Why Does Testing Matter?

- A video , tiltled “***Famous Failures***”**,** presents some famus people who failed at some point in their life, but they prove themselves later on.
- Although this so called failures were not due to errors, after being rejected by society some of these people continuously tried to improve themselves.
- What is the ***similarity*** between these ***failures*** and ***software testing***?
- By testing the software or product, it's quality improves, it becomes better.

Why Does Testing Matter?

- It is important to understand the consequences that some bugs might have.
- Ron Patton in his book titled "**Software Testing-second edition**", describes some famous cases of software errors or bugs and their consequences:
- **Disney's Lion King, 1994 –1995**
  - In the fall of 1994, the Disney company released its first multimedia CD-ROM game for children, ***The Lion King*** Animated Storybook.
  - Sales were huge. It was "the game to buy" for children that holiday season.
  - Soon the phone support technicians were swamped with calls from angry parents with crying children who couldn't get the software to work.

The problem?

Oregon State UNIVERSITY

Why Does Testing Matter?

- **Disney's Lion King, 1994 –1995**
  - **Problem:**
    - It turns out that **Disney** failed to properly test the softwa different PC models available on the market.
    - The software worked on a few systems—likely the ones that the **Disney** programmers used to create the game—but not on the most common systems that the general public had.

OSU
Oregon State
UNIVERSITY

Why Does Testing Matter?

- **NASA Mars Polar Lander, 1999**
  - On December 3, 1999, NASA's Mars Polar Lander disappeared during its landing attempt on the Martian surface.
  - In theory, the plan for landing was this:
    - As the lander fell to the surface, it was to deploy a parachute to slow its descent.
    - A few seconds after the chute deployed, the probe's three legs were to snap open and latch into position for landing.
    - When the probe was about 1,800 meters from the surface, it was to release the parachute and ignite its landing thrusters to gently lower it the remaining distance to the ground.
  - A Failure Review Board investigated the failure and discovered:
    - In most cases when the legs snapped open for landing, a mechanical vibration also tripped the touch-down switch, setting the fatal bit.
    - It's very probable that, thinking it had landed, the computer turned off the thrusters and the **Mars Polar Lander** smashed to pieces after falling 1,800 meters to the surface.

why the problem wasn't caught by internal tests!

Why Does Testing Matter?

- **NASA Mars Polar Lander, 1999**
  - Problem:
    - The lander was tested by multiple teams.
    - **One team** tested the **leg fold-down** procedure and another the landing process from that point on.
    - The first team never looked to see if the touch-down bit was set—it wasn't their area; the **second team** always **reset the computer**, clearing the bit, before it started its testing.
    - Both pieces **worked** perfectly **individually**, but **not** when put **together**.

Software Testing Myths

- **If we were really good at programming**, there would be no bugs to catch. People who claim that they write bug-free software probably haven't programmed much.
- **Complete Testing is Possible.** There might be some scenarios that are never executed by the test team or the client during the software development life cycle and may be executed once the project has been deployed.
- **A Tested Software is Bug-Free.** No one can claim with absolute certainty that a software application is 100% bug-free even if a tester with superb testing skills has tested the application.
- **Missed Defects are due to Testers.** It relates to Time, Cost, and Requirements changing Constraints.
- **Anyone can a Tester.** Thinking alternative scenarios, try to crash a software with the intent to explore potential bugs is not possible for the person who developed it. The developers are only responsible for the specific component or area that is assigned to them but testers understand the overall workings of the software, what the dependencies are, and the impacts of one module on another module.

Static Analysis vs Dynamic Analysis

- Before we get to testing (our first big main topic), I want to discuss another method for finding bugs which is "Static Analysis".
- Called "**static**" analysis because it analyzes program code for all possible run-time behaviors and seeks out coding flaws "bad patterns" without running it
  - Happens to some extent every time you build a program.
  - The compiler has to analyze the code to compile and optimize it.
    - e.g., the compiler warns you about some problems that might show up in testing, like some unidientified varialbes, potentially bad transformations of the type of something like casting an integer to another types.
- Analysis that runs the program is called "**dynamic**" analysis (testing is the most common dynamic analysis)
- Differs in a few key ways:
  - **Static** analysis can detect possible bugs in an early stage, and catch bugs without a test case – just by structure of code
  - **Dynamic** analysis (testing) involves test cases for execution
  - **Static analysis** can give "**false positives**" – warn you about a problem that can't actually show up when the program runs.
  - **Dynamic** analysis (testing) is about finding and fixing the defects.

