What is testing?

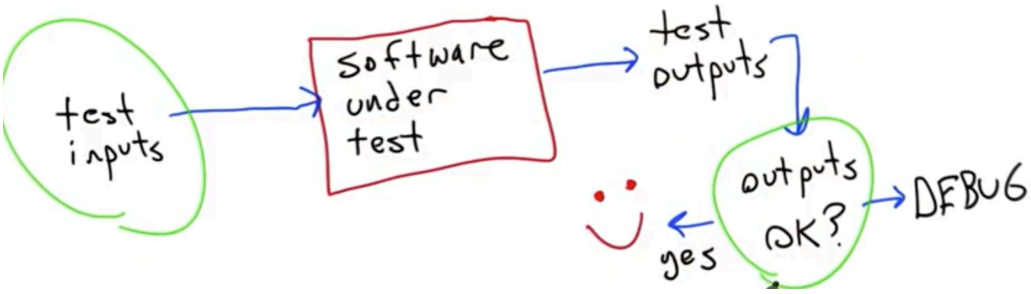
https://www.udacity.com/course/software-testing--cs258

1. Introduction

- In the big picture it's not necessarily failures that we're interested, but rather the fact that **if we can find failures in software and if we can fix these failures, then eventually we're going to run out of bugs and we'll have a software that actually works reliably**.
- Testing software is a large problem. We can see that:
  - Microsoft didn't manage to eliminate all the bugs in their products,
  - Google didn't manage to eliminate all the bugs in their services.
  - **How can we possibly get rid of all the bugs in our own software?**
- Testing problem is not really this large monolithic problem, but rather can be broken down into a lot of smaller sub-problems, and by looking at those sub-problems, **we can apply known techniques and things that people have done before**, and we could sort of pattern match on these problems, and once we've become good at these smaller problems, then we can become much better testers as a whole.

2. What is testing?

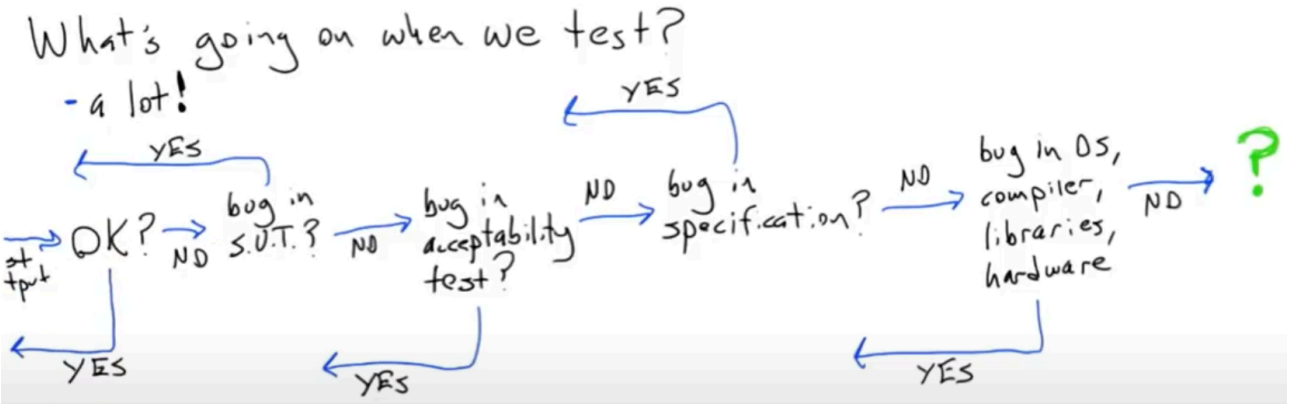
- It's always the case that we have some software under test (SUT).



- On the other hand, **selecting a good set of test inputs, and designing good acceptability test end up being actually really hard**, and basically, these are what we are going to be spending this course talking about.
- The goal of testing isn't so much as finding bugs, but rather it's finding bugs as early as possible. If our goal is just to find some bugs, we go ahead and give the software to our customers and let them find the bugs, but of course, there are huge cost associated with doing this.

- What we rather want to do is to move the time in which to find those bugs early. And the fundamental reason for that is, is that it's almost always the case that **the bug that's found earlier is cheaper to fix.**
- The second fact is that **it's possible to spend a lot of time and effort on testing and still do a really bad job.** Doing testing right requires some imagination and some good taste.
- Third, **more testing is not always better.** In fact, the quality of testing is all about the cost/benefit tradeoff. And fundamentally, testing is an economic activity. We're **spending money or** we're spending **effort** on testing in order to save ourselves money and effort later. Going along with this, testing methods should be evaluated about the cost per defect found.
- Fourth, **testing** can be made much **easier by designing software to be testable**.
- Fifth, **quality cannot be tested into software**. It is surprisingly easy to create software but it's impossible to test effectively at all.
- Finally, testing your own software is really hard.

3. What happens when we test software?



- We start off with the output produced by a test case. It's going to pass through the acceptability check.
- What we're doing when we check the output of a test case for acceptability is we're running a little experiment, and this experiment can have two possible results. One result is the output is okay. In which case, we are to go run another test case. And the question is, what do we learn in that case? And the answer is unfortunately not very much. What we might have done at best is increased our confidence just a tiny, tiny bit but the SUT is correct. And as it so happens, we stand to learn a whole lot more when the output is not okay. And the process I'm going to talk about right now is if the

acceptability check fails that is to say the test output is not okay, we have to discover what's happened. And so of course what we expect is much of the time we'll find a **bug in the SUT**. If that's the case, we're going to go fix it. And if not, there's still plenty of other possibilities: **a bug in** our **acceptability check,** or in our **specification**, in **OS, compiler, libraries, hardware**, etc.

