Key Differences

Aspect	Verification	Validation
Purpose	Are we building the software right?	Are we building the right software?
When it occurs	During development (static checks)	After development or during testing
Focus	Conformance to requirements/specs	Meeting user needs/expectations
Methods	Reviews, inspections, walkthroughs	Testing, user feedback, demos

Verification - does it pass all the tests (unit, integration, system tests)
Validation - does it meet the requirements (user acceptance tests)

Mistake - a human error that produces something incorrect
Error - the difference between the current state and the correct value/condition
Fault / Defect - a latent problem according to the specifications in the product that has not been discovered
Failure - the inability of a system to perform its function according to specification

The percentage likelihood a feature will be used is called an app's **operational profile**

Test Case: Inputs, controlled conditions, expected output

Black Box / Closed Box
- Ignores how a function/component is written
- Focuses solely on the outputs generated based upon particular inputs

White Box / Open Box
- Testing that specifically takes into account how a function/component is written

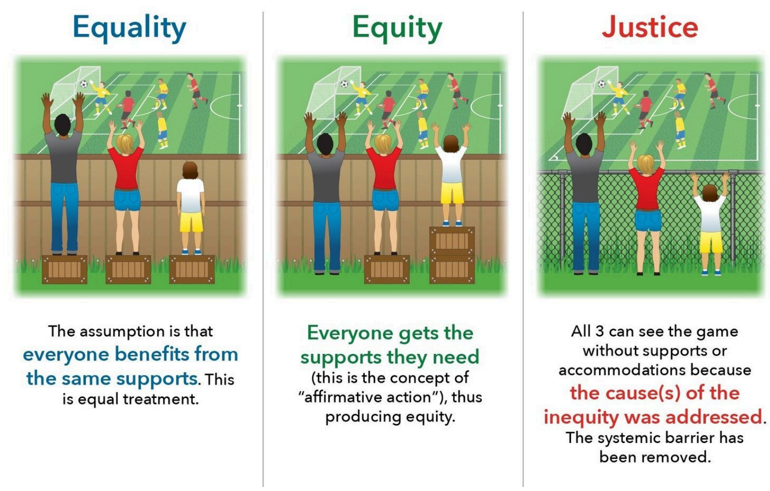
Gray Box / Combo Box
- Knowing that X was used to write the function, how might you test that particular component?

Unit - Do individual functions work?
Integration - Do components work when combined? Can be black, gray, or white box
System - Does the system as a whole work (function and non-functional)? Entirely black box
Acceptance - Does the system meet customer requirements? Entirely black box
Beta - What happens when non-developers test the system? Gray box for those that are familiar with the system, and black box for everyone else
Regression - Does old code work when we add features? Mainly black box; almost always automated

System testing: verify against technical specifications
Acceptance testing: verify against original business requirements

Processing an Ethical Scenario

- **Scoping**
    - Stakeholders
    - Issues, problems, risks
- **Analysis**
    - Responsibilities of decision-maker
    - Rights of stakeholders
    - Impact of actions on stakeholders
- **Determination**
    - Map to sections in codes of ethics
    - Classify each potential action as ethically obligatory, prohibited, or acceptable
    - Consider ethical merits of each option and select



Equality (minimize disparate treatment)

- Focus on relevant inputs, it gives everyone the same opportunity based on individual attributes

Equity (minimize disparate impact)

- Gives more resources to members of disadvantaged groups with the goal of achieving more equal outcomes

Justice (aspirational)

- Removes initial imbalance

Discrimination can be helpful

- Loan lending: Gender discrimination is illegal
- Medical diagnosis: Gender-specific diagnosis may be desirable
- Discrimination is a **domain-specific** concept!

Key fairness challenges in software:

- **Requirements**: identifying fairness concerns, analyzing potential harm, and design metrics

- **Design**: embedding fairness as a design criteria, including redundant checks and balances
- **Quality assurance**: evaluating how the entire system is fair and how this can be assured continuously in production
- **Process**: creating awareness for fairness concerns and document fairness results, across teams with different backgrounds and priorities

Things you must consider:
- Security is not a wrapper
- Security is not an add-on
- Security is not a module

Security must be built-in, pervasive in the system

CIA: Confidentiality, Integrity, Availability
- **Confidentiality**: Making sure data only viewable by authorized users
    - Authentication
    - Authorization
    - Encryption
    - Data at rest vs. data in transit
- **Integrity**: Ensuring accuracy and consistency of data over its entire lifecycle
    - Can we ensure that data is accurate and consistent over its lifetime?
- **Availability**: Data and services are available when needed, avoiding single points of failure, etc.
    - Can we ensure data and services are available when needed?
        - Avoid single point of failure
        - If something breaks, can we recover data etc?
    - Related issues:
        - Deployment, architecture

“White Hat” testing - “friendly” security and penetration testing done to expose vulnerabilities

Licensing

There are two definitions for freedom
- **Gratis** - having no price (free beer)
- **Libre** - lack of restrictions (free speech)

Copyleft: An arrangement where the software may be used, modified, and distributed freely on the condition that anything derived from it is bound by the same condition

Maintenance is “all the phases”
- To be good at maintenance, you need to be good at ALL of phases of development

Categories of Maintenance
- **Corrective** - fixing faults, no matter where they appear (code, docs, etc.)