Static Analysis: Not Just Compilers

- While the compiler does some limited “bug hunting” during compilation, that’s not its main job
  - There are dedicated tools for analyzing source code for bugs
  - A few such tools include:
    - **FindBugs** (open source tool that finds defects in Java programs)
    - Uno (open source, available on the web)
    - Coverity (paid software, quite pricey but very powerful, used by NASA and others)
    - Klocwork
    - CodeSonar

Static Analysis: Not Just Compilers

- Testing, on the other hand, requires more work from the programmer/test engineer
- So why not prefer static analysis in general?
  - Static analysis is generally limited to simple properties – don't reference null pointers, don't go outside array bounds
  - Also good for some security properties.
  - But very hard/impossible to check things like "**this sort routine really sorts things**"

Why Testing?

- Ideally: we *prove* code correct, using formal mathematical techniques (with a computer, not chalk).

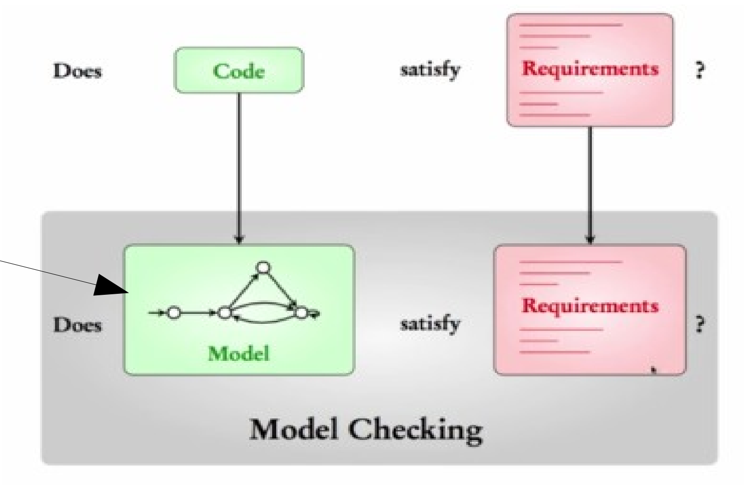
- Formal verification is the process of checking whether a design (an algorithm) satisfies some requirements (properties).

- Extremely difficult: for some trivial programs (100 lines) and many small (5K lines) programs

Model Checking

- Nearly ideally: use symbolic or abstract *model checking* to prove the system correct
  - Automatically extracts a mathematical abstraction from a system/program

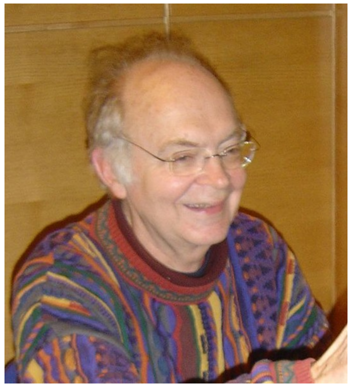
It uses finite state machines to model and verfiy a program



  - Proves properties over all possible executions
    - In practice, can work well for very simple properties (“this program never crashes in this particular way”), but can’t handle complex properties (“this is a working file system”)
    - Doesn’t work well for programs with complex data structures (like a file system)

As a last resort…

- … we can actually run the program, to see if it works
- This is software testing
  - Always necessary, even when you *can* prove correctness – because the proof is seldom directly tied to the actual code that runs



"Beware of bugs in the above code; I have only proved it correct, not tried it" – Knuth

