. Some people do not really differentiate between the terms and use the terms interchangeably.

The following comes from Jenkins, N., 2017. A Software Testing Primer. *An Introduction to Software Testing v2*.

3.1.Static vs dynamic analysis

Static Analysis

Static analysis techniques revolve around looking at the source code, or uncompiled form of software. They rely on examining the basic instruction set in its raw form, rather than as it runs. They are intended to trap semantic and logical errors or security flaws.

Code inspection is a specific type of static analysis. It uses formal or informal reviews to examine the logic and structure of software source code and compare it with accepted best practices.

In large organisations or on mission-critical applications, a formal inspection board can be established to make sure that written software meets the minimum required standards. In less formal inspections a development manager can perform this task or even a peer.

Code inspection can also be automated. Many syntax and style checkers exist today which verify that a module of code meets certain pre-defined standards. By running an automated checker across code it is easy to check basic conformance to standards and highlight areas that need human attention.

Dynamic Analysis

While static analysis looks at source code in its raw format, dynamic analysis looks at the compiled/interpreted code while it is running in the appropriate environment. Normally this is an analysis of variable quantities such as memory usage, processor usage or overall performance.

One common form of dynamic analysis used is that of memory analysis. Given that memory and pointer errors form the bulk of defects encountered in software programs, memory analysis is extremely useful. A typical memory analyser reports on the current memory usage level of a program under test and of the disposition of that memory. The programmer can then ‘tweak’ or optimise the memory usage of the software to ensure the best performance and the most robust memory handling.

Often this is done by ‘instrumenting’ the code. A copy of the source code is passed to the dynamic analysis tool which inserts function calls to its external code libraries. These calls then export run time data on the source program to an analysis tool. The analysis tool can then profile the program while it is running. Often these tools are used in conjunction with other automated tools to simulate realistic conditions for the program under test. By ramping up loading on the program or by running typical input data, the program’s use of memory and other resources can be accurately profiled under real-world conditions.

3.2.White-box vs Black-box testing

Black-box Testing

Testing of completed units of functional code is known as black-box testing because the object is treated as a black-box: input goes in; output comes out. These tests concern themselves with verifying specified input against expected output and not worrying about the mechanics of what goes in between.

User Acceptance Testing (UAT) is the classic example of black-box testing.

White-box Testing

White-box or *glass-box* testing relies on analysing the code itself and the internal logic of the software. White-box testing is often, but not always, the purview of programmers. It uses techniques which range from highly technical or technology specific testing through to things like code inspections or pairing.

Unit testing is a classic example of white-box testing. In unit testing developers use the programming language they are working in (or an add-on framework like xUnit) to write short contracts for each unit of code. What constitutes a unit is the subject of some debate but unit tests are typically very small, independent tests that run quickly and assert the behaviour of a particular function or object. For example a function that adds numbers together might be tested by checking that 1 plus 1 returns 2.

While this might sound trivial, unit tests are useful for debugging code, particularly complex code that changes frequently. A developer may make a change that they think does not affect the output of a function, but a well written unit test will prove that the functions contract has not been broken. Unit tests also help design and refactor complex code as they force programmers to break down the logic of their application in to manageable chunks.

3.3.Functional tests

If the aim of a software development project is to “deliver widget X to do task Y” then the aim of “functional testing” is to prove that widget X actually does task Y.

Simple? Well, not really.

We are trapped by the same ambiguities that lead developers into error. Suppose the requirements specification says widget “X must do Y” but widget X actually does Y+Z? How do we evaluate Z? Is it necessary? Is it desirable?

Does it have any other consequences the developer or the original stake-holder has not considered? Furthermore how well does Y match the Y that was specified by the original stake-holder?

3.4.Structural tests

Just section 3.4 comes from What is White Box testing? Techniques of White-box testing by Penna Sparrow.

In this kind of testing the internal structure of the application is exposed to the tester. While designing the test cases internal coding and logic of the software is considered as well as the programming skills of the developer are tested.

3.5.Unit, Integration and System Testing

Unit Testing

The first type of testing that can be conducted in any development phase is *unit testing*.

In this, discrete components of code are tested independently before being assembled into larger units. Units are typically tested through the use of ‘test harnesses’ which simulate the context into which the unit will be integrated. The test harness provides a number of known inputs and measures the outputs of the unit under test, which are then compared with expected values to determine if any issues exist.

Integration Testing

In *integration testing* smaller units are integrated into larger units and larger units into the overall system. This differs from unit testing in that units are no longer tested independently but in groups, the focus shifting from the individual units to the interaction between them.

At this point “stubs” and “drivers” take over from test harnesses.

A stub is a simulation of a particular sub-unit which can be used to simulate that unit in a larger assembly. For example if units A, B and C constitute the major parts of unit D then the overall assembly could be tested by assembling units A and B and a simulation of C, if C were not complete. Similarly if unit D itself was not complete it could be represented by a “driver” or a simulation of the super-unit.

Unit D (complete)

Unit A Complete	Unit B Complete	Unit C Not Complete
--------------------	--------------------	------------------------

Unit D (incomplete)

Unit A Complete	Unit B Complete	Unit C Complete
--------------------	--------------------	--------------------

DRIVER

		STUB
--	--	------

--	--	--

Figure. Integration Testing

As successive areas of functionality are completed they can be evaluated and integrated into the overall project. Without integration testing you are limited to testing a completely assembled product or system which is inefficient and error prone. Much better to test the building blocks as you go and build your project from the ground up in a series of controlled steps.

System Testing

System testing represents the overall test on an assembled software product. Systems testing is particularly important because it is only at this stage that the full complexity of the product is present. The focus in systems testing is typically to ensure that the product responds correctly to all possible input conditions and (importantly) the product handles exceptions in a controlled and acceptable fashion. System testing is often the most formal stage of testing and more structured.

3.6.Installation testing

This sub-section was written by Dr. Michael Brown of UMGC.

Installation testing is just like it sounds; it is testing of the installation of the software. This is normally done with a script or installation program.

One good example of installation problems was with iPlanet Application Server 6.0. In 1999 the Java community release a certification suite for Java application servers. iPlanet rushed to be the first certified application server and they were successful. But to make it quickly to market, the installation program asked the user for each installation parameter and gave them all possible choices. However some combinations of choices were invalid and produced an installation that did not run. Good installation programs will evaluate user parameter choices and only show them valid choices for future prompts. The problem was that most user didn't know the correct combination of choices and produced installations that did not run.

To make matters worse, the uninstall didn't remove the registry entries. So, if you installed it using invalid parameter and then uninstalled you could not install it correctly. You would have to manually delete the registry entries, which wasn't documented.

3.7.Regression testing

You must retest fixes to ensure that issues have been resolved before development can progress. So, *retesting* is the act of repeating the test that found a defect to verify that it has been correctly fixed.

Regression testing on the other hand is the act of repeating other tests in 'parallel' areas to ensure that the applied fix or a change of code has not introduced other errors or unexpected behaviour.

For example, if an error is detected in a particular file handling routine then it might be corrected by a simple change of code. If that code, however, is utilised in a number of different places throughout the software, the effects of such a change could be difficult to anticipate. What appears to be a minor detail could affect a separate module of code elsewhere in the program.

A bug fix could in fact be introducing bugs elsewhere.

You would be surprised to learn how common this actually is. In empirical studies it has been estimated that up to 50% of bug fixes actually introduce additional errors in the code. Given this, it's a wonder that any software project makes its delivery on time.

Better QA processes will reduce this ratio but will never eliminate it. Programmers risk introducing casual errors every time they place their hands on the keyboard. An inadvertent slip of a key that replaces a full stop with a comma might not be detected for weeks but could have serious repercussions.

Regression testing attempts to mitigate this problem by assessing the 'area of impact' affected by a change or a bug fix to see if it has unintended consequences. It verifies known good behaviour after a change. Regression testing is particularly suited to automated testing.

3.8.Examples

This sub-section was written by Dr. Michael Brown of UMGC.

Now, let's look at some examples and determine which categories they fall into.

Example 1

I am working on building a simple calculator. All of the parts are finished and the application is compiled. To test that it works correctly, I am going to try the test $1 + 1 =$. I expect the result to be 2.

- Since I am executing the code that makes it Dynamic, not Static.
- Addition is a function or feature of the software, so that makes it Functional.
- I didn't mention about looking at the code, so it is Black-box, not White-box.
- I have not even looked at the source code, so it cannot be Structural.
- Even though a calculator is a small piece of code this is System.
- I didn't have working code and then made a change to it. It is not Regression.

So, my test is a Dynamic, Functional, Black-box, System.

Example 2

I have a utility function in my software called $fun1(x)$. This function is used throughout my program and it is very important that it works correctly. In the body of the function is a condition. If $(x > 5)$. I set up a small program that runs tests against this function. One of the tests checks the output when x is 4, 5 and 6 to make sure that this condition is working as expected.

- Since I am executing the code that makes it Dynamic, not Static.
- This function is not a direct feature of the application, so it is not Functional.

- I read the code to come up with these tests, so it is White-box, not Black-box.
- Because I am testing this condition that is part of the structure of the program, so it is Structural.
- I am only testing a single function with these tests, so this is Unit.
- I didn't have working code and then made a change to it. It is not Regression.

So, my three tests are Dynamic, White-box, Structural, Unit.

Example 3

My software development team always has this problem. We write code that gets data from a database, but we forget to put indexes on the fields that we are searching by. Our program runs fine in our small test environment, but when we go live with the large production database it runs slowly.

So, when we are finished coding I have a developer sit and read through the code. For every call to the database to get data he confirms that there is an index on the search fields. If there isn't one, he adds it to the release.

- The code is not being executed so it is Static, not Dynamic.
- This function is not a direct feature of the application, so it is not Functional.
- The programmer is reading the code, so it is White-box, not Black-box.
- It is not Structural Testing because this does not relate to the structure of the program.
- I probably wouldn't even say that it is Unit, Integration or System Testing.
- I didn't have working code and then made a change to it. It is not Regression.

So, this activity is Static and White-box.

4. Common Testing Techniques

4.1.Code/Statement coverage

The following comes from What is White-Box testing? Techniques of White-box testing by Penna Sparrow.

In statement coverage each line in the module of the application is tested. It is used to check whether each of the statements in the module is covered or not. It checks if any of the code is not executed because of the blockage or any other reason. Here too redundant statements can be identified and eliminated.

4.2.Boundary test

There is a good short chapter on boundary testing (Chapter 5) in Software Testing: A Craftsman's Approach, Fourth Edition by Paul C Jorgenson, copyright 2014.

You should be able to access it through this link. If not, go to www.umgc.edu/library Click on E-Books, Click on What's in this collection, click on Safari Online.

<https://learning.oreilly.com/library/view/software-testing-4th/9781466560680/ch05.html>

4.3.Code inspect

Read section 7.5 in Software Reading Techniques: Twenty Techniques for More Effective Software Review and Inspection, by Yang-Ming Zhu, copyright 2016.

You should be able to access it through this link. If not, go to www.umgc.edu/library Click on E-Books, Click on What's in this collection, click on Books 24x7.

<https://library-books24x7-com.ezproxy.umgc.edu/assetviewer.aspx?bookid=125437&chunkid=141472919>

4.4.Code review

There is a good video explaining what a code review is. It was created by Glen Clarke. Copyright 2012.

You should be able to access it through this link. If not, go to <http://www.umgc.edu/library> Click on E-Books, Click on What's in this collection, click on Books 24x7.

<http://library.books24x7.com.ezproxy.umgc.edu/toc.aspx?bookid=77475>Examples

4.5.Examples

Let's look at some sample code. It is a division operator in a calculator. Let's assume that we want a test of test cases that will have 100% code coverage.

```

public double dev(double a, double b)
{
    if (b != 0.0)                // line 1
        double c = a / b;        // line 2
        System.out.println("Answer it " + c); // line 3
        return c;                // line 4
    else
        System.out.println("Div by 0."); // line 5
        return null;             // line 6
}

```

Here is one test case that we can create.

Test case name: 7 by 2

Method being tested: double dev(double a, double b)

Short Description: This test case will take 7 and divide it by 2.