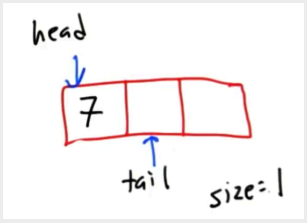
4. **Mars climate orbiter**

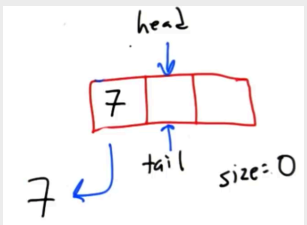
- **A famous bug** that happened to the Mars Climate Orbiter that was sent off to Mars in 1998, and there were **some miscommunications**, between NASA and the people they contracted out to which was Lockheed Martin. By the time Mars Climate Orbiter actually got to Mars which was in 1999, quite a while later, there had been some problems that caused the orbiter to drift off course enough that it basically ran into a suicide mission and crashed into the Martian atmosphere and broke up and crashed in the planet. What happened was **a basic unit error.**
- NASA expected units in metric, for example - meters per second, and Lockheed Martin programmed in English units, for example - feet per second.

5. **Fixed sized queue**

- enqueue
- dequeue
- FIFO order

```
# the Queue class provides a fized-size FIFO queue of
integers
# the constructor takes a single parameter: an integer >0
that
# is the maximum number of elements the queue can hold
# empty() returns True iff the queue holds no elements
# full() returns True iff the queue cannot hold any more
elements
# enqueue(i) attempts to put the integer i into the queue;
it returns
# True if successful and False if the queue is full
# dequeue() removes an integer from the queue and returns
it,
# or else returns None if the queue is empty


import array
import random

class Queue:
      def __init__(self,size_max):
            assert size_max > 0
            self.max = size_max
            self.head = 0
            self.tail = 0
            self.size = 0
            self.data = array.array('i', range(size_max))

      def empty(self):
            return self.size == 0

      def full(self):
            return self.size == self.max

      def enqueue(self, x):
            if self.size == self.max:
                  return False
            self.data[self.tail] = x
            self.size += 1
            self.tail += 1
            if self.tail == self.max:
                  self.tail = 0
            return True

      def dequeue(self):
            if self.size == 0:
                  return None
            x = self.data[self.head]
            self.size -= 1
            self.head += 1
            if self.head == self.max:
                  self.head = 0
            return x
```





6. Filling the queue

```
q = Queue(2)
c1 = enqueue(6)
c2 = enqueue(7)
c3 = enqueue(8)
c4 = dequeue()
c5 = dequeue()
c6 = dequeue()
```

The values in c1, c2, c3, c4, c5, c6 are:

True, True, False, 6, 7, None

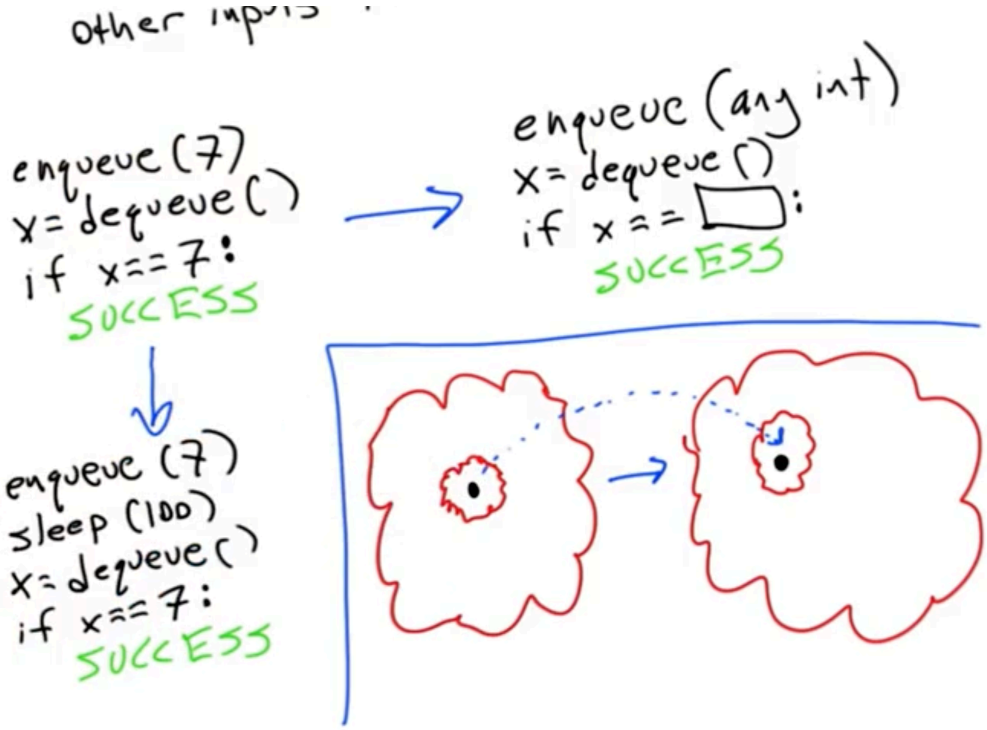
7. What we learn

```
enqueue(7)
x = dequeue()
if x == 7:
    print "Success!"
else:
    print "Error!"
```

If we pass this test case, what have we learned about our code?

- [x] Our code passes this test case.
- [ ] Our code passes any test case where we replace 7 with a different integer.
- [x] Our code passes many test cases where we replace 7 with a different integer.

8. Equivalent tests



- The question is given the above test case, is it possible to conclude without trying it that this test case will succeed?
- We're trying to make arguments that **a single test case is a representative of a whole class of actual executions of the system that we're testing**.
- The idea is we have a single test case that represents a point in the input space for the system that we're testing. By running the code, we mapped that point in the input space to a single point in the output space. But the problem is, there is of extremely large input space and we can't test it all.
- What we're trying to do here is build up an intuition for when we can make an argument that, in fact, we haven't made an argument about the mapping of a single input to a single output. But rather, **some class of inputs that are closely related, that are somehow equivalent for purposes of the SUT, and that if the system executes correctly for the single input that we've tried, it's going to execute correctly for all of the inputs in this particular bit of the input space**.

9. Testing the queue

- We can enqueue some numbers and dequeue them and check if they are correct.