Basic Definitions: Testing

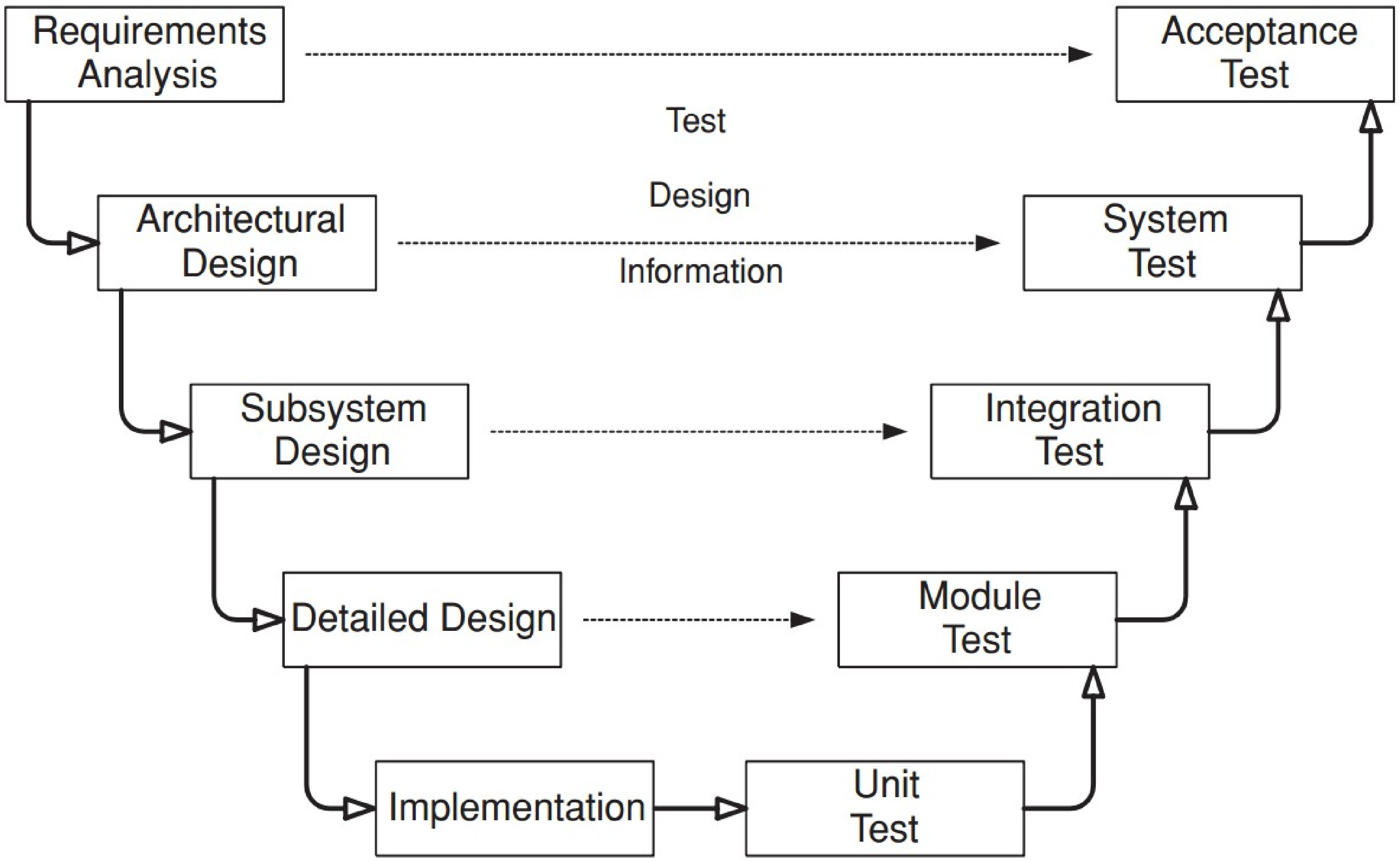
- What is software testing?
  - Running a program
  - in order to find faults
    - a.k.a. defects
    - a.k.a. errors
    - a.k.a. flaws
    - a.k.a. faults
    - a.k.a. **BUGS**

- Software testing is a critical element of software quality assurance.

- Contrary to life-cycle models, testing is an activity that must be carried out throughout the life-cycle.