Input Data to constructor and/or method you are testing: a = 7.0, b = 2.0.

Expected Results: 3.5

This test case will execute lines 1 through 4. We still do not have 100% coverage because of line 5 and 6. So, we add another test case to execute these lines.

Test case name: 2 by 0

Method being tested: double dev(double a, double b)

Short Description: This test case will take 2 and divide it by 0.

Input Data to constructor and/or method you are testing: a = 7.0, b = 0.0.

Expected Results: null

Between these two test cases all of the lines of the code are executed.

5. Unit Testing Automation

This sub-section was written by Dr. Michael Brown of UMGC.

5.1.Introduction to JUnit

Tests Cases and Test Suites are great, but it is time consuming to re-run them. This often forces us to write short Test Suites. But if we could automate the running of them, then we could build large Test Suites that are very thorough. JUnit is a tool that allows you to build sets of test cases.

- It can execute them.
- It keeps track of the test cases that fail.
- It is often built into many Java development environments.
- Other programming languages have similar tools.

Within your Java project you can create a JUnit Test file. Within the file you can have methods. Each method can run one or more tests. A test needs three things (1. the method in your application to call; 2. the expected value; 3. any input values for the method) The test will call the method with the input values and compare the actual value to the expected value. Then it reports the results. So, you can run the test many times.

You certainly can put all of your test cases into a single JUnit Test file. The only problem with that is that they all run or none of them run. There is no way to run just some of them. You can make multiple JUnit Test files. You can make a JUnit Test Suite that can execute multiple JUnit files. A JUnit Test Suite is another type of JUnit file. You can have multiple JUnit Test Suites. In very large programs it may take hours or even days to run all of the tests.

There are different ways that you can organize your automation of testing. Here are some examples. Let's assume that you have an application for a pharmacy. There are three submodules within the application: drug inventory, customer management and prescription processing.

You could organize testing according to the application functionality. You could create a JUnit Test File for each feature within each submodule. Then create three JUnit Test Suite Files, one for drug inventory, customer management and prescription processing.

Another way to do it is to organize according to types of tests. You could create two JUnit Test Files for each feature of the application: one for coverage tests and one for boundary tests. Then have two JUnit Test Suite Files. One could be for all coverage tests and one could be for all boundary tests.

Yet another way to organize them is how likely the test occurs in the application. For each feature of the application you could create a JUnit Test File for tests that are more likely to occur and a second for test that are less likely to occur. Then you could create two JUnit Test Suite Files for likely and less likely tests. When you do a major version release of your software you run both suites. When you do a minor version release you only run the ones that are likely to occur.

When a defect is found

When JUnit discovers a defect (the actual value and expected value do not match), one of two things has occurred. The obvious conclusion is that there is a defect in the code. But it is also possible that there is a defect in the test. JUnit Test Files are written code and they might have defects also.

Another important point to make is that when requirements change, you need to reanalyze your tests. If you have a version 1 of an application and 100 tests, then you get new requirements. You make the changes and rerun the 100 tests and get 3 defects. It is possible that you have introduced 3 new defects. But it is also possible that the expected values of the test need to be changed due to the new requirements. Do not always jump to the conclusion that there is a defect in the code.

5.2.Additional Readings on JUnit

One of the most common products to automate unit testing is JUnit. To get a good understanding of JUnit read Chapter 2 of Java Unit Testing with JUnit5: Test Driven Development with JUnit 5 by Shekhar Gulati and Rahul Sharma, copyright 2017.

You should be able to access it through this link. If not, go to www.umgc.edu/library Click on E-Books, Click on What's in this collection, click on Safari Online.

https://learning.oreilly.com/library/view/java-unit-testing/9781484230152/A429029_1_En_2_Chapter.html

5.3.JUnit Website

You can also checkout www.junit.org

6. User Interface Testing Automation

One of the most common products to automate user interface testing is Selenium. To get a good understanding Selenium read Chapter 1 of Selenium Essentials by Prashanth Sams, Copyright 2015.

You should be able to access it through this link. If not, go to www.umgc.edu/library Click on E-Books, Click on What's in this collection, click on Safari Online.

<https://learning.oreilly.com/library/view/selenium-essentials/9781784394332/ch01.html>

7. Human-Based Testing Techniques

7.1. Act Like A Customer

Read the Act-like-a-customer (ALAC) bullet in section 9.1.2 of Software Verification and Validation: A Practitioner's Guide by Steven R. Rakitin. You should be able to access it through this link. If not, go to <http://www.umgc.edu/library> Click on E-Books, Click on What's in this collection, click on Books 24x7.

<http://library.books24x7.com.ezproxy.umgc.edu/assetviewer.aspx?bookid=360&chunkid=404951175>

7.2. User Acceptance testing

Large scale software projects often have a final phase of testing called *Acceptance Testing*.

Acceptance testing forms an important and distinctly separate phase from previous testing efforts and its purpose is to ensure that the product meets minimum defined standards of quality prior to it being accepted by the client or customer.

This is when someone has to sign the cheque.

Often the client will have his end-users to conduct the testing to verify the software has been implemented to their satisfaction; this is called User Acceptance Testing or UAT. Often UAT tests extend to processes outside of the software itself to make sure the whole solution works as advertised in the context in which it is intended.

Other forms of acceptance testing include Factory Acceptance Testing (FAT) where the software is tested by a vendor to the client's specification before it leaves the factory or vendor's premises. Site Acceptance Testing is the next phases where the test is conducted at the customer's site (on their infrastructure in software terms). Some time around SAT, ownership of the software transfers to the customer, cheques are signed and the software becomes theirs (but subject to whatever support agreement is in place). Sometimes this is part of SAT, some time it is a separate step called 'commissioning' or similar.

While other forms of testing can be more 'free form', the acceptance tests should represent a planned series of tests and release procedures to ensure the output from the production phase reaches the enduser in an optimal state, as free of defects as is humanly possible. Often there are contractual treatments required for defects found during an acceptance test, so accuracy and detail are at a premium.

In theory acceptance testing should also be fast and relatively painless. Previous phases of testing will have eliminated any issues and this should be a formality. In immature or low quality software development, the acceptance test becomes the only trap for issues, back-loading the project with risk.

Acceptance testing also typically focusses on artefacts outside the software itself. A solution often has many elements outside of the software itself. These might include : manuals and documentation; process changes; training material; operational procedures; operational performance measures (SLA's).

These are typically not tested in previous phases of testing which focus on functional aspects of the software itself. But the correct delivery of these other elements is important for the success of the solution as a whole. They are typically not evaluated until the software is complete because they require a fully functional piece of software, with its new workflows and new data, to evaluate properly.

7.3.Alpha and Beta Testing

There are some commonly recognised milestones in the testing life cycle of a product.

Typically these milestones are known as “alpha”, “beta”. There is no precise definition for what constitutes alpha and beta test but the following are offered as common examples of what is meant by these terms :

Alpha Testing

Alpha: enough functionality has been reasonably completed to enable the first round of (end-to-end) system testing to commence. At this point the interface might not be complete and the system may have many bugs.

Beta Testing

Beta: the bulk of functionality and the interface has been completed and remaining work is aimed at improving performance, eliminating defects and completing cosmetic work. At this point many defects still remain but they are generally well understood.

Beta testing is often associated with the first end-user tests: the product is sent out to prospective customers who have registered their interest in participating in trials of the software. Beta testing, however, needs to be well organised and controlled otherwise feedback will be fragmentary and inconclusive. Care must also be taken to ensure that a properly prepared prototype is delivered to end-users, otherwise they will be disappointed and time will be wasted.

8. Static Analysis – FindBug

This chapter was written by Dr. Michael Brown and Dr. Mir Mohammed Assadullah of UMG.

As mentioned previously any technique that does not execute the software is static analysis. There are many easy cases that you can look at and find defects. For example, look at this example of pseudocode.

1. Print “Enter the numerator”
2. Put user input into variable x
3. Print “Enter the denominator”
4. Put user input into variable y
5. Print “The answer is”, x / y

Clearly this is a defect here. If y is 0, the program will crash. You cannot divide a number by 0. It is possible that we could have written and executed hundreds of test cases and never found this defect.

There are many static analysis tools; FindBug is one of the more popular ones for Java. It was developed at the University of Maryland. It will read your code and locate possible bugs. It is important to know that output of FindBug is not necessarily a bug. In some cases it is code that could lead to bugs in the future.

9. Formal methods (proofs)

Introduction

Let's say that we visit an engineer that just built a bridge. We ask him, "How do you know that your bridge works correct? Will it break? Will it fall over? Will it sink into the river?"



What if his response was, "I know that the bridge works correctly, because I had this guy, Bill, walk over it and then I had this lady, Mary, walk over it and it didn't fall down. Therefore, it must work correct!" What do you think of this answer? I am not sure that I have any faith that his bridge works.



What he will probably do to prove that his bridge works correctly, is to show you the mathematics behind the bridge.



So, why don't software engineers use mathematics to show that their programs work correctly? Actually, some do. If we can describe what software does using mathematics, then we could create mathematical proofs to prove that our software does what we claim it does.

All Engineers use Mathematics. It is Mathematics that proves why airplanes fly, airbags deploy, buildings don't fall over, spaceships orbit and hang-gliders glide.

And we as Software Engineers should use Mathematics also.

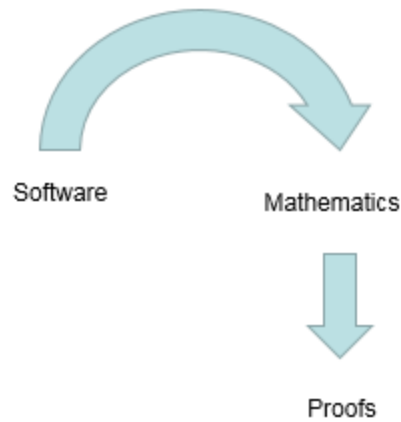
Terms:

Formal Methods: Formal Methods are actually precise techniques; tools support is provided for development of software as well as hardware systems. Mathematical techniques of formal methods enable developers to examine and prove models at any stage of the software development life-cycle.

Specification Language: Specification languages are formalisms to write specifications that capture properties required of a system (or better, of its behavior).

Proof: A mathematical proof.

So, we will take software and map it into Mathematics. Once the program is mapped into Mathematics we can create Mathematical proofs.



Background

When software runs there are many different things happens and states change with each line of code. But we write software for very specific objectives. Let's look at an example, written in pseudo code.

```
// Assume that there is an array called nums of numbers
int numTotal = 0
int numCount = 0
loop - get next element from nums, call it num, until array is empty
    numTotal = numTotal + num
    numCount = numCount + 1
end loop
float numAverage = numTotal / numCount
```

Maybe this code works correctly; maybe it doesn't. We don't know. But we know that when the code ends there are three new variables created: numTotal, numCount and numAverage. The code obviously averages a set of numbers; which value is stored in numAverage. This is the value that we are concerned with.

Maybe in the software requirements, we have a requirement like:

The software shall display the average number of runs scored per inning.

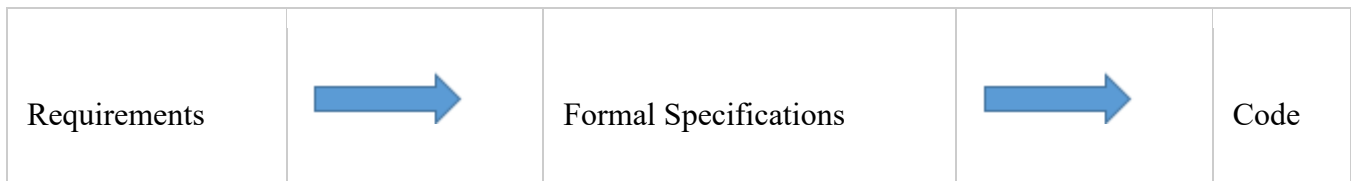
But this requirement is written in words, not Mathematics. So, we need a formal way to define requirements. This is a *Formal Specification Language*.

There are many Specification Languages: Java Modeling Language, Z (pronounced Zed) and Object-Z; RESOLVE, Eiffel, ANNA and others. For now, let's just talk about software in general Mathematics terms and not get tied down to a specific Specification Language.