- We can, also, write test functions that are not testing equivalent functionality in the queue but the if statements. For example, if the size is equal to max, then we will return false and if the tail is equal to max, then we will reset the tail to zero.
- Below is the code for 3 test functions for the fixed size queue mentioned before.

```
def test1():
    q = Queue(3)
    res = q.empty()
    if not res:
        print "test1 NOT OK"
        return
    res = q.enqueue(10)
    if not res:
        print "test1 NOT OK"
        return
    res = q.enqueue(11)
    if not res:
        print "test1 NOT OK"
        return
    x = q.dequeue()
    if x != 10:
        print "test1 NOT OK"
        return
    x = q.dequeue()
    if x != 11:
        print "test1 NOT OK"
        return
    res = q.empty()
    if not res:
        print "test1 NOT OK"
        return
    print "test1 OK"
```

```
def test2():
   q = Queue(2)
   res = q.empty()
   if not res:
      print "test2 NOT OK"
      return
   res = q.enqueue(1)
   if not res:
      print "test2 NOT OK"
      return
   res = q.enqueue(2)
   if not res:
      print "test2 NOT OK"
      return
   res = q.enqueue(3)
   if q.tail != 0:
      print "test2 NOT OK"
      return
print "test2 OK"

def test3():
   q = Queue(1)
   res = q.empty()
   if not res:
      print "test3 NOT OK"
      return
   x = q.dequeue()
      if not x is None:
      print "test3 NOT OK"
      return
   res = q.enqueue(1)
      if not res:
      print "test3 NOT OK"
      return
   x = q.dequeue()
      if x != 1 or q.head != 0:
      print "test3 NOT OK"
      return
print "test3 OK”

test1()
test2()
test3()
```

10. Creating testable software

- SUT:
  - clean code
  - refactor
  - should always be able to describe what a module does & how it interacts with other code
  - no extra threads
  - no swamp of global variables
  - no pointer soup
  - Modules should have unit tests
  - when applicable, support fault injection
  - assertions, assertions, assertions

11. Assertions

Assertion: Executable check for a property that must be true for your code

Rule 1: Assertions are not for error handling

```
def sqrt (arg):
      ... compute result ...
      assert result >= 0
      return result
```

Rule 2: NO SIDE EFFECTS

```
assert foo() == 0
where:
foo() changes a global variable
```

Rule 3: No silly assertions

```
assert (1 + 1) == 2
```

- The best assertions are those which check a nontrivial property that could be wrong but only if we actually made a mistake in our logic. It's not something that could be wrong if the user did something wrong, and it's not something that's wrong that’s just completely silly to check.

12. Checkrep

- Let’s see what assertions we can add to the queue code presented before that would make it more robust with respect to mistakes.

- We add a checkRep function that stands for check representation, and this is a function that we commonly add to a data structure or to other functions that checks the variables in the program for self-consistency. And so what it's going to do is basically try and terminate the program if some invariant that we know should hold over the program's data structures fails to hold.
- Next we are going to break our queue by making the enqueue fail to properly reset the tail variable when it overflows. What we have here is logic that when self.tail is equal to self.max, we reset it back to 0. And we are going to set it to 1 instead. What we want is to write a tighter set of assertions for our queue so that our checkRep can catch this before we actually return the wrong thing to the user.

```
def enqueue(self, x):
      if self.size == self.max:
            return False
      self.data[self.tail] = x
      self.size += 1
      self.tail += 1
      if self.tail == self.max:
            self.tail = 1 # code defect
      return True

def checkRep(self):
      assert self.size >= 0 and self.size <= self.max
      if self.tail > self.head:
            assert (self.tail - self.head) == self.size
      if self.tail < self.head:
            assert (self.head - self.tail) == (self.max - self.size)
      if self.tail == self.head:
            assert (self.size == 0) or (self.size == self.max)
       return
```

13. Why assertions?

- Make code self-checking, leading to more effective testing
- Make code fail early, closer to the bug
- Assign blame
- Document assumptions, preconditions, postconditions, & invariants

14. Are assertions used in production?

- The GCC source code base contains more than 9000 assertions.
- The LLVM compiler suite contains about 13,000 assertions, and that's over about 1.4 billion lines of code for a total of about 1 assertion per 110 lines of code.
- We are counting raw lines of code, not source lines of code. This includes blanks, comments, and everything.

- The LLVM and GCC developers have made a pretty serious commitment to checking assumptions, preconditions, post conditions, and invariant in the code that they wrote.
- Much of the time the bugs show up as assertion violations.

15. Disabling assertions

- "`Python -O`" disable assertions
- Advantages:
  - Code runs faster
  - Code keeps going

    What is it that we're trying to do with our system? Is it better to keep going or is it better to stop? Keeping going after some condition is true that will lead to an assertion violation may lead to a completely erroneous execution. On the other hand, possibly, that's better than actually stoping.
- Disadvantages:
  - What if our code relies on a side-effecting assertion?
  - Even in production code may be better to fail early

    Do our users want the system to die or do they want the system give them some completely wrong result?

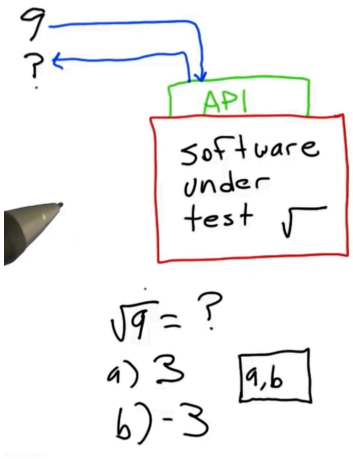
16. When to use assertions

- Disadvantages:
  - What if our code relies on a side-effecting assertion?
  - Even în production code, may be better to fail early
- **Julian Seward**, Valgrind:

  "This code is absolutely loaded with assertions, and these are permanently enabled … as Valgrind has become more widely used, they have shown their worth, pulling up various bugs which otherwise have appeared as hard-to-find segmentation faults.
  I am of the view that it's acceptable to spend 5% of the total running time … doing assertion checks."
- On the other hand… if you're doing something so critical that it keeps going that it resembles landing a spaceship on Mars, then go ahead and turn off assertions in your production software.