Oregon State UNIVERSITY

Stages of testing



Software development activities and testing levels- the "V Model"

Stages of testing

1. **Unit testing** is designed to assess the units produced by the implementation phase and is the "lowest" level of testing.

2. **Module testing** is designed to assess individual modules in isolation, including how the component units interact with each other and their associated data structures.

3. **Integration testing** is designed to assess whether the interfaces between modules in a given subsystem have consistent assumptions and communicate correctly.

4. **System testing** is designed to determine whether the assembled system meets its specifications. It assumes that the pieces work individually, and asks if the system works as a whole.

5. **Acceptance testing** is designed to determine whether the completed software in fact meets these needs. In other words, acceptance testing is performed in a user environment and using real data.

Why is Testing Hard?

- Because the only way to be SURE a program has no bugs is to run all possible executions.

- We can’t do that

- Exhaustive testing is infeasible. The space of possible test cases is generally too big to cover exhaustively.

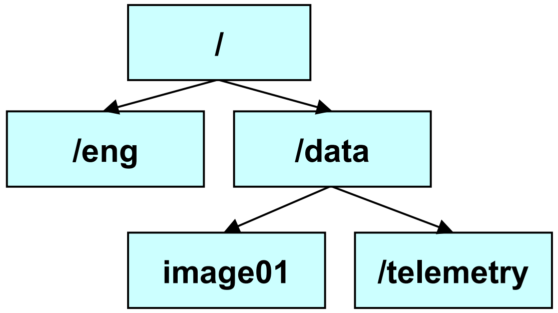
- One of the most important limitations of software testing is that testing can show only the presence of failures, not their absence.

Example: File System Testing

- File system is a library, called by other components of the flight software
- Accepts a fixed set of operations that manipulate files:

Operation	Result
mkdir ("/eng", …)	SUCCESS
mkdir ("/data", …)	SUCCESS
creat ("/data/image01", …)	SUCCESS
creat ("/eng/fsw/code", …)	ENOENT error
mkdir ("/data/telemetry", …)	SUCCESS
unlink ("/data/image01")	SUCCESS

File system



Example: File System Testing

- How hard would it be to just try “all” the possibilities?
- Consider only core 7 operations (mkdir, rmdir, creat, open, close, read, write)
  - Most of these take either a file name or a numeric argument, or both
  - Even for a “reasonable” (but not provably safe) limitation of the parameters, there are (266^{10}) executions of length 10 to try
  - Not a realistic possibility (unless we have (10^{12}) years to test)

Terminology

- **Fault:** A condition that causes a program to fail in performing its required function. A fault is a passive flaw.
  - What we usually think of as "**a BUG**"

- **Error:** A bad program state that results from a fault.
  - Not every fault always produces an error
    - If, for example, a programmer had meant to type char x[**10**]; for a local variable , and accidentally typed char x[**11**];
    - That is unlikely to cause an error unless the system is very short of memory. It is, however, a fault.

- **Failure:** An observable incorrect behavior of a program as a result of an error.

- **Debugging:** The activity by which faults are identified and rectified.