Software projects that do not use Formal Methods will build code off of requirements.



In Formal Methods build code from specifications, which are built from requirements.



So, we know the requirements for this code, but what is the specification? How can we convey in Mathematics what this software needs to do?

First, let's look at the array `nums`. We don't have arrays in Mathematics, but we do have Strings. So what if we think of the array `nums` as a String of Integers `<nums1,nums2,nums3...numsx>`.

In Mathematical terms what we would like the code to do is:

$$numAverage = \left(\sum_{x=1}^{x=|nums|} nums_x \right) / |nums|$$

, where Σ is the summation and $//$ is the cardinality. So, if I can prove that the code fulfills this equation, then the code is correct. Let's not worry about the proofs for now.

Now, let's look at another example. Consider this requirement:

The software shall display the total number of runs scored in the game.

The code might look like this:

```
// Assume that there is an array called nums of numbers
int numTotal = 0
loop - get next element from nums, call it num, until array is empty
    numTotal = numTotal + num
end loop
```

Here we might see a specification like:

$$numTotal = \sum_{x=1}^{x=|nums|} nums_x$$

Pre and Post Conditions

Most formal methods have a notion of a pre-condition and post-condition for each section of code. A pre-condition is something that must be true for the software to work correctly. A post-condition is something that is true when the software finishes execution. Ideally, a proof will show that the post condition is achieved by the code.

Simple Proof

Let's look at the follow section of code written in Java.

```
I = I + j;  
J = I - j;  
I = I - j;
```

I claim that this section of code swaps the numbers between I and J . Whatever number is in I when it starts ends up in J and likewise, whatever number is in J ends up in I . Let's see if we can prove this.

Let's introduce a little notation. When you see the # in front of a variable, it will indicate that value of the variable when the code begins. No # indicates the value of a variable when the code ends. So, I would like to prove two things about this code.

$I = \#J$ and
 $J = \#I$

I want to prove that the final value of I is equal to the original value of $J(\#J)$. And I want to prove that the final value of J is equal to the original value of $I(\#I)$.

A second notation concerns all of the different values that a variable could have. Let's put a number after variable corresponding to the line of the proof. So, I_1 is the value of I after the first line of the proof.

Proofs are generally considered tables with a column for Line number, Proof statements and Required conditions. Proof statements are statements that we know to be true. Requires are the conditions which the Proof statements need to hold to be true.

Line	Proof	Requires
------	-------	----------

The first line of our proof represents the state before the first line of code is executed. At that point we only know that there are integers I and J .

Line	Proof	Requires
------	-------	----------

1	$I1 = \#I$ $J1 = \#J$	$\text{minInt} \leq \#I \leq \text{maxInt}$ and $\text{minInt} \leq \#J \leq \text{maxInt} \rightarrow$ $\text{minInt} \leq I1 \leq \text{maxInt}$ and $\text{minInt} \leq J1 \leq \text{maxInt}$
---	--------------------------	--

So, we begin with just $\#I$ and $\#J$. So, that implies that $I1=\#I$ and $J1=\#J$. If $\#I$ and $\#J$ are integers, then they must be between the minimum integer (minInt) and maximum integer (maxInt) values. This goes in the Requires section. Now if $\#I$ is between minInt and maxInt and $I1=\#I$, then $I1$ must be between minInt and maxInt. That is implied in the Requires section.

Line 2 of the proof represents that state of the program after the first line of code is executed. Here we see that $I2=I1+J1$. Through substitution, I can say that $I2=\#I+\#J$. Nothing happened to J , so $J2=J1$, which is the same as $\#J$.

Line	Proof	Requires
1	$I1=\#I$ $J1=\#J$	$\text{minInt} \leq \#I \leq \text{maxInt}$ and $\text{minInt} \leq \#J \leq \text{maxInt} \rightarrow$ $\text{minInt} \leq I1 \leq \text{maxInt}$ and $\text{minInt} \leq J1 \leq \text{maxInt}$
2	$I2=I1+J1 \rightarrow I2=\#I+\#J$ $J2=J1=\#J$	$\text{minInt} \leq I2 \leq \text{maxInt} \rightarrow$ $\text{minInt} \leq (\#I+\#J) \leq \text{maxInt}$

Now, let's look at the Requires side. If $I2$ is an integer, it must be between minInt and maxInt. If $I2$ is really the $\#I+\#J$, then $(\#I+\#J)$ must be between minInt and maxInt. If not, there would be an overflow error.

Line 3 of the proof reflects line 2 of the code. Variable I does not change, so $I3=I2$, which is $\#I+\#J$. Variable J changes however, $J3=I2-J2$. Through substitution I know that $I2$ is $\#I+\#J$ (line 2 of proof). So, I can make $J3=(\#I+\#J)-J2$. Line 2 of the proof also says that $J2=\#J$. So, with substitution again, I can replace $J2$ with $\#J$. Then I have $(\#I+\#J)-\#J$. Through Arithmetic I can make that $\#I$. Nothing changes in I , $I3=I2$.

Line 4 of the proof reflects line 3 of the code. In this line of code, $I = I - J$, we have $I_4 = I_3 - J_3$. We know from line 3 that $I_3 = \#I + \#J$. We also know from line 3 that $J_3 = \#I$. Arithmetic reduces this to line $I_4 = \#J$.

Line	Proof	Requires
1	$I_1 = \#I$ $J_1 = \#J$	$\text{minInt} \leq \#I \leq \text{maxInt}$ and $\text{minInt} \leq \#J \leq \text{maxInt} \rightarrow$ $\text{minInt} \leq I_1 \leq \text{maxInt}$ and $\text{minInt} \leq J_1 \leq \text{maxInt}$
2	$I_2 = I_1 + J_1 \rightarrow I_2 = \#I + \#J$ $J_2 = J_1 = \#J$	$\text{minInt} \leq I_2 \leq \text{maxInt} \rightarrow$ $\text{minInt} \leq (\#I + \#J) \leq \text{maxInt}$
3	$I_3 = I_2 = \#I + \#J$ $J_3 = I_2 - J_2 \rightarrow$ $J_3 = (\#I + \#J) - \#J \rightarrow$ $J_3 = (\#I + \#J) - \#J$ $J_3 = \#I$ $I_3 = I_2 = \#I + \#J$	$\text{minInt} \leq J_3 \leq \text{maxInt} \rightarrow$ $\text{minInt} \leq (\#I + \#J) - \#J \leq \text{maxInt}$
4	$I_4 = (I_3 - J_3) \rightarrow$ $I_4 = (\#I + \#J) - \#I \rightarrow$ $I_4 = \#J$ $J_4 = J_3 = \#I$	$\text{minInt} \leq J_4 \leq \text{maxInt} \rightarrow$ $\text{minInt} \leq (\#I + \#J) - \#I \leq \text{maxInt}$

Now our proof is finished. We see here that the final value for I is the original value of J ; $I_4 = \#J$. We also see that the final value for J is the original value for I ; $J_4 = \#I$. But this is only true if the most constrictive Requires clauses hold:

- $\text{minInt} \leq \#I \leq \text{maxInt}$
- $\text{minInt} \leq \#J \leq \text{maxInt}$
- $\text{minInt} \leq (\#I + \#J) \leq \text{maxInt}$

Now we have a very nice proof that shows our original assertion of $I = \#J$ and $J = \#I$, assuming the min and max constraints listed above.

Loops

Most programs have some type of loops in them. We can proof for code with loops by using Mathematical Induction. Let's look at this code. The person writing the code claims that the code produces the factorial of a number. (Note: in Mathematics a factorial of a number is all of the integer from 1 to that number multiplied together. We normally use the ! mark to indicate factorial. So the factorial of 6 is $6! = 6 * 5 * 4 * 3 * 2 * 1$).

```
Int result = 1;
For (int I = 1; I <= count; I++)
{
    result = result * I;
}
```

So the author of the code claims that the pre-condition is $I \geq 1$. If $I \geq 1$ then when the code is finished $\text{result} = \text{count}!$ (post-condition).

In Proof by Induction we start with a base case. Let's assume that count is 1.

Line	Proof	Requires
1	$\text{result}_1 = \# \text{result}$ $\text{count}_1 = \# \text{count}$	$\text{minInt} \leq \# \text{count} \leq \text{maxInt}$ and $\text{minInt} \leq \# \text{result} \leq \text{maxInt} \rightarrow$ $\text{minInt} \leq \text{result}_1 \leq \text{maxInt}$ and $\text{minInt} \leq \text{count}_1 \leq \text{maxInt}$
2	$\text{result}_2 = 1$	
3	$I_3 = 1$	$\text{minInt} \leq I_3 \leq \text{maxInt}$

	result3=result2	
4	result4=I3*result3=1*1=1	

The factorial of 1 is 1. That is correct.

Now, the next part of Mathematical Induction is to assume x is true and prove $x+1$ is true. I am going to skip the Requires section, just to speed things up.

Line	Proof	Requires
1	Assume that result1=I!	
2	$I2=I1+1$ result2=result1	
3	result3=result2*I2→ result3=result1*I2→ result3=I!*(I1+1)	

Formal Method Dispute

Wow. This seems like a lot of work to confirm the code works correctly. It takes almost a full-page proof to prove just a few lines of code. You are probably wondering if this is worth the effort. Well, you are not alone. There has been a debate over the usefulness of Formal Methods for decades. The debate can be summed up with quotes from two famous Software Engineers.

Edsger Dijkstra is a famous computer scientist. Information travels around networks using his algorithm. In 1970 in Notes On Structured Programming (EWD249), Dijkstra wrote:

When we take the position that it is not only the programmer's responsibility to produce a correct program but also to demonstrate its correctness in a convincing manner, then the above remarks have a profound influence on the programmer's activity: the object he has to produce must be usefully structured.

He also wrote in the same source

Program testing can be used to show the presence of bugs, but never to show their absence!

Clearly Dijkstra saw the need for formal correctness. But in the 1970's and 1980's computer scientists were eager to start writing code to take advantage of ever increasingly powerful computers. There was no time for formal Mathematical correctness.

The opposite point of view can be summarized by Larry Constantine, another famous Software Engineer, who wrote in his book Constantine on Peopleware in 1995.

This is one reason why academic computer scientists can waste entire careers on elegant mathematics and methods of formal proof that are hopelessly inadequate to everyday programming problems.

Constantine was very practical and favored working code, not formal proofs. It believed the formal proofs was a complete dead-end.

For a long time, Formal Methods were only used on a certain type of software, like for space ships and military uses.

And then ...

Then something exciting happened. Another area of Computer Science began developing and improving on Proof Generators. These programs would generate Mathematical proofs for you, given definitions and theorems. They were basic at first and then got better.

The technology is kind of simple. It basically explores all possible ways to constructs a proof and sees if one can reach the post-condition. If not, it concludes that the post-condition cannot be proven. As computers got faster, Proof Generators could handle ever larger proofs.

Around the mid-1990's Formal Method researchers began incorporating Proof Generators into their tools. Now the proofs are done for you. In fact, you may not even care what the proof looks like. You put in your pre-conditions and post-conditions and click the button. The Proof Generator tells you if it can prove the post-condition.

Now Formal Methods are still not used in a lot of programs, but they are gaining ground and there are some very good stories about them.

Formal Methods and Software

There are a number of Formal Methods that have tools for proof generation. Here is a list of some of the more common ones. Feel free to investigate some in more detail.

RESOLVE

B-Method

Java Modeling Language

SPARK Ada

Z (pronounced Zed)

Eiffel is an old programming language built on the concept for Formal Methods.

10. Symbolic execution

Introduction

Symbolic Execution is a verification technique developed in the mid-1970's by James King. In a traditional test case values are chosen for the different input values. An issue with traditional test cases is that they only test one set of values. In Symbolic Execution symbols are used for input values and can represent arbitrary values. The results of the test can then be checked against the program's requirements.

We already discussed how sometimes traditional testing using test scenarios and expected results are insufficient, or may even be an uneconomical way of testing. Some problems may have inputs that can be classified as ranges, logical variables, or otherwise translate into logical variables. Using traditional testing technique of test scenarios formed by explicit test variables may result in too many test cases. Using Model Checking may also result in too many states to explore, especially if one of the variables is continuous. Symbolic Execution offers another way of static testing that has found ready acceptance in the industry. It is also being used in finding security gaps in applications.