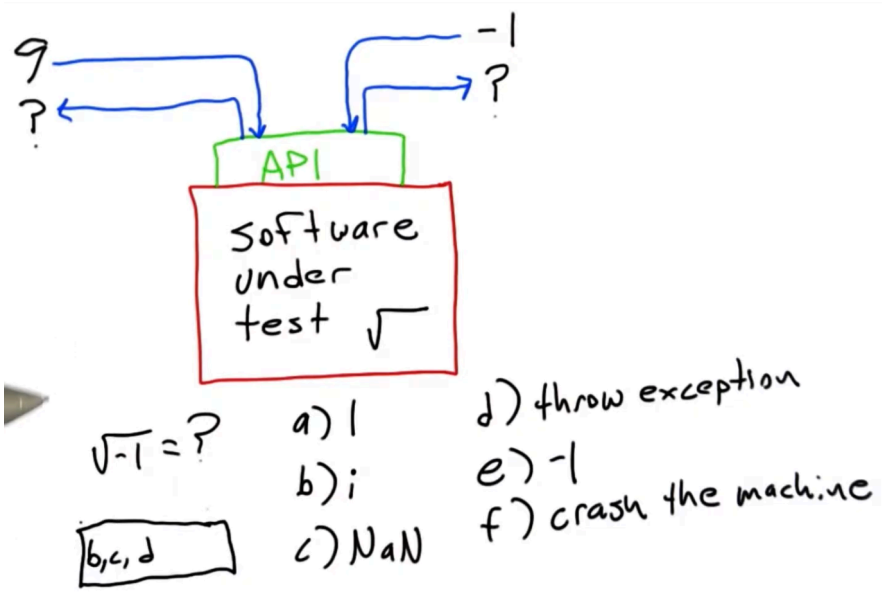
17. Specifications

- The SUT is providing some set of APIs, i.e. a set of function calls that can be called by another software.

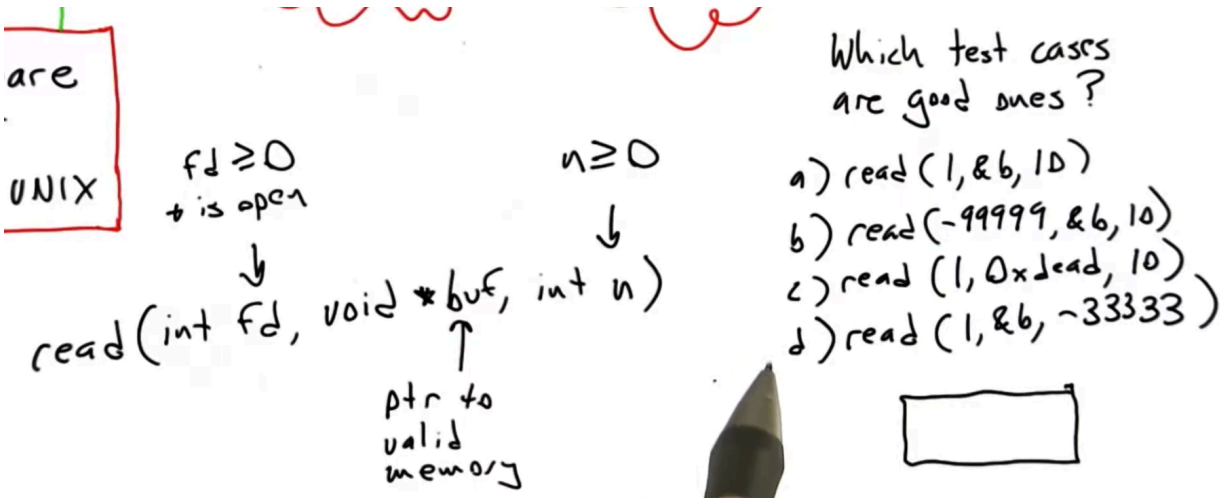


- The answer to this quiz is any answer is acceptable. The issue that we're getting at here is what's the specification for the SUT, for our square root routine?
  Is it defined to return the positive value? the negative value? Can it return either of them? And this is what we're getting at here is that software always operates under some sort of a specification.

18. Refining the specification



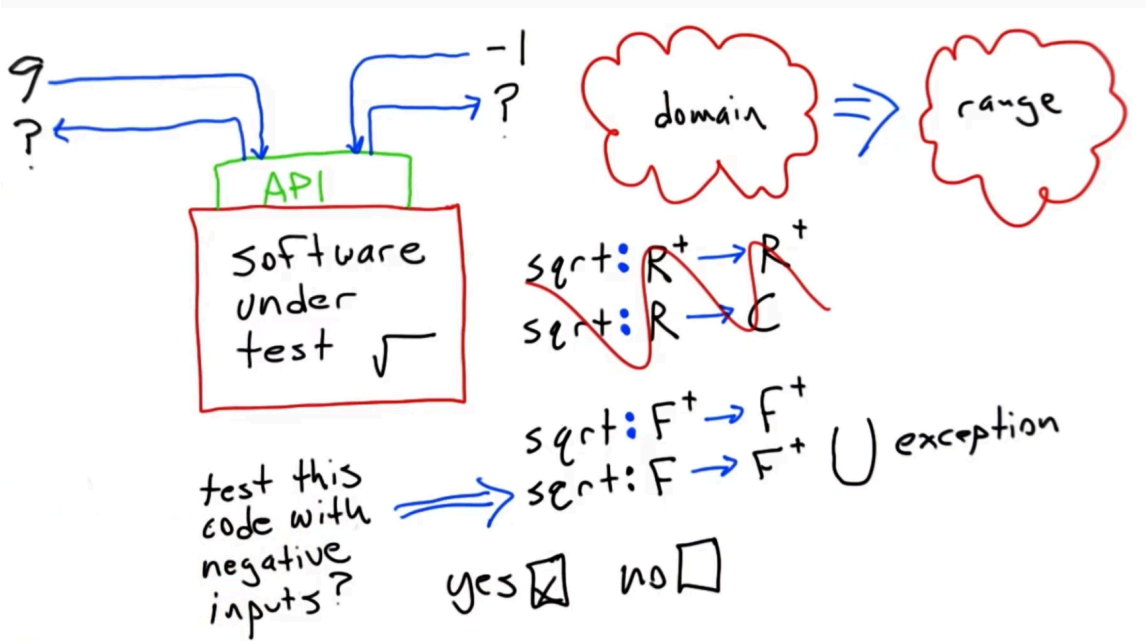
- Running a simple test case is forcing us to think about the specification for the SUT. And in fact, this is really common that as soon as we start testing a piece of software, we start to think about what the software is actually supposed to be doing.