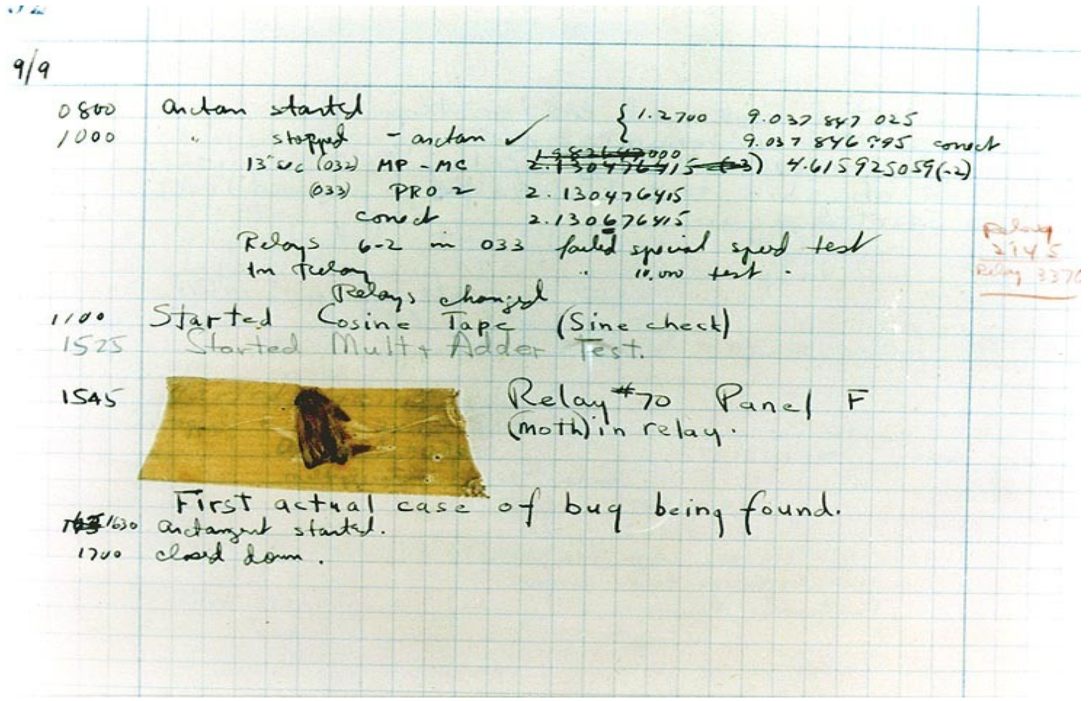
Terminology

- **Test (case)**: A *test case* is a set of inputs, execution condition, and a pass/fail criterion. In general, it is one execution of the program, that may expose a bug
- **Test suite**: A *test suite* is a set of test cases. In other words, it is a *set* of executions of a program, grouped together.

suite::tests   flock:sheep

The first “bug”

The term "bug" was used in an account by computer pioneer Grace Hopper. A typical version of the story is:
In 1946, when Hopper was released from active duty, she joined the Harvard Faculty at the Computation Laboratory where she continued her work on the Mark II and Mark III. Operators traced an error in the Mark II to a moth trapped in Really# 70 and Panel F, coining the term bug. This bug was carefully removed and taped to the log book. Stemming from the first bug, today we call errors or glitches in a program a bug.



9/9

0800 anctan started
1000 " stopped - anctan ✓ $\{$ 1.2700 9.037 847 025
9.037 846 995 conect
13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ ~~(-3)~~) 4.615925059(-2)
(033) PRO 2 2.130476415
conect 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 10,000 test .
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
~~1600~~ 1630 anctangent started.
1700 closed down.