Symbolic Execution can be done using an Execution Tree or and Execution Table. This material uses Execution Tables but it is important to note that Execution Trees work just as well.

Advantages and Disadvantages

Symbolic Execution is really good at validating certain types of code and locating certain types of problems. It is really good at validating numerical calculations such as large equations. In some cases large Mathematical equations are implemented in numerous lines of code. Validation that the code correctly implements the equation is not trivial. Symbolic Execution would be a good choice to validate this code.

Symbolic Execution is also good for locating infeasible paths (paths that cannot be taken) and paths that are taken when they should not have been.

Like many other validation methods Symbolic Execution can be difficult to use if there are a large number of paths. This is often called the Path Explosion program. However, later we will talk about some automated tools that will perform Symbolic Execution that can help address this issue.

Execution Table with Specific Values

Let's consider a simple procedure below.

```
1 Sum: Procedure (A,B,C)
2   X = A+B;
3   Y = B+C;
4   Z = X+Y-B;
```

```

5 Return Z
6 End

```

To see how this program executes we can build an execution table. Execution tables have a row for each statement and a column for each variable and parameter. Each cell should contain the value of the variable or parameter after that corresponding statement is executed.

Let's complete the Execution Table for the example code, but let's use specific values. Let's assume that A is 1, B is 2 and C is 3. If we don't know what a value is, we often use question marks as holding places. So, we have question marks for X , Y and Z . So, after line 1 of the code, we have this table.

After Statement	X	Y	Z	A	B	C
1	?	?	?	1	2	3
2						
3						
4						
5						

Line 2 only changes one variable, X . It sets X to the value of $A+B$. So, we can now complete row 2 of the table. X after statement 2 has the value of 3.

After Statement	X	Y	Z	A	B	C
1	?	?	?	1	2	3

2	3					
3						
4						
5						

Line 3 of the program changes the variable Y , setting it to $B+C$. So, we can now complete line 3 of the table setting the value for Y to 5.

After Statement	X	Y	Z	A	B	C
1	?	?	?	1	2	3
2	3					
3		5				
4						
5						

Let's jump through the rest of the program completing values where appropriate.

After Statement	X	Y	Z	A	B	C
-----------------	---	---	---	---	---	---

1	?	?	?	1	2	3
2	3					
3		5				
4			6			
5			(Returns 6)			

We see from our example that the program returns 6. We might compare this result to our requirements and find that it is correct. 6 is the expected result.

Before we continue, why don't you try to create a simple execution table for some code given a set of input data. Then we will show you the correct answer. Consider this code. This code will switch the values of parameters *A* and *B*. Create the table for the set of values *A*=4 and *B*=2.

```

1 Swap: Procedure (A,B)
2   A = A + B;
3   B = A - B;
4   A = A - B;
5   Output A, B;
6 End

```

Here is the correct answer.

After Statement	A	B
1	4	2

2	6	
3		4
4	2	
5	Output(2)	Output(4)

Execution Table with Symbolic Values

As the name implies in Symbolic Execution we are going to use symbolic values to execute the program. Let's consider the code in our first example.

```

1 Sum: Procedure (A,B,C)
2   X = A+B;
3   Y = B+C;
4   Z = X+Y-B;
5 Return Z
6 End

```

We believe that this procedure will return the sum of A , B and C ($A+B+C$). In our previous Execution Table we had values $A=1$, $B=2$ and $C=3$ and the result was 6, which is correct. But that is only one test case. Let's produce the Execution Table again, but with symbolic values.

Here we set the values for A , B and C to be α_1 , α_2 and α_3 .

After Statement	X	Y	Z	A	B	C
-----------------	---	---	---	---	---	---

1	?	?	?	$\alpha 1$	$\alpha 2$	$\alpha 3$
2	$\alpha 1 + \alpha 2$					
3		$\alpha 2 + \alpha 3$				
4			$(\alpha 1 + \alpha 2) + (\alpha 2 + \alpha 3) - \alpha 2,$ reduces to $\alpha 1 + \alpha 2 + \alpha 3$			
5			Return $(\alpha 1 + \alpha 2 + \alpha 3)$			

When we get to line 4 of the table we complete the cell for row 4, column Z. We get that Z is $(\alpha 1 + \alpha 2) + (\alpha 2 + \alpha 3) - \alpha 2$. Using Algebra this can reduce to $\alpha 1 + \alpha 2 + \alpha 3$. Since A is $\alpha 1$, B is $\alpha 2$ and C is $\alpha 3$, we can conclude that the code returns the sum of parameters A, B and C.

Now, you take the Swap method that we defined earlier and create a Symbolic Execution table for it. We will then show you the correct answer.

```

1 Swap: Procedure (A,B)
2   A = A + B;
3   B = A - B;
4   A = A - B;
5   Output A, B;
6 End

```

The correct answer is:

After Statement	A	B

1	$\alpha 1$	$\alpha 2$
2	$\alpha 1 + \alpha 2$	
3		$(\alpha 1 + \alpha 2) - \alpha 2$, reduces to $\alpha 1$
4	$(\alpha 1 + \alpha 2) - \alpha 1$, reduces to $\alpha 2$	
5	Output($\alpha 2$)	Output($\alpha 1$)

Incorporating Conditions into Symbolic Execution

In the examples that we have looked at so far we have no conditions or loops. But there are few programs that don't have some type of condition or loop. To do Symbolic Execution on programs that have conditions and loops we need to add a new column to the table. The column is Path Condition, normally written pc. Sometimes for programs that do not have conditions or loops a pc column is added and the first cell just says "TRUE".

For instance, let's consider the following example. It should return the absolute value of the parameter x .

```

1 public static int abs(int x) {
2   if (x < 0) {
3     x = -x;  }
4   return x; }

```

So, let's create the Symbolic Execution Table. We begin by use $\alpha 1$ to symbolize all values for x . And the Path Condition, pc , is True, meaning that it is True for all cases.

After Statement	X	Pc
-----------------	---	----

1	$\alpha 1$	TRUE
---	------------	------

But when we get to line 2 of the code, the code branches. It might go to line 3 or it might go to line 4, depending upon the value of x . But we don't know the value of x ; we just have been using the symbol $\alpha 1$. So we need two cases; one for the condition being true and one for it being false.

At line 2 we need to fork. So, we begin with the Case of $x < 0$. Since this fork is only true when $x < 0$, then we need to indicate this in the pc column. Since x is equal to $\alpha 1$, we can enter $(\alpha 1 < 0)$. So, in line 3 we can say that x is $-\alpha 1$.

After Statement	X	Pc
1	$\alpha 1$	TRUE
2	Need for fork	
Case ($x < 0$)		$\alpha 1 < 0$
3	$-\alpha 1$	
4	Return ($-\alpha 1$)	

But now we have to handle the other case. What happens if the condition is false? We add a new line for the other case. We need to indicate this in the pc column. We use this symbol for *not* (\neg).

After Statement	X	Pc
-----------------	---	----

1	$\alpha 1$	TRUE
2	Need for fork	
Case ($x < 0$)		$\alpha 1 < 0$
3	$-\alpha 1$	
4	Return ($-\alpha 1$)	
Case $\neg (x < 0)$		$\neg(\alpha 1 < 0)$

Then we build out this case using the similar method.

After Statement	X	Pc
1	$\alpha 1$	TRUE
2	Need for fork	
Case ($x < 0$)		$\alpha 1 < 0$
3	$-\alpha 1$	$\alpha 1 < 0$

4	Return ($-\alpha 1$)	$\alpha 1 < 0$
Case $\neg(x < 0)$		$\neg(\alpha 1 < 0)$
4	Return ($\alpha 1$)	$\neg(\alpha 1 < 0)$

Now we can make some symbolic execution statements. Since $\alpha 1$ is the symbolic values for x , we can say:

1. When $x < 0$: returns $-x$
2. When $\neg(x < 0)$: returns x

Here is a small program. It is often used in industry to compute the percent change in sales. Companies want to know what percentage their sales are growing or shrinking for each product that they sell. But for new products the OldSalesAmt is 0 and you cannot do division by 0, so the sales grow is defined to be 0. You create the Symbolic Execution table and determine symbolic execution statements. Then we will give you the correct answer.

```

1 PercentSalesGrowth: Procedure (OldSalesAmt, NewSalesAmt)
2   Diff = NewSalesAmt - OldSalesAmt;
3   if (OldSalesAmt != 0)
4     PercentGrowth = Diff / OldSalesAmt;
5   else
6     PercentGrowth = 0;
7   Return PercentGrowth;
8 End

```

Here is the correct answer.

After Statement	Diff	PercentGrowth	OldSalesAmt	NewSalesAmt	pc
-----------------	------	---------------	-------------	-------------	----

1	?	?	$\alpha 1$	$\alpha 2$	True
2	$\alpha 2 - \alpha 1$				
3	Need for fork				
Case (OldSalesAmt!= 0)					(OldSalesAmt!= 0)
4		$(\alpha 2 - \alpha 1) / \alpha 1$			(OldSalesAmt!= 0)
7		Return (($\alpha 2 -$ $\alpha 1$)/ $\alpha 1$)			(OldSalesAmt!= 0)
Case \neg (OldSalesAmt!= 0)					\neg (OldSalesAmt! =0)
6		0			\neg (OldSalesAmt! =0)
		Return (0)			

The symbolic execution statements are:

1. When $(\text{OldSalesAmt} \neq 0)$: returns $(\text{NewsSalesAmt} - \text{OldSalesAmt}) / \text{OldSalesAmt}$
2. When $\neg(\text{OldSalesAmt} \neq 0)$: returns 0

Incorporating Loops into Symbolic Execution

Now let's talk about loops. Here is some code that will compute the Mathematical operation To-The-Power-Of, another words x^y . Notice that there is a loop between lines 4 and 7. We don't know how many times that loop will execute; it really depends upon the value of Y .

```

1  POWER: PROCEDURE (X, Y) ;
2      Z = 1;
3      J = 1;
4      WHILE (Y >= J)
5          Z = Z * X;
6          J = J + 1;
7      END;
8  RETURN (Z);
9  END;
```

Let's create an Execution Table assuming the value of X is 2 and Y is 3. 23 is 8. Using what we have seen so far, it is pretty easy to see how we get to this bottom of the first loop.

After Statement	Z	J	X	Y	Pc
1	?	?	2	3	True
2	1				
3		1			
4					

5	2				
6		2			

In the second loop, we have different values for Z and J . We can use the `//` notation to store these values.

After Statement	Z	J	X	Y	P_c
1	?	?	2	3	True
2	1				
3		1			
4					
5	$2 // 4 // 8$				
6		$2 // 3 // 4$			
7					
8		Return (8)			

We see that the code returns the correct answer.

Let's go through the code using Symbolic Execution.

After Statement	Z	J	X	Y	Pc
1	?	?	$\alpha 1$	$\alpha 2$	True
2	1				
3		1			
4					

When we get to line 4 there is a fork. It is possible that after line 4 executes, line 5 could execute or line 7/8 could execute. Another words, it loops again or it terminates the loop. The rule is to always evaluate the termination loop first.

After Statement	Z	J	X	Y	Pc
1	?	?	$\alpha 1$	$\alpha 2$	True
2	1				
3		1			
4	Need to fork				

Case $\neg(\alpha2 \geq 1)$					$\neg(\alpha2 \geq 1)$
7	1	1	$\alpha1$	$\alpha2$	$\neg(\alpha2 \geq 1)$
8	Return(1)				$\neg(\alpha2 \geq 1)$

But before I write my first symbolic execution statement, I am going to change the format of the path condition. If it is $\neg(\alpha2 \geq 1)$, then it must be that $\alpha2 < 1$. So the first symbolic execution statement is:

1. When $Y < 1$: returns 1

Now, let's create the other part of the fork.