- Often when we're testing software, we're not so much just looking for bugs in the software, but we're helping to refine the specification for the SUT.

19. Domains and ranges

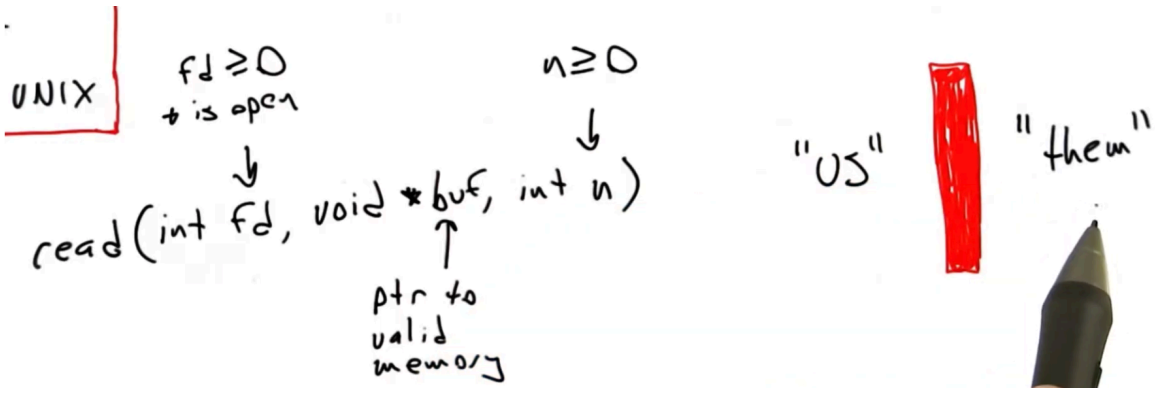
- If we think of a piece of software as a mathematical object, we'll find the software has a **domain of values**. Correspondingly, every piece of software also has a **range**.
- Sometimes as a software tester, you'll test code with an input that looks like it should be part of the domain and the code will malfunction, will crash with some sort of a bad error, perhaps maybe not throw an exception but rather actually exit abnormally.
- **Restrictions on the domain of functions** are actually a very **valuable tool in practice** because otherwise, every function or piece of software that we implement, has to contain maximal defensive code against illegal inputs. And in practice, this kind of defensive coding is not generally possible.



20. Good test cases

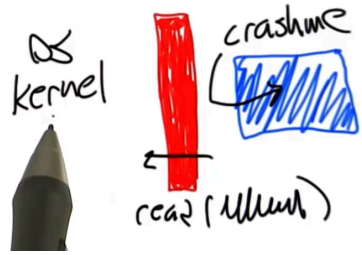
- Testing UNIX platform. All that read system call does is takes a file that's already open and reads some bytes out of it into the address space of the process that calls a read.

- Answer: all of the above.
- We're the kernel implementers, and our job is to keep the machine running to provide isolation between users and basically to enforce all of the security policies that operating systems are designed to enforce.
- On the other side of the boundary we have them. These are our users. They might not actually be malicious but writing buggy codes.
- Therefore, if we're testing the operating system interface, we really need to be issuing calls like read with garbage.



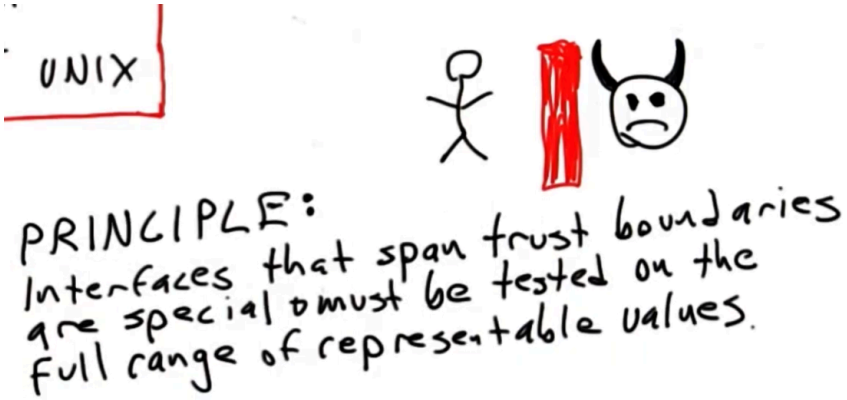
21. Crashme

- This is one of the ways that we actually test operating systems: using a tool called crashme that allocates a block of memory, writes totally random garbage into it, then it masks off all signal handlers, i.e. system level exception handlers and it jumps into the block of garbage, i.e. it starts executing completely garbage bytes.