Relay 2145
Relay 3370

https://en.wikipedia.org/wiki/Software_bug

Myths About Bugs

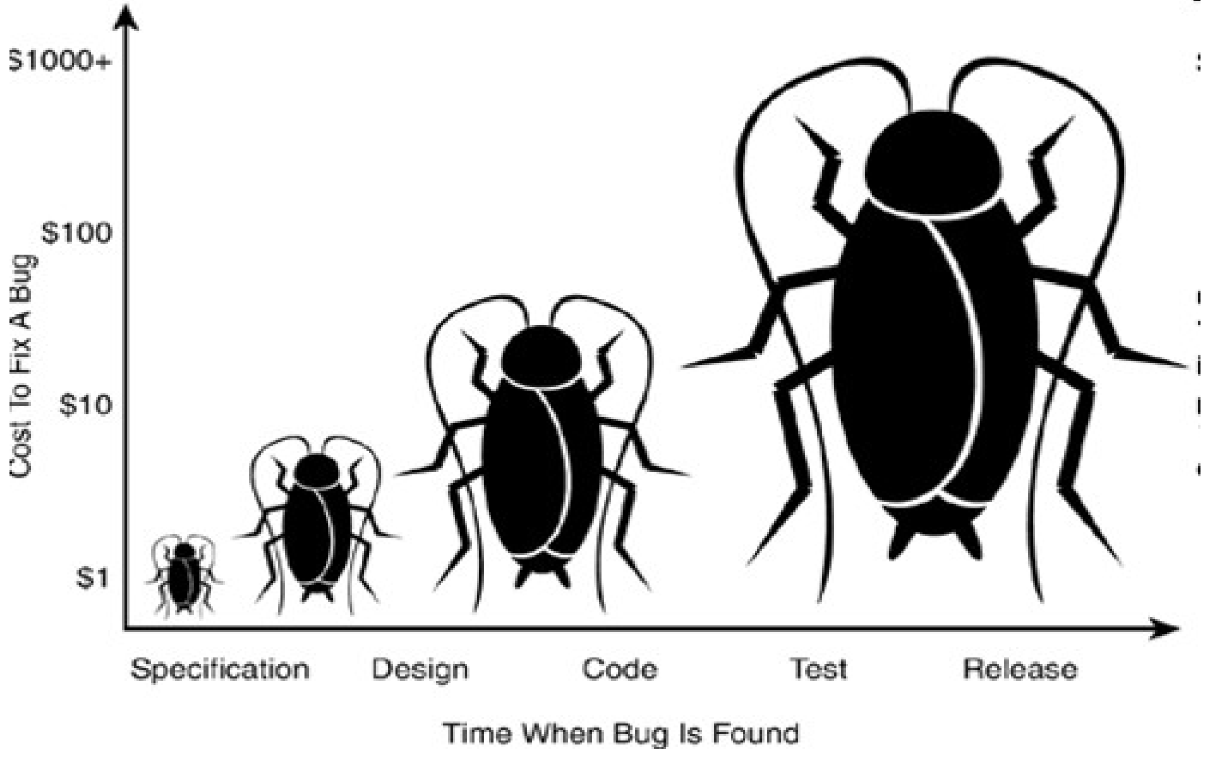
- **Bug Locality Hypothesis**: A bug discovered within a component affects only that component’s behavior.
- **Corrections Abide:** A corrected bug remains correct.
- **Silver Bullets**: A language, design method, environment (e.g. Structured Coding, Strong Typing, etc) grants immunity from bugs.
- **Control Bug Dominance:** The belief that most bugs are in the control structure (if, switches, etc) of programs.

Sources of Problems

- **Requirements Definition:** Erroneous, incomplete, inconsistent requirements.
- **Design:** Fundamental design flaws in the software.
- **Support Systems:** Poor programming languages, faulty compilers and debuggers, misleading development tools.
- **Inadequate Testing of Software:** Incomplete testing, poor verification, mistakes in debugging.
- **Evolution:** Sloppy maintenance, introduction of new flaws in attempts to fix old flaws.

The cost of bugs

- **The later the bug is found, the more expensive to fix:**



Consider the Disney Lion King case.

- If, in the early specification stage, someone had researched whats PCs were popular and specified that what needed to be designed and tested, the cost would have been minimal.
- The development team could have also sent a preliminary version of the software to a small group of customers, bet version. Those customers would have likely discovered the problem

To Expose a Fault with a Test (RIP) Model

- **Reachability**: the test much actually reach and execute the location of the **fault**
- **Infection**: the fault must actually corrupt the program state (produce an **error**)
- **Propagation**: the error must persist and cause an incorrect output – a **failure**

Which tests will achieve all three?

What is Testing?

- What is software testing?
  - Running a program
  - In order to find faults
  - But also, in order to
    - Increase our confidence that the program has high quality and low risk
    - Because we can never be sure we caught all bugs
  - How does a set of executions increase confidence?
    - Sometimes, by algorithmic argument
    - Sometimes by less formal arguments

References:

http://classes.engr.oregonstate.edu/eecs/summer2015/
Ammann, Paul, and Jeff Offutt. Introduction to software testing. Cambridge University Press, 2008.
www.cs.drexel.edu/~spiros/teaching/CS576/
https://www.st.cs.uni-saarland.de/edu/testingdebugging10
Patton, R. (2005). Software Testing (2nd ed.).
https://www.tutorialspoint.com/software_testing/software_testing_quick_guide.htm