After Statement	Z	J	X	Y	Pc
1	?	?	$\alpha1$	$\alpha2$	True
2	1				
3		1			
4	Need to fork				
Case $\neg(\alpha2 \geq 1)$					$\neg(\alpha2 \geq 1)$
7	1	1	$\alpha1$	$\alpha2$	$\neg(\alpha2 \geq 1)$

8	Return(1)				$\neg(\alpha_2 \geq 1)$
Case ($\alpha_2 \geq 1$)	1	1	α_1	α_2	$(\alpha_2 \geq 1)$
5	α_1				$(\alpha_2 \geq 1)$
6		2			$(\alpha_2 \geq 1)$
4	Need to fork				

We finish the body of the loop and it goes back to line 4 and we fork again. Since J is now 2 we want to compare α_2 to 2. The termination condition is α_2 is not greater than or equal to 2. Let's follow that fork first.

After Statement	Z	J	X	Y	Pc
1	?	?	α_1	α_2	True
2	1				
3		1			
4	Need to fork				

Case $\neg(\alpha 2 \geq 1)$					$\neg(\alpha 2 \geq 1)$
7	1	1	$\alpha 1$	$\alpha 2$	$\neg(\alpha 2 \geq 1)$
8	Return(1)				$\neg(\alpha 2 \geq 1)$
Case $(\alpha 2 \geq 1)$	1	1	$\alpha 1$	$\alpha 2$	$(\alpha 2 \geq 1)$
5	$\alpha 1$				$(\alpha 2 \geq 1)$
6		2			$(\alpha 2 \geq 1)$
4	Need to fork				
Case $\neg(\alpha 2 \geq 2)$					$(\alpha 2 \geq 1) \wedge \neg(\alpha 2 \geq 2)$
7	$\alpha 1$	2	$\alpha 1$	$\alpha 2$	$(\alpha 2 \geq 1) \wedge \neg(\alpha 2 \geq 2)$
8	Return($\alpha 1$)				$(\alpha 2 \geq 1) \wedge \neg(\alpha 2 \geq 2)$

This is a termination condition, so I can write a symbolic execution statement. Our path condition says: $(\alpha 2 \geq 1) \wedge \neg(\alpha 2 \geq 2)$ and $\neg(\alpha 2 \geq 2)$. The only way that $\alpha 2$ can be ≥ 1 but not ≥ 2 is if $\alpha 2$ is 1 (think about it). So I will create by symbolic execution statement to be:

2. When Y is 1: returns X

Next, we would follow the other fork of Case $(\alpha 2 \geq 2)$

After Statement	Z	J	X	Y	Pc
1	?	?	$\alpha 1$	$\alpha 2$	True
2	1				
3		1			
4	Need to fork				
Case $\neg(\alpha 2 \geq 1)$					$\neg(\alpha 2 \geq 1)$
7	1	1	$\alpha 1$	$\alpha 2$	$\neg(\alpha 2 \geq 1)$
8	Return(1)				$\neg(\alpha 2 \geq 1)$
Case $(\alpha 2 \geq 1)$	1	1	$\alpha 1$	$\alpha 2$	$(\alpha 2 \geq 1)$
5	$\alpha 1$				$(\alpha 2 \geq 1)$
6		2			$(\alpha 2 \geq 1)$
4	Need to fork				

Case $\neg(\alpha_2 \geq 2)$					$(\alpha_2 \geq 1) \wedge \neg(\alpha_2 \geq 2)$
7	α_1	2	α_1	α_2	$(\alpha_2 \geq 1) \wedge \neg(\alpha_2 \geq 2)$
8	Return(α_1)				
Case					$(\alpha_2 \geq 2)$

I can simplify my path condition for this case. It would be $(\alpha_2 \geq 1) \wedge (\alpha_2 \geq 2)$, in other words $(\alpha_2 \geq 1)$ and $(\alpha_2 \geq 2)$. But if α_2 is ≥ 1 and is also $\alpha_2 \geq 2$, we can just say $\alpha_2 \geq 2$.

We quickly see that this will continue forever. So, at some logical point we stop and look for the pattern.

Here is some code to produce the Mathematical operation Factorial. A Factorial of a number is all of the integers leading up to the number multiplied by each other. So, the Factorial of 3 is $3 * 2 * 1$ or 6.

```

1  FACT: PROCEDURE (X) ;
2      T = 1;
3      J = 1;
4      WHILE (J <= X)
5          T = T * J;
6          J = J + 1;
7      END;
8  RETURN (T);
9  END;
```

Try to create a Symbolic Execution Table and symbolic execution statements for this program. Try to create it for 3 execution of the loop body.

Software Used for Symbolic Execution

In industry there isn't really enough time to sit around and do Symbolic Execution by hand. Luckily there are a number of programs that preform the Symbolic Execution for us. You can investigate them

on your own for whatever programming language you are using. Also note, this might not be a complete list.

Programming Language	Tool(s)
Java	Java Path Finder (JPF), jCUTE, janala2, JBSE, KeY
C	Otter
Ruby	Rubyx
JavaScript	SymJS, Jalangi2
.NET Framework	Pex
Binary	Mayhem
PL/1	EFFIGY

11. Model checking

When testing software, software testers generally think about the intended functionality of the software and then create as many test scenarios as it takes to test all possible desired functionality. For instance, when testing a login page, a tester may check for:

1. Login page allowing to log in when a valid user name and password is entered
2. Login page rejecting to log in when an invalid user name and password is entered
3. Login page rejecting to log in when a valid user name and password is entered but the account has been administratively disabled and showing an appropriate message
4. Login page allowing to log in with an expired password but requiring the user to immediately change his or her password
5. Login page disabling the account after a predetermined number of attempts to log in

Where this list may be an appropriate set of test scenarios and expected results for a log in page, it says nothing about variability of results based on changes in inputs. For instance, it doesn't check for all possible character strings for user names and passwords and their combinations. Whoa! No one does that! That may be considered too much testing of a small component of a larger system. If you agree, you are not alone. This indeed is a lot of work to perform to test just the log in functionality of a larger system, especially if your team regularly performs regression testing every time the software is changed, possibly in some other part of the larger system.

But what if you write software to perform this exact exhaustive check for you? There are two issues with it:

1. You would have to customize the testing software for each type of functionality that you add to your system, meaning additional time and effort needed to write the automated tests
2. It would still take too much time to execute, if you try to do so in a brute force manner and create every possible string of characters for user names and passwords, meaning additional hardware resources would be needed

For most of the software written today, the two reasons above are enough to deter one from performing an exhaustive tests like this and consider the earlier list of test scenarios to be sufficient for the purpose. However, many systems that support human life have a valid reason to take on this challenge and go through the extra effort of validating software for every possible input. This task becomes increasingly complex when the software being tested has time or computer network dependencies and the execution paths it takes for different runs are not identical, even for the same inputs. In other words, even if the inputs are the same, just because when the software was executed, or because two other computers on the same network decided to communicate to each other during the time it was executed, the execution path, and thus the output, would be different for each run.

Model Checking can help in some of these cases. It would automatically go through the software intended to be tested, determine all the different execution paths that the software can possibly take, and

store it in its internal data structure. It is then ready to walk through its internal model of all execution paths and check.

But what would it check for? Most Model Checking software comes with some basic checks out of the box. For instance, we know that division by zero, dead lock, race conditions, etc. are all bad for normal execution of programs. All of these checks, and then some, are pre-packaged into modern Model Checking tools.

These checks are called ‘properties’ in the Model Checking world. It is relatively easy to create custom properties for a Model Checking software. These custom properties can check for business logic applicable to the system being checked, something that the Model Checking software developers could not have thought about when developing the Model Checking software. In other words, creating properties is independent of the internal workings of the Model Checking software. For instance, when testing software controlling a liquid fuel propelled rocket engine, a tester may want to ensure that fuel is pumped above a certain rate only when the engine is fired and no more than a certain amount total is pumped if it is not fired.

Generally, these properties are in Temporal Logic. Here are some examples of Temporal Logic operators, for some atomic propositions p and q :

- Fp — p holds some time in future
- Gp — p holds globally in the future
- Xp — p holds next time
- $p \cup q$ — p holds until q holds

An example of using these Temporal Logic operators may be:

The car doesn’t start until the gear is in ‘Park’.

Not start holds UNTIL gear state is ‘Park’

$(\sim \text{start}) \cup \text{gear_park}$

Recall that $p \cup q$ represents p holds until q holds.

Normally model checking is thought of as a graph. In the graph one hand side is generated directly from the program being analyzed. Each node represents a state within the program. Arrows between nodes represent state transitions from one state to another. In other words, as the variables in a program change, the state of that program changes and thus it transitions from one state to another. The execution flow or transitions from a state to another are controlled by branching or loop statements. This sort of graph is often referred to as *Finite State Automaton*.

Any Finite State Automaton can be converted from a graph form to a tree form as shown above. The starting state, here S_0 , becomes the root of the tree. Each child node of the current node is visited and as the graph is parsed, the associated tree is produced, generally in a *Breadth First* manner.

Microwave Example

Let's build a Finite State Automaton from a program. This software simulates a microwave oven. If the door is closed, it can be opened. If it is open, it can be closed. Likewise, if the heat is off, it can be turned on but only if the door is closed. If the heat is on, it can be turned off. This simple logic can be represented with the software below.

```
public class Microwave {
    boolean doorOpen = false;
    boolean heatOn = false;

    public Microwave() {
    }

    public void openDoor() {
        if (!doorOpen) {
            doorOpen = true;
        }
    }

    public void closeDoor() {
        if (doorOpen) {
            doorOpen = false;
        }
    }

    public void turnHeatOn() {
        if (!heatOn && !doorOpen) {
            heatOn = true;
        }
    }

    public void turnHeatOff() {
        if (heatOn) {
            heatOn = false;
        }
    }
}
```

Figure. A program simulating basic functions of a microwave oven

The program above allows open and close door functions as well as turning heat on and off functions. As a safety feature, before it turns the heat on, it checks to make sure the door is closed, otherwise it would not turn it on. Now, let's try to create Finite State Automata for the door functionality and the heat functionality.

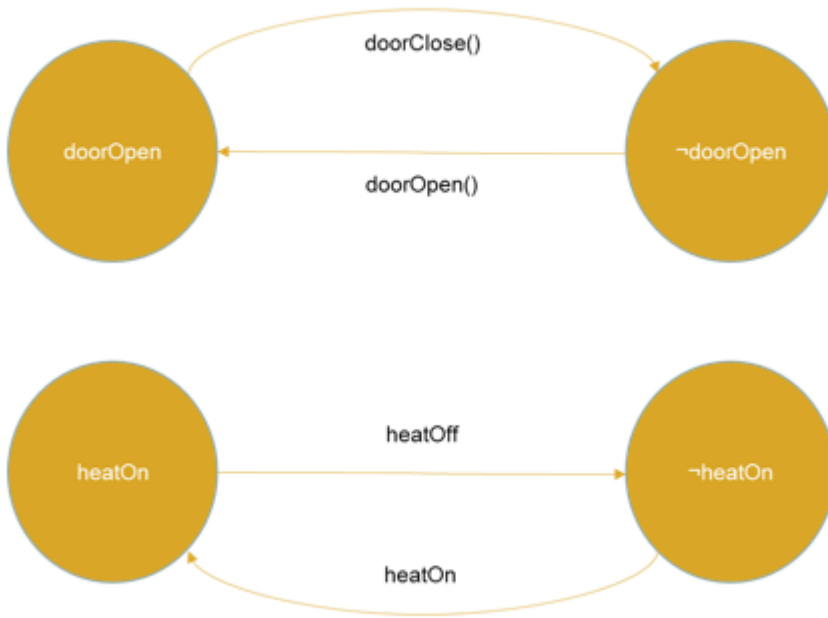


Figure. Finite State Automata of door and heat functions described in the code above

The diagram above describes how the program state, essentially stored in the member variables `doorOpen` and `heatOn` of the program above, can transition from one state to another using the methods or the arrows in the diagram above. Now, let's create a state diagram of both the variables together.