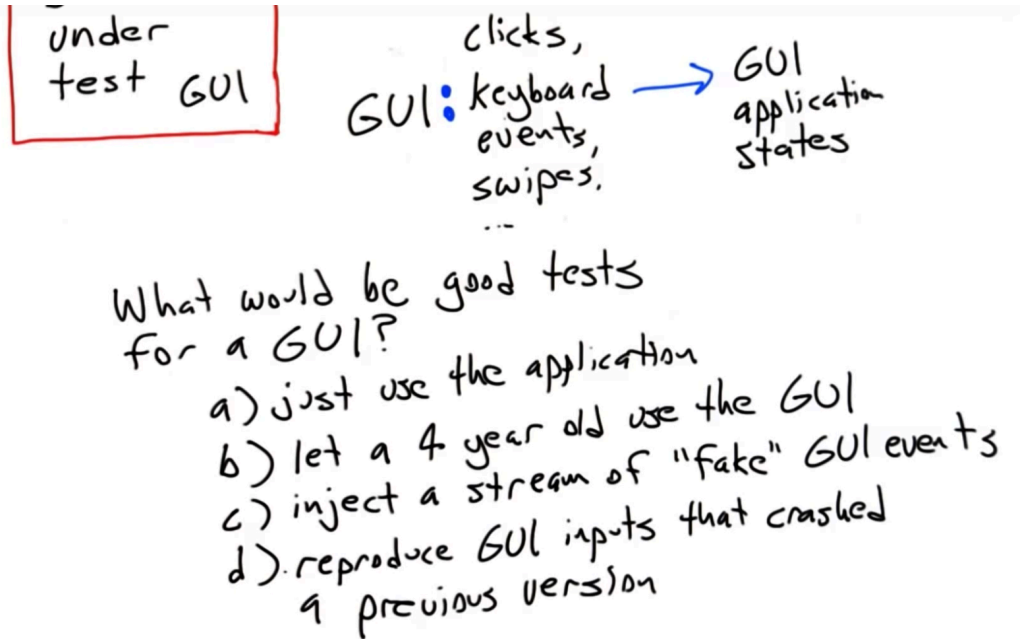


•So over time what we end up doing is exploring, i.e. testing a lot of operating system calls, that contain really weird and unexpected values.

- And if the kernel goes ahead and keeps running properly and keeps trying to kill the crashme process, then the operating system kernel is succeeding.
- If the kernel ever falls over, if it crashes, then we've discovered a bug.



22. Testing a GUI



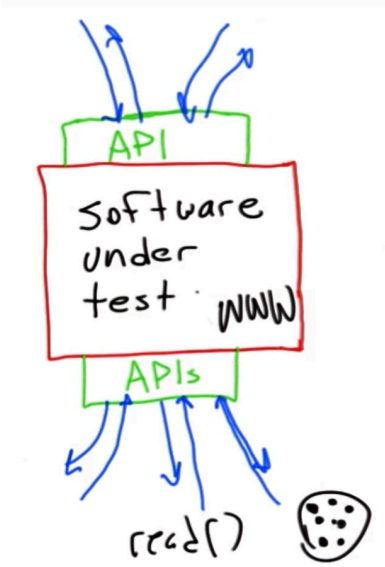
- Answer: all of the above.

23. Trust relationships

- The question: can I trust my teammates, and can my teammates trust me to always generate inputs when using the various APIs that remain within the domain of those inputs? And of course the answer is generally no.

•In fact, I probably can't even trust myself to always generate inputs that are within the domain of acceptable inputs for APIs.

- This brings us to is the idea of **defensive coding** i.e, error checking for its own sake to detect internal inconsistencies.
- Software doesn’t just provide APIs, it also uses them.



- Let’s say the **SUT** is something like a **web browser**. One thing we can do is test the web browser using the APIs that it provides, i.e. using its GUI and not worry about testing it with respect to the APIs that it uses.
- Let's take the case where our web browser is storing cookies. Most of the time during testing, we expect the storage and retrieval of cookies to operate perfectly normally. But what happens if, for example, the hard disk is full when the web browser tries to store a cookie?
- If we just hope that the software does the right thing, then one of the **golden rules of testing is we shouldn't ever just hope that it does something; we need to actually check this.**

- Back to the UNIX read system call. Before, we were concerned with the domain of the read system call, i.e. the set of possible valid arguments to the read system call and now we're concerned with the range because now we're not testing the UNIX operating system anymore; we're testing a program that runs on top of the UNIX OS.
- read is allowed to read less bytes than you actually asked for. It's going to return some number between 0 and count, but we don't know what number it's going to return.
- Another thing that read can do is just fail outright, i.e. it can return -1 to the application but it turns out that there are a whole lot of different reasons for that kind of a failure.

```
READ(2)                          Linux Programmer's Manual                          READ(2)


NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

       If  count is zero, read() returns zero and has no other results.  If count is greater than SSIZE_MAX,
       the result is unspecified.

RETURN VALUE
       On success, the number of bytes read is returned (zero indicates end of file), and the file  position
       is  advanced  by  this number.  It is not an error if this number is smaller than the number of bytes
       requested; this may happen for example because fewer bytes are actually available  right  now  (maybe
       because  we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or
       because read() was interrupted by a signal.  On error, -1 is returned, and  errno  is  set  appropri-
       ately.  In this case it is left unspecified whether the file position (if any) changes.

ERRORS
       EAGAIN The  file  descriptor fd refers to a file other than a socket and has been marked non-blocking
              (O_NONBLOCK), and the read would block.
```

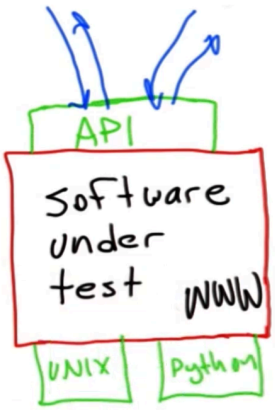
- We can see here that there are at least 9 different error conditions that read can return.

```
ERRORS
       EAGAIN The  file  descriptor fd refers to a file other than a socket and has been marked non-blocking
              (O_NONBLOCK), and the read would block.

       EAGAIN or EWOULDBLOCK
              The file descriptor fd refers to a socket and has been marked non-blocking  (O_NONBLOCK),  and
              the read would block.  POSIX.1-2001 allows either error to be returned for this case, and does
              not require these constants to have the same value, so a portable application should check for
              both possibilities.

       EBADF  fd is not a valid file descriptor or is not open for reading.

       EFAULT buf is outside your accessible address space.

       EINTR  The call was interrupted by a signal before any data was read; see signal(7).

       EINVAL fd  is  attached to an object which is unsuitable for reading; or the file was opened with the
              O_DIRECT flag, and either the address specified in buf, the value specified in count,  or  the
              current file offset is not suitably aligned.

       EINVAL fd  was created via a call to timerfd_create(2) and the wrong size buffer was given to read();
              see timerfd_create(2) for further information.

       EIO    I/O error.  This will happen for example when the process is in a  background  process  group,
              tries  to  read from its controlling tty, and either it is ignoring or blocking SIGTTIN or its
              process group is orphaned.  It may also occur when there is a low-level I/O error while  read-
              ing from a disk or tape.

       EISDIR fd refers to a directory.
```