- **Perfective** - changes to improve the system performance
- **Adaptive** - changes in the environment necessitate the change, not a request from the customer, per se
- **Preventive** - changes to avoid future problems

Lifespan of a system

- **Initial development**
    - First delivered version is produced
    - Knowledge about the system is fresh and constantly changing
    - An architecture emerge and stabilizes
- **Evolution (maintenance is part of it)**
    - Simple changes are easy, major changes are possible at higher cost/risk
    - Knowledge about system is good, but some original developers have left
    - For many systems, most of its lifespan is spent in this phase
- **Servicing**
    - The system is no longer a key asset
    - Effects of changes become harder to predict, only minor changes are made
    - Knowledge about the system has lessened
- **Phase out**
    - Decision to replace or eliminate the system
    - Exit strategy is devised and implemented, including wrapping and data migration
    - System is shut down

Technical debt

- "The extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution"

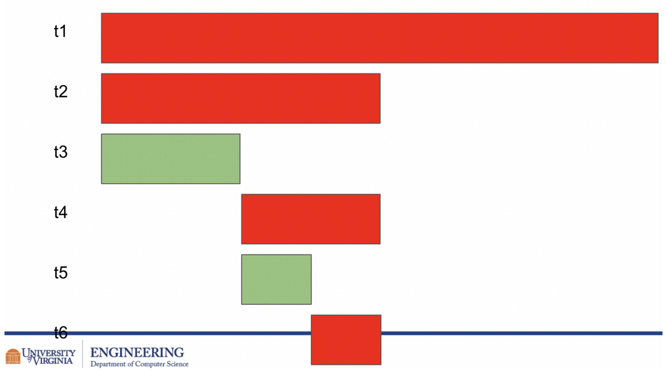
Failure Simplification: Why?

- **Ease debugging**
    - Smaller tests have smaller states / shorter execs
- **Ease communication**
    - Simpler test case is easier to communicate
- **Identify duplicates**
    - Simplified tests subsumes duplicates

Approach (**delta debugging**): iterative search

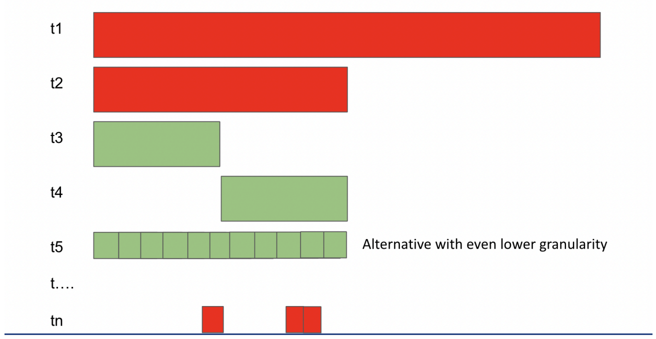
Repeat until "small-enough" input causes failure

1. Throw away portion of input
2. If the output is still wrong go back to 1
3. Else go back to previous state and discard other portion.



Isolating where the error is in the input

Approach: iterative n-ary search - adjust granularity



Sometimes need to adjust granularity

Spectrum-based Localization

- Insight
  - Executing faulty statement increases chances of failure
  - Not executing faulty line decreases chances of failure
- Estimation of fault location based on
  - Test results and coverage

Spectrum-based Localization

Jones et al. Tarantula

	Test Cases: 3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	suspiciousness	rank
mid() { int x,y,z,m;								
1: read("Enter 3 numbers:",x,y,z);	●	●	●	●	●	●	0.5	7
2: m = z;	●	●	●	●	●	●	0.5	7
3: if (y<z)	●	●	●	●	●	●	0.5	7
4: if (x<y)	●	●			●	●	0.63	3
5: m = y;		●					0.0	13
6: else if (x<z)	●				●	●	0.71	2
7: m = y; // *** bug ***	●					●	0.83	1
8: else			●	●			0.0	13
9: if (x>y)			●	●			0.0	13
10: m = y;			●				0.0	13
11: else if (x>z)				●			0.0	13
12: m = x;							0.0	13
13: print("Middle number is:",m);	●	●	●	●	●	●	0.5	7
} Pass/Fail Status	P	P	P	P	P	F		

- **Delta Debugging (Input)**
  - Simple algorithm renders significant simplification
  - Can require many tests -- so they better be fast
  - Needs domain tailoring (reasonable deltas)
  - Reducing test input
  - Relies on deterministic failures (not randomized stuff)
- **Spectrum-Based Localization (Code)**
  - Simpler/faster algorithms to detect suspicious statements
  - There may be many suspicious statements
  - Identifying faulty code regions

- X.Y.Z bxxxx
- X: major version, deals with architecture. When this changed, some fundamental change has occurred (moved to new platform, re-built a part of the program, completely changed UI, etc.). This number changes when something significant happens and you are not guaranteeing continuity between versions
- Y: minor version, this changes when there is perfective maintenance for the customer (same piece of SW which is why the X changes, but incrementing Y means this version of software is a much better version from Y - 1)
- Z: patch version, bug fix number. These usually change, but customer should see no change. Z number increase fixes bugs/updated functionality
- bxxxx: don't always see this, sometimes 'b' at the beginning other times not. Like the GitHub commit number. These don't go up incrementally to users, but do to developers (developers usually won't publish all 'commits')