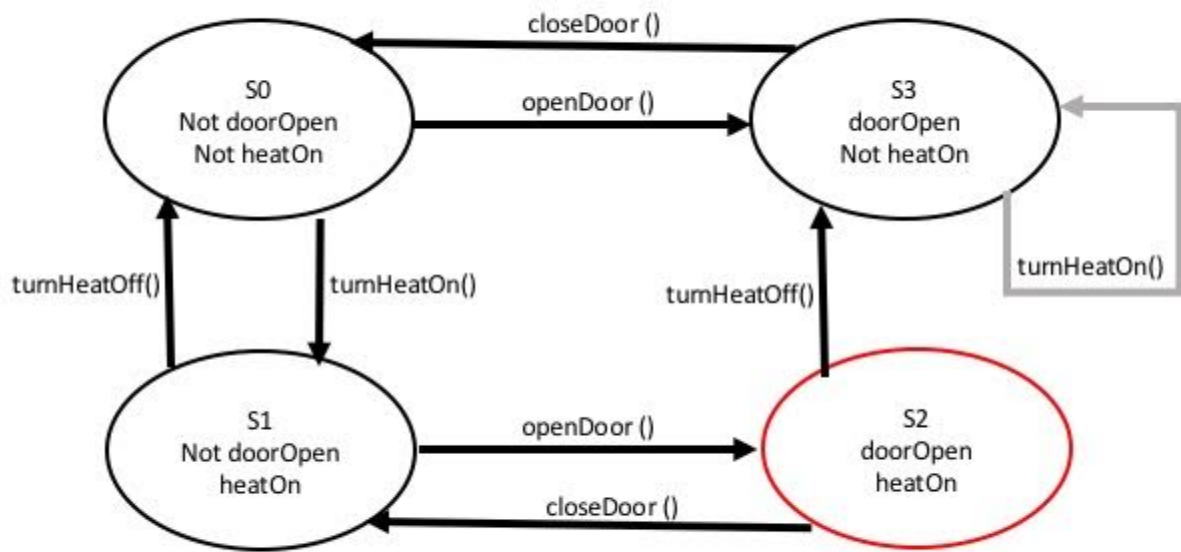


Figure. Finite State Automata of the microwave example in the code above, considering both variables to constitute state

In the diagram above, we can see the four states arising from two boolean variables. The state of the variables are written inside the state circles, along with the state name. The system may transition from one state to another by using the methods shown in Figure 2. The state where the door is open and the heat is on is S2. This state is a hazardous state and is represented by a red outline. Where it is possible to call the `turnHeatOn()` method from S3, it would not be able to transition the state to S2 because this method only transitions when the door is closed (e.g. transitioning from S0 to S1), but calling this method from S3 would have no effect on the state. This is represented by a grey arrow between S3 and S2.

However, there still remains a way for the system to arrive to the S2 state. They system can arrive to it by first getting into S1 and then calling `doorOpen()` method. In other words, by opening the door while the heat is already on. This path may not have been obvious to the reader when reading the code in Figure 2 and even when looking at the state transitions due to just one variable at a time. It only became obvious when the state diagram for all possible states was constructed.

Once a Model Checking has a model of the program being tested and a list of properties that need to be tested, it is ready to perform the checking. It would parse through each possible execution path and check for the validity of the listed properties. For each of the states, it checks all the properties to see if they hold. When all properties are checked, it moves on to the next state. When all states of an execution path are checked, it moves on to the next execution path. In this manner it goes through the entire *state space*, checking for all the properties.

If at any time it finds a property to be violated, it would report the violated property and, since it knows how it got to that state, it would also report the execution path that it took. This is very valuable for diagnostic purposes. The user may choose to stop execution of the Model Checking process at the first property violation and then tweak the model, or, continue with the process and report all violation in the entire state space.

Division by Zero Example

Let's go through an example. Consider the program below in JAVA.

```
import java.util.Random;

public class Rand {
    public static void main (String[] args) {
        Random random = new Random();           // (1)

        int a = random.nextInt(2);              // (2)
        System.out.println("a=" + a);

        //... lots of code here

        int b = random.nextInt(3);              // (3)
        System.out.println("  b=" + b);

        int c = a/(b+a -2);                     // (4)
        System.out.println("    c=" + c);
    }
}
```

Figure. A program that produces different results for each execution based on a Random Number Generator (RNG)

Here, in the main method, the program produces different results based on the JAVA built-in Random Number Generator (RNG) used on line marked 1. The `random.nextInt(2)` call on line marked 2 is requesting a random number between 0 and 1. Likewise, the `random.nextInt(3)` call on line marked 3 is requesting a random number between 0 and 2. On the line marked 4, possibly unbeknownst to us, we would be dividing by zero whenever $a + b$ is equal to 2. This program may execute several times and terminate normally before reporting a division by zero. If you replace the parameters passed to `random.nextInt()` on lines marked 2 and 3 by much larger numbers, say 2,000,000 and 3,000,000, the program may run normally for a lot more times before reporting a division by zero.

In this case, a traditional approach of providing a predetermined values for variables a and b would very likely give back expected results, but for some values of a and b , the program would terminate execution due to a division by zero. Let's look at how Model Checking would approach this problem in the figure below.

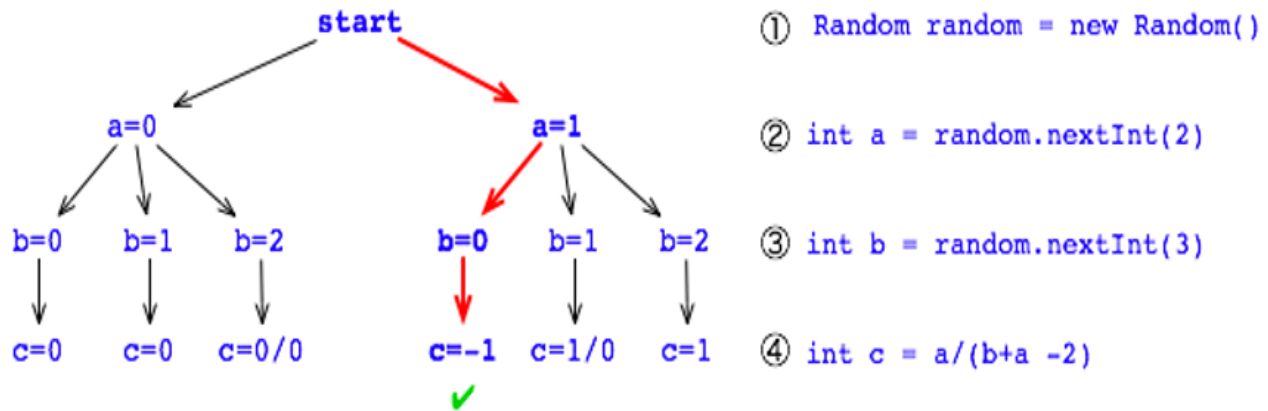


Figure 6 Red path showing a normally terminated execution path.
Source: https://babelfish.arc.nasa.gov/trac/jpf/wiki/intro/random_example

Model Checking would figure out all possible scenarios as shown above. The statements towards the right hand side are from the program and they correspond to the nodes at that level in the tree above. The red path is a possible path, perhaps one of the likely paths, which ends with a normal termination of the program. However, there are other paths that lead to division by zero. Model checking would perform a *Depth First Search* (DFS) on the tree above to simulate executing all states of all execution paths, as shown in the figure below.

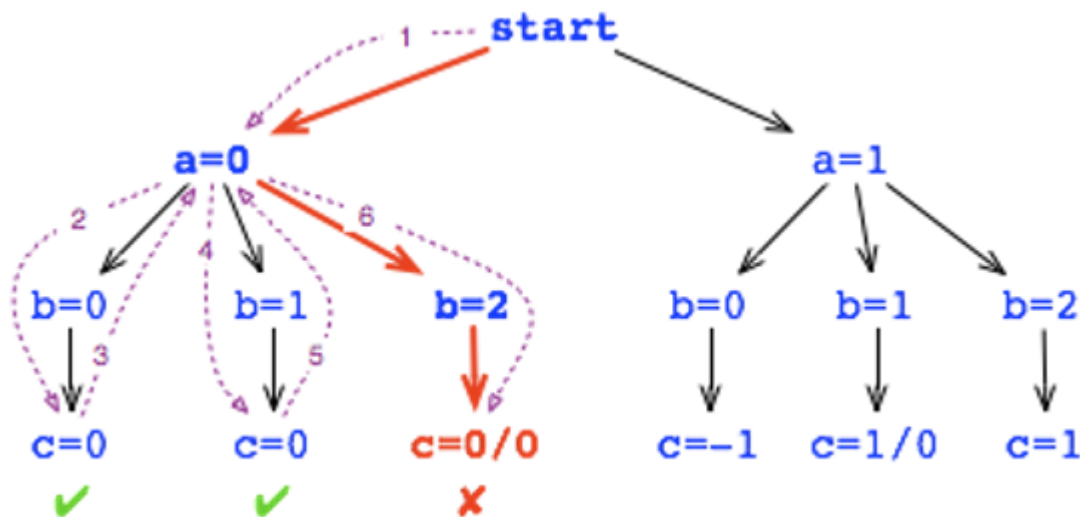


Figure 7 Systematic Depth First Search (DFS) of all possible execution paths. Source: https://babelfish.arc.nasa.gov/trac/jpf/wiki/intro/random_example

In the figure above, the purple dashed arrows show the way Model Checking would traverse the state tree. The green check marks at the leaves of the tree indicate a normal termination of the program. The red cross mark on the third leaf from the left represents the run-time exception thrown due to division by

zero and an abnormal termination of the program. In the figure above, the Model Checking process was stopped at the occurrence of the first property violation. There is another case, represented by the second leaf from the right, where division by zero also occurs.

At any state, represented by a node in the tree above, the path to that node from the root is known. This path comprises of all the state changes made from the beginning of the execution of that execution path until when the property was violated. This path, also known as a counterexample, is the diagnostic information that can be displayed for the modeler to tweak the model or for the tester to report the failure.

Software Used for Model Checking

In industry there are software tools that do Modeling Checking for you. Most are customized to a specific programming language. Feel free to investigate some for programming languages that you use.

Java - NASA's PathFinder, BANDERA, CBMC, DBRover, Alloy

C/C++ - BLAST, CBMC, PCAchecker, DBRover, DSVerifier, ESBMC, SATABS,

Ada - DBRover,

Ruby - Alloy

12. Throughout the SDLC

The following comes from Jenkins, N., 2017. A Software Testing Primer. *An Introduction to Software Testing v2*.

12.1. Verification and Validation

Sometime individuals lose sight of the end goal. They narrow their focus to the immediate phase of software development and lose sight of the bigger picture.

Verification

Verification tasks are designed to ensure that the product is internally consistent. They ensure that the product meets the specification, the specification meets the requirements... and so on. The majority of testing tasks are verification – with the final product being checked against some kind of reference to ensure the output is as expected.

For example, test plans can be written from requirements documents and from specifications. This verifies that the software delivers the requirements or meets the specifications. This however does not however address the ‘correctness’ of those documents!

Validation

Validation tasks are just as important as verification, but less common.

Validation is the use of external sources of reference to ensure that the internal design is valid, i.e. it meets user’s expectations. By using external references (such as the direct involvement of end-users) the test team can validate design decisions and ensure the project is heading in the correct direction (but sometimes all you have is validation, there is no ‘internal’ point of reference).

Examples

This sub-section – Examples of Verification and Validation - was written by Dr. Michael Brown of UMGC.

A good rule of thumb to use when determining if an activity is Verification or Validation, is that Verification is “how” the product is built. Validation activities relate to “what” product is built. Validation typically looks at the mapping between requirements and product. Verification looks at internal steps within the project.

Let’s assume that you are name a plant manager for Ford Motor Company. You are given a plant that produces F150 pickup trucks and Ford has a certain assembly line construction that they want you to follow. Upon starting your job, you change everything in the plant. Steps are done in different order; you change how the product moves throughout the factory. But you still are producing F150 pickup trucks. You would fail Verification, but pass Validation. Now let’s assume that you don’t change anything in the plant; it runs just like Ford wanted. But instead of making F150 pickup trucks, you make Ford Mustangs. So, you are building it the way Ford wants vehicles built. But you are not building what Ford wants built. You pass Verification, but fail Validation.

Here are some software examples.