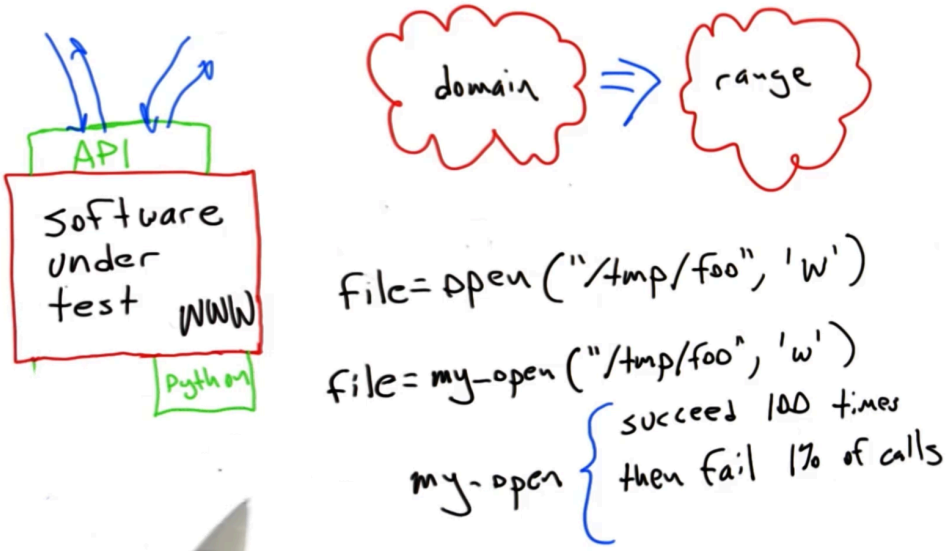
- The application might have to do different things depending on which of these values it gets. And the point is it might be very hard as people testing the web browser to actually make the operating system read call return all of those different values.
- And until we've tested it with all of those different values, we're left with software whose behavior we probably don't understand, and, therefore, it's software that hasn't been tested very well.

24. Fault injection

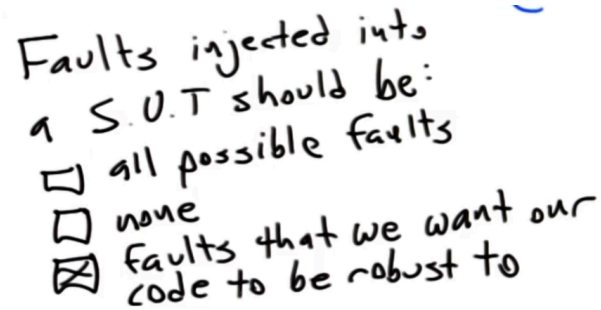
- What we have is a fairly difficult testing problem. In practice, there are a couple of different ways to deal with it.
- First, you should always try to use low level programming or interfaces that are predictable and that return friendly error codes. Given a choice between using the UNIX system call and using the Python libraries, you'd almost always choose the Python libraries.



- We don't always have the option of doing this so we're forced to use these bad style APIs sometimes. From a testing point of view, we can often use the technique called **fault injection** to deal with these kind of problems.
- Let's assume for the moment that we're using the Python library to create a file. We might have a fairly hard time testing the case where the open call fails because this call might almost always succeed. So what we can do instead is call a different function, my_open, which has an identical interface and almost an identical implementation.
- So we have a stub function and we can sometimes cause the open system call to fail.



- In practice, you have to be pretty careful with fault injection. One thing that can happen if you make my_open fail too often, for example, if it fails 50% of the time, then a program that's using it probably will never get off the ground.
- We would have to experiment with what kind of failure rates are good at testing the SUT:



25. Timing dependent problems

- We have our SUT, and it provides some APIs. Each API's collection of functions and most of the work that we have during testing is going to be calling these functions with various values and looking at the results.
- The issue is that the overt explicit interfaces that we see here don't represent all the possible inputs and outputs that we might care about.
- For example, on the input side it's completely possible that our SUT cares about the time at which inputs arrive.
- It might be the case that our software responds differently if 2 inputs arrive very close together than it does if 2 inputs arrive separated by a large amount of time.
- Another example is a web browser where if the data corresponding to a web page is returned in short time, this data will get rendered as a web page. But if the data that comes from the network is scattered across too much time, this is going to result in some sort of a timeout, i.e. SUT will render some sort of an error page.

26. Therac 25

- An extreme example of timing-dependent input being difficult to deal with is presented in the following. In the 1980s, there was a r**adiation therapy machine** called a Therac-25.

•It was used to deliver a highly concentrated beam of radiation to a part of a human body where that beam could be used to destroy cancerous tissue without harming tissue that's nearby.

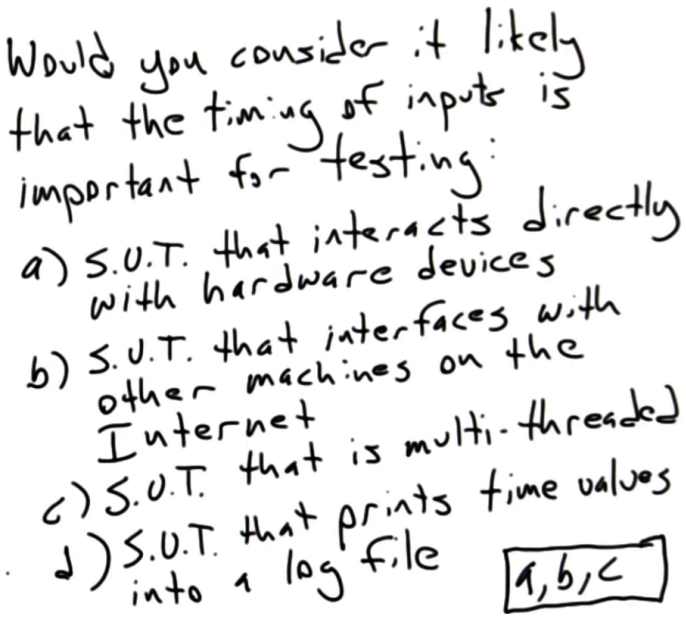
•This was not a safe technology as it depended on skilled operators and also highly safe software.

• There was a tragic series of mistakes where 6 people were subjected to massive radiation overdoses and several of these people died.

•The Therac-25 had **serious issues with its software**.

- It turned out that the people developing the software put a number of bugs into it.
- The particular bug was a software bug called a **race condition** that involved the keyboard input to the radiation therapy machine. If the operator of the machine typed slowly, the bug was very unlikely to be triggered. As they treated hundreds and hundreds of patients, they typed faster and faster, and eventually they started triggering this bug.
- The kind of quandary that this scenario raises for us as software testers is **do we have to care about the time at which inputs arrive at our SUT, or can we not worry about that?**
- And so obviously, for the Therac-25 and obviously, also for something like a Linux kernel, the time at which inputs arrive is relevant.
- On the other hand, unless we've been extremely sloppy, the square root function that we've been talking about won't care about the time at which its inputs arrive.

27. Testing timing

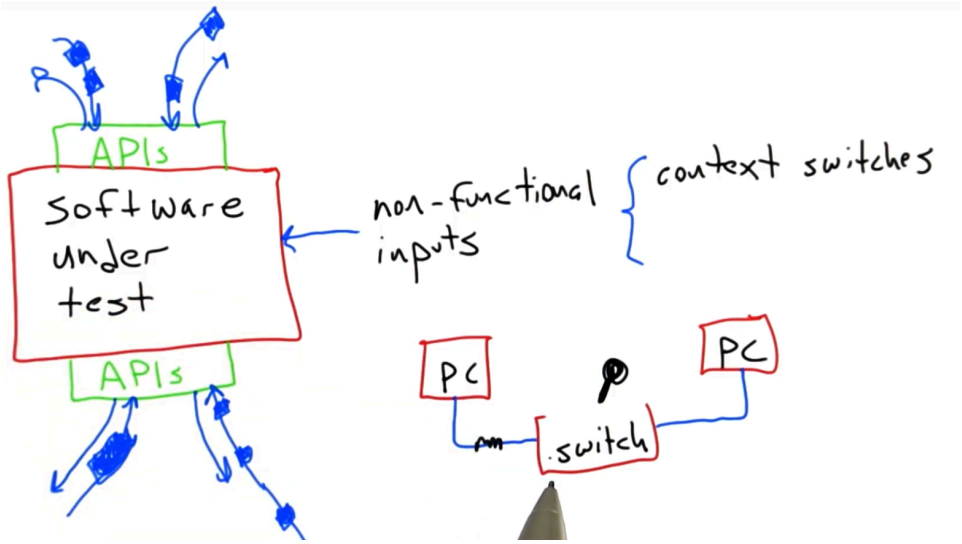


28. Taking time into account

- In order to figure out if timing matters for the inputs for the SUT is think about its specification, its requirements, and also look at the source code for things like timeout, time sleep, and basically values or computations that depend on the time at which things happen.

29. Nonfunctional inputs

- These are inputs that affect the operation of a SUT that have nothing to do with the APIs provided or that are used by the software that we're testing.
- Context switches are switches between different threads of execution in a multi-threaded SUT. The issue is that multiple threads of execution are scheduled along different processors on the physical machine that we're running on at different times, and it's the OS that makes the decisions about what thread goes on what processor at what time, and depending on the scheduling of these threads bugs in the SUT can either be concealed or revealed, and the problem is that the timing of these context switches is completely not under the control of our application.
- For example, in the late 1990s a company made very, very fast networking hardware. A problem was that the network was extremely reliable implying a real difficulty in testing the end-to-end reliability software.
- So the tester opened up a switch, exposing all of the electrical contacts inside, and then took a metal key, and run it across the contacts that were exposed. Either the network would glitch for a moment or else it would fail to resume properly:



30. Testing survey

- It's not intended to be exhaustive because there's going to be a bunch of overlap between these different categories.

•**White box testing** refers to is the fact that the tester is using detailed knowledge about the **internals of the system** in order to construct better test cases and **black box testing** refers to the fact that we're rather testing the system based on what we know about **how it's supposed to respond** to our test cases.

31. Unit testing

- Unit testing means looking at some **small software module** at a time and **testing** it **in an isolated fashion**.
- The main thing that distinguishes unit testing from other kinds of testing is that we're **testing a smaller amount of software**.
- Often the person performing the unit testing is the same person who implemented the module, and in that case we may well be doing white box testing but unit testing can also be black box testing.
- The goal of unit testing is to **find defects in the internal logic of the SUT as early as possible**, in order to create more robust software modules that we can compose later with other modules and end up with a system that actually works.
- Another thing that distinguishes unit testing from other kinds of testing is that generally, at this level, **we have no hypothesis about the patterns of usage of SUT**.

- In other words, we're going to try to test the unit with inputs from all different parts of its domain (the set of possible inputs).
- Python has a number of frameworks for unit testing and also for mock objects (fake objects).

32. Integration testing

- Integration testing refers to taking **multiple software modules that have already been unit tested and testing them in combination with each other**.

- What we're really testing are the interfaces between modules, and the question is did we define them tightly enough and specify them tightly enough that the different groups of people implementing the different modules were able to make compatible

assumptions, which are necessary for the software modules to all actually work together.

- Coming up with a software module that survives integration testing is a lot harder than creating reliable small units.

33. System testing

- Here we're asking the question **does the system as a whole meet its goals**?
- And often at this point we're doing black box testing, and that's for a couple of reasons:
  - the system is probably large enough,
  - we're not so much concerned with what's going on inside the system,
  - at this level we are often concerned with how the system will be used,
  - we may not care about asking the system work for all possible use cases. Rather, we would simply like to make sure that it performs acceptably for the important use cases.

34. Other kinds of testing

- In **differential testing** we are taking the same test input delivering it to 2 different implementations of the SUT and comparing them for equality.

- **Stress testing** is a kind of testing where a system is tested at or beyond its normal usage limits, and it's probably best described through a couple of examples.
  - With the square root function we might test it with very large numbers or very tiny numbers.
  - For an OS we might test it by making a huge number of system calls or by making very large memory allocations or by