Example	Type of Task
The Software Design Document says that there will be a BankAccount class that inherits from the Account class.	Confirming that this is true (ie, that the programmer built it according to the design document) in the code is a Verification task.
The Project Plan says that programmers will not start writing code until the Software Design Document has been signed off on. A Quality Assurance Engineer compares that date that the SDD was signed to the original date in the source code control repository.	This activity confirms that people are following the Project Plan. It is Verification.
A series of tests evaluate if the bank account deposits appear on the customer bank statement.	This matches the product built to what the customer wants. It is Validation.

12.2. Complexity of code

Many people underestimate the complexity of software. Software, they argue, is simply a set of instructions which a computer must follow to carry out a task. Software however can be unbelievably complex and as, Bruce Sterling puts it, 'protean' or changeable.

For example, the “Savings Account” application pictured at right is a simple application written in Visual Basic which performs simple calculations for an investment. To use the application, the user simply fills in three of four of the possible fields and clicks “Calculate”. The program then works out the fourth variable.

If we look at the input space alone we can work out its size and hence the number of tests required to achieve “complete” coverage or confidence it works.

Each field can accept a ten digit number. This means that for each field there are 10^{10} possible combinations of integer (without considering negative numbers). Since there are four input fields this means the total number of possible combinations is 10^{40} .

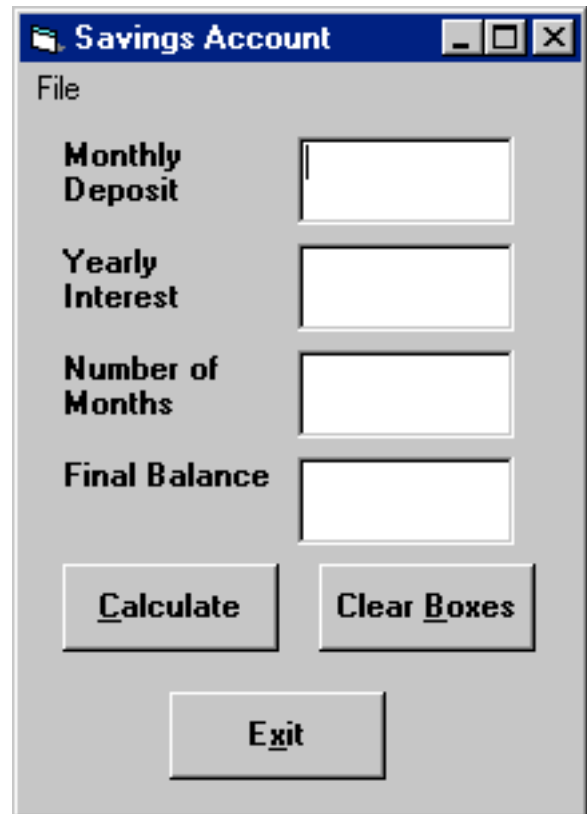
If we were to automate our testing such that we could execute 1,000 tests every second, it would take approximately 3.17×10^{29} years to complete testing. That's about ten billion, billion times the life of the universe.

An alternative would be to seek 100% confidence through 100% coverage of the code. This would mean making sure we execute each branch or line of code during testing. While this gives a much higher confidence for much less investment, it does not and will never provide a 100% confidence. Exercising a single pass of all the lines of code is inadequate since the code will often fail only for certain values of input or output. We need to therefore cover all possible inputs and once again we are back to the total input space and a project schedule which spans billions of years.

There are, of course, alternatives.

When considering the input space we can use “equivalence partitioning”. This is a logical process by which we can group “like” values and assume that by testing one we have tested them all. For example, in a numerical input I could group all positive numbers together and all negative numbers separately and reasonably assume that the software would treat them the same way. I would probably extend my classes to include large numbers, small numbers, fractional numbers and so on but at least I am vastly reducing the input set.

Note however that these decisions are made on the basis of our assumed knowledge of the software, hardware and environment and can be just as flawed as the decision a programmer makes when implementing the code. Woe betide the tester who assumes that $2^{64}-1$ is the same as 2^{64} and only makes one test against them!



The image shows a screenshot of a Windows application titled "Savings Account". The application has a menu bar with "File". Below the menu bar, there are four input fields with labels: "Monthly Deposit", "Yearly Interest", "Number of Months", and "Final Balance". At the bottom of the window, there are three buttons: "Calculate", "Clear Boxes", and "Exit".

Measuring Complexity

This subsection “Measure Complexity” comes from Rong Jiang. 2015. Research and Measurement of Software Complexity Based on Wuli, Shili, Renli (WSR) and Information Entropy. Entropy, 17, pp. 2094-2116.

American computer scientist Frederick Phillips Brooks, the winner of the Turing Award, known as Nobel Award in the field of computers, pointed out that complexity is one of the four essential issues of software and software project management in his paper *No Silver Bullet: Essence and Accidents of Software Engineering*. Famous computer expert, Grady Booch, one of the founders of Unified Modeling Language (UML), believed that excessive complexity was one of the main reasons for the failure of software projects. Narciso Cerpa had made a thorough investigation and research of 70 failed software projects in America, Australia, Chile and other countries and found that the complexity of 81.4% of the projects was underestimated.

The current research of software complexity is mainly involved in the fields:

1. Program complexity
2. Component complexity
3. Architecture complexity
4. Requirement complexity
5. Project complexity
6. Web complexity

Program complexity normally refers to an abstraction of the software scale, software development difficulty and the quantity of possible hidden errors in software. The measurement of program complexity has the earliest research history and is fruitful. It can be divided into process-oriented structured measurement of program complexity and object-oriented program complexity. Classical measurement methods are shown in the table below.

Table. Several classical measurements methods of program complexity.

Category	Name	Introduction
	Line of Code (LOC) Measurement	With the total number of LOC of a program as a measure of program complexity, it is generally used to estimate the workload for program development. Generally speaking, the program is increasingly complex with the increase in the number of LOC.
	Program Control Structure Complexity Measurement	This is a complexity measurement model based on the topological structure of a program, which measures the program complexity by calculating

Complexity Measurement of Structured Program	(namely the McCabe Measurement Method)	the number of linearly-independent directed cycles in the strongly-connected program chart.
	Program Text Complexity Measurement (namely the Halstead Measurement Method)	This measures the program complexity by calculating the number of operators in the program.
	Harrison Complexity Measurement	This is a measurement method of program complexity based on decomposing of the flow graph into ranks. It can determine the nesting levels of nodes in the flow chart.
Complexity of Measurement of Object-Oriented Programs	Chidamber & Kemerer (C&K) Measurement	This is a measurement method based on the inheritance tree and composed of six measurement standards: complexity of all methods of each class (WMC), depth of inheritance (DIT), number of children (NOC), number of responses of each class (RFC), coupling degree between objects (CBO) and lack of cohesion in method (LCOM).
	Metric for Object-Oriented Design (MOOD) Measurement	Six measurement indicators are given from four aspects of encapsulation, inheritance, coupling and polymorphism, namely method hiding factor (MHF), attribute hiding factor (AHF), methodinherited factor (MIF), attribute-inherited factor (AIF), coupling factor (CF) and polymorphism factor (PF).
	Chen and Liu Measurement	This measures from eight aspects of operation complexity, operation parameter complexity, attribute complexity, operation coupling factor, class inheritance, cohesion, class coupling and reusability.

In traditional research on the complexity of structured software, more attention is paid to the internal details of the module, but component software pays more attention to the interaction relationship between components. Some work proposes a measurement model of component software complexity based on the dependency matrix.

Too complex of an architecture may cause low comprehensibility and maintainability, but too simple of an architecture may cause reduced reliability and safety. Some work proposes an approach to evaluate the complexity of software architecture. Other work measures the visual complexity of the software architecture by the use of an algebraic expression. Some work studies software complexity metrics based on complex networks.

Requirement analysis is a very important link in the process of software development. If requirement analysis fails, all may fail. Based on requirement statement templates, there are quantitative measurement methods of complexity during the phase of software requirements.

The complexity of software projects is a very important indicator in software project management. Some methods propose evaluation models and methods for the complexity of software projects based on evidence reasoning.

Some work proposes entropy-based complexity measures, WCOXIN and WCOXOUT, for web applications. Other work proposes a complexity measure for web application, WCOX, which is defined by entropy theory, and a web model extracted by static analysis.

13. Defect management

13.1. Defect Report

Defects need to be handled in a methodical and systematic fashion.

There's no point in finding a defect if it's not going to be fixed. There's no point getting it fixed if it you don't know it has been fixed and there's no point in releasing software if you don't know which defects have been fixed and which remain.

How will you know?

The answer is to have a defect tracking system.

The simplest can be a database or a spreadsheet. A better alternative is a dedicated system which enforce the rules and process of defect handling and makes reporting easier. Some of these systems are costly but there are many freely available variants.

Importance of Good Defect Reporting

Cem Kaner said it best - “the purpose of reporting a defect is to get it fixed.”

A badly written defect report wastes time and effort for many people. A concisely written, descriptive report results in the elimination of a bug in the easiest possible manner.

Also, for testers, defect reports represent the primary deliverable for their work. The quality of a tester's defect reports is a direct representation of the quality of their skills.

Defect Reports have a longevity that far exceeds their immediate use. They may be distributed beyond the immediate project team and passed on to various levels of management within different organisations. Developers and testers alike should be careful to always maintain a professional attitude when dealing with defect reports.

Characteristics of a Good Defect Report

- Objective – criticising someone else's work can be difficult. Care should be taken that defects are objective, non-judgemental and unemotional. e.g. don't say “your program crashed” say “the program crashed” and don't use words like “stupid” or “broken”.
- Specific – one report should be logged per defect and only one defect per report.
- Concise – each defect report should be simple and to-the-point. Defects should be reviewed and edited after being written to reduce unnecessary complexity.
- Reproducible – the single biggest reason for developers rejecting defects is because they can't reproduce them. As a minimum, a defect report must contain enough information to allow anyone to easily reproduce the issue.

- Explicit – defect reports should state information clearly or they should refer to a specific source where the information can be found. e.g. “click the button to continue” implies the reader knows which button to click, whereas “click the ‘Next’ button” explicitly states what they should do.
- Persuasive –the pinnacle of good defect reporting is the ability to champion defects by presenting them in a way which makes developers want to fix them.

Isolation and Generalisation

Isolation is the process of examining the causes of a defect.

While the exact root cause might not be determined it is important to try and separate the symptoms of the problem from the cause. Isolating a defect is generally done by reproducing it multiple times in different situations to get an understanding of how and when it occurs.

Generalisation is the process of understanding the broader impact of a defect.

Because developers reuse code elements throughout a program a defect present in one element of code can manifest itself in other areas. A defect that is discovered as a minor issue in one area of code might be a major issue in another area. Individuals logging defects should attempt to extrapolate where else an issue might occur so that a developer will consider the full context of the defect, not just a single isolated incident.

A defect report that is written without isolating and generalising it, is a half reported defect.

Severity

The importance of a defect is usually denoted as its “severity”.

There are many schemes for assigning defect severity – some complex, some simple.

Almost all feature “Severity-1” and “Severity-2” classifications which are commonly held to be defects serious enough to delay completion of the project. Normally a project cannot be completed with outstanding Severity-1 issues and only with limited Severity-2 issues.

Often problems occur with overly complex classification schemes. Developers and testers get into arguments about whether a defect is Sev-4 or Sev-5 and time is wasted.

I therefore tend to favour a simpler scheme.

Defects should be assessed in terms of impact and probability. Impact is a measure of the seriousness of the defect when it occurs and can be classed as “high” or “low” – high impact implies that the user cannot complete the task at hand, low impact implies there is a workaround or it is a cosmetic error.

Probability is a measure of how likely the defect is to occur and again is assigned either “Low” or “High”.

This removes the majority of debate in the assignment of severity.

Probability	High	Minor	Major
	Low	Trivial	Minor
		Low	High
		Impact	

Figure. Relative severity in defects

Status

Status represents the current stage of a defect in its life cycle or workflow.

Commonly used status flags are :

- New – a new defect has been raised by testers and is awaiting assignment to a developer for resolution
- Assigned – the defect has been assigned to a developer for resolution
- Rejected – the developer was unable to reproduce the defect and has rejected the defect report, returning it to the tester that raised it
- Fixed – the developer has fixed the defect and checked in the appropriate code
- Ready for test – the release manager has built the corrected code into a release and has passed that release to the tester for retesting
- Failed retest – the defect is still present in the corrected code and the defect is passed back to the developer
- Closed – the defect has been correctly fixed and the defect report may be closed, subsequent to review by a test lead.

The status flags above define a life cycle whereby a defect will progress from “New” through “Assigned” to (hopefully) “Fixed” and “Closed.”

Elements of a Defect Report

Item	Description
Title	<p>A unique, concise and descriptive title for a defect is vital. It will allow the defect to be easily identified and discussed.</p> <p>Good : “Closed” field in “Account Closure” screen accepts invalid date</p> <p>Bad : “Closed field busted”</p>
Severity	An assessment of the impact of the defect on the end user (see above).
Status	The current status of the defect (see above)
Initial Configuration	The state of the program before the actions in the “steps to reproduce” are to be followed. All too often this is omitted and the reader must guess or intuit the correct pre-requisites for reproducing the defect.
Software Configuration	The version and release of software-under-test as well as any relevant hardware or software platform details (e.g. WinXP vs Win95)
Steps to Reproduce	<p>An ordered series of steps to reproduce the defect</p> <p>Good :</p> <ol style="list-style-type: none">1. Enter “Account Closure” screen2. Enter an invalid date such as “54/01/07” in the “Closed” field3. Click “Okay” <p>Bad: If you enter an invalid date in the closed field it accepts it!</p>
Expected Behavior	<p>What was expected of the software, upon completion of the steps to reproduce.</p> <p>Good: The functional specification states that the “Closed” field should only accept valid dates in the format “dd/mm/yy”</p> <p>Bad: The field should accept proper dates.</p>

Actual Behavior	<p>What the software actually does when the steps to reproduce are followed.</p> <p>Good: Invalid dates (e.g. “54/01/07”) and dates in alternate formats (e.g. “07/01/54”) are accepted and no error message is generated.</p> <p>Bad: Instead it accepts any kind of date.</p>
Impact	<p>An assessment of the impact of the defect on the software-under-test. It is important to include something beyond the simple “severity” to allow readers to understand the context of the defect report.</p> <p>Good: An invalid date will cause the month-end “Account Closure” report to crash the mainframe and corrupt all affected customer records.</p> <p>Bad: This is serious dude!</p>
(Proposed solution)	<p>An optional item of information testers can supply is a proposed solution.</p> <p>Testers often have unique and detailed information of the products they test and suggesting a solution can save designers and developers a lot of time.</p>
Priority	<p>An optional field to allow development managers to assign relative priorities to defects of the same severity</p>
Root Cause	<p>An optional field allowing developers to assign a root cause to the defect. such as “inadequate requirements” or “coding error”</p>

13.2. Reporting and Metrics

Defect Reporting

At a basic level defect reports are very simple: number of defects by status and severity.

Something like the diagram to the right.

This shows the number of defects and their status in testing. As the project progresses you would expect to see the columns marching across the graph from left to right – moving from New to Open to Fixed to Closed.

More complicated defect reports are of course possible. For example you might want to have a report on defect ageing – how long have defects been at a certain status. This allows you to target defects which have not progressed, which have not changed status over a period of time. By looking at the average age of defects in each status you can predict how long a given number of defects will take to fix.

By far and away, the best defect report is the defect status trend as a graph. This shows the total number of defects by status over time (see below).

The great advantage of this graph is that it allows you to predict the future.

If you take the graph at September and plot a line down the curve of the 'fixed' defects – it cuts the x-axis after the end of December. That means that from as early as September, it was possible to predict that not all of the defects would be fixed by December, and the project would not finish on time.

Similarly by following the curve of the 'new' defects you can intuit something about the progress of your project.

If the curve peaks and then is flat, then you have a problem in development – the bugs aren't staying fixed. Either the developers are reintroducing bugs when they attempt to fix them, your code control is poor or there is some other fundamental issue.

Time to get out the microscope.

Metrics of Quality and Efficiency

The ultimate extension of data capture and analysis is the use of comparative metrics.

Metrics (theoretically) allow the performance of the development cycle as a whole to be measured. They inform business and process decisions and allow development teams to implement process improvements or tailor their development strategies.

Metrics are notoriously contentious however.

Providing single figures for complex processes like software development can over-simplify things. There may be entirely valid reasons for one software development having more defects than another. Finding a comparative measure is often difficult and focussing on a single metric without understanding the underlying complexity risks making ill informed interpretations.

Most metrics are focussed on measuring the performance of a process, an organisation or an individual. Far too often they are applied without any deep understanding of the metric, general statistics or organizational psychology. There has been a strong realisation in recent years that metrics should be useful for individuals as well. The use of personal metrics at all levels of software development allow individuals to tune their habits towards more effective behaviours.

Defect Injection Rate

If the purpose of developers is to produce code, then a measure of their effectiveness is how well that code works. The inverse of this is the more defects, the less effective the code.

One veteran quality metric that is often trotted out is “defects per thousand Lines Of Code” or “defects per KLOC” (also known as defect density). This is the total number of defects divided by the number of thousands of lines of code in the software under test.

The problem is that with each new programming paradigm, defects per KLOC becomes shaky. In older procedural languages the number of lines of code was reasonably proportional. With the introduction of object-oriented software development methodologies which reuse blocks of code, the measure becomes largely irrelevant. The number of lines of code in a procedural language like C or Pascal, bears no relationship to a new language like Java or .Net.

The replacement for “defects/KLOC” is “defects per developer hour” or “defect injection rate”.

By dividing the number of defects by the total hours spent in development you get a comparative measure of the quality of different software developments:

$$\text{Defect Injection Rate} = \frac{\text{Number of Defects created}}{\text{Developer hours}}$$

Note that this is not a measure of efficiency, only of quality. A developer who takes longer and is more careful will introduce less defects than one who is slapdash and rushed. But how long is long enough? If a developer only turns out one bug free piece of software a year, is that too long?

This cannot be used as measure of developer skill. Each problem in software is different and potentially each solution is different, so using this number as a performance indicator will only lead to problems.

Defect Discovery Rate

An obvious measure of testing effectiveness is how many defects are found – the more the better.

But this is not a comparative measure. You could measure the defects a particular phase of testing finds as a proportion of the total number of defects in the product. The higher the percentage the more effective the testing. But how many defects are in the product at any given time? If each phase introduces more defects this is a moving target.

And suppose that the developers write software that has little or no defects? This means you will find little or no defects. Does that mean your testing is ineffective? Probably not, there are simply less defects to find than in a poor software product.

Instead, you could measure the efficiency of individual testers.

In a 'script-heavy' environment notions of test efficiency are easy to gather. The number of test cases or scripts a tester prepares in an hour could be considered a measure of his or her productivity; or the total number executed during a day could be considered a measure of efficiency in test execution.

But is it really?

Consider a script-light or no-script environment. These testers don't script their cases so how can you measure their efficiency? Does this mean that they can't be efficient? I would argue they can. And what if the tests find no defects? Are they really efficient, no matter how many are written?

Let's return to the purpose of testing – to identify and remove defects in software.

This being the case, an efficient tester finds and removes defects more quickly than an inefficient one. The number of test cases is irrelevant. If they can remove as many defects without scripting, then the time spent scripting would be better used executing tests, not writing them.

So the time taken to remove a defect is a direct measure of the effectiveness of testing.

Measuring this for individual defects can be difficult. The total time involved in finding a defect may not be readily apparent unless individuals keep very close track of the time they spend on testing particular functions. In script-heavy environments you also have to factor in the time taken scripting for a particular defect, which is another complication.

But to provide a comparative measure for a particular test effort is easy – simply divide the total number of hours spent in testing by the total number of defects (remember to include preparation time):

$$\text{Defect Discovery Rate} = \frac{\text{Number of Defects found}}{\text{Tester hours}}$$

Note that you should only count defects that have been fixed in these equations.

New defects that haven't been validated are not defects. Defects which are rejected are also not defects. A defect which has been fixed, is definitely an error that need to be corrected. If you count the wrong numbers, you're going to draw the wrong conclusions.

14. Testing the test suite

14.1. Mutation Testing

This subsection “Mutation Testing” comes from Takawale, S. R. and Kadam, S. (2017). Survey on Object Oriented Mutation Testing. *International Journal of Science and Research* 6(5), pp. 605-610.

Software testing is a practical technique to efficiently detect errors in software systems. Mutation testing is fault based testing techniques which is used to measure effectiveness of test suits. Using mutation testing, efficiency of test suits is measured. Mutation testing technique can be used in order to estimate the fault coverage of test suits. The idea of mutation testing was introduced by Richard Lipton in 1978.

Mutant generation is the first step of mutation testing process. Mutant is a copy of original program containing one fault which is syntactically correct. These faults are introduced using pre defined set of faults called mutation operator. After mutant generation, next step is to execute test cases against mutants and compare output with original program's output. When test case produces different output on mutant then mutant is said to be Killed mutant otherwise mutant is said to be live mutant. If no test case can distinguish its output from original program's output then, that is said to be Equivalent mutant. It is not possible to kill an equivalent mutant, as it is semantically equivalent to original program. The mutation score can be calculated by ratio of killed and live mutant. Mutation score indicates how effective given test case or test suit is. Testers can regenerate test cases to kill the remaining alive mutants and to raise the mutation score because a test case set with higher mutation score is considered more effective.

Object oriented program and language that solves the problem and provide the solution for old problem.

Mutation testing is based on the assumption that a program will be well tested if a majority of simple faults are detected and removed. Simple faults are introduced into the program by creating a set of faulty versions, called mutants. These mutants are created from the original program by applying mutation operators, which describe syntactic changes to the programming language. Test cases are used to execute these mutants with the goal of causing each mutant to produce incorrect output. A test case that distinguishes the program from one or more mutants is considered to be effective at finding faults in the program. Mutation testing involves many executions of programs; thus cost has always been a serious issue. Many techniques for implementing mutation testing have proved to be too slow for practical adoption. This paper presents a design and results from an implementation of a mutation system that is based on a novel execution strategy that combines mutation schemata with byte code translation.

Example

This example was written by Dr. Michael Brown of UMGC.

I wrote a method to determine the distance between two points on a grid. The code looks like this.


```

/* This method will return the distance between two points (x1, y1)
 * and (x2, y2).
 */
float pointDistance(float x1, float y1, float x2, float y2)
{
    float xdiff = Math.abs(x1 - x2);
    float ydiff = Math.abs(y1 - y2);
    return Math.sqrt(Math.pow(xdiff, 2) + Math.pow(ydiff, 2));
}

```

The code looks good. But I want to make sure, so I create some test cases.

Test Name	Input	Expected Output
All zeros	(0, 0, 0, 0)	0
One unit straight line	(0, 0, 1, 0)	1
Diagonal	(1, 1, 2, 2)	1.41

So, now I have this small suite of 3 test cases. I execute them and they all pass. But as a Tester I am very suspicious. How do I know that this is a good test suite? Another words, how can I test that my test suite is good? The answer it Mutation Testing. Mutation Testing is a way to test a test suite.

First I need to copy the code, probably into different folders. In each copy of the code I mutate it; I make a change to the code.

Copy 1	Copy 2
<pre> float xdiff = Math.abs(x1 * x2); float ydiff = Math.abs(y1 - y2); </pre>	<pre> float xdiff = Math.abs(x1 - x2); float ydiff = Math.abs(y1 - y2); </pre>

<pre>return Math.sqrt(Math.pow(xdiff, 2) + Math.pow(ydiff, 2));</pre>	<pre>return Math.sqrt(Math.pow(xdiff, 2) - Math.pow(ydiff, 2));</pre>
---	---

In Copy 1 I changed line 1. The + was mutated into a *. In Copy 2 I changed the + in line 3 to a -. Now I run my test suite against these copies of the code. If my test suite is good, then at least one test case will fail for each copy of the code. If all of my test cases pass, then obviously my test suite is lacking. After all, it could have let a defect through. I should then append my test suite with additional test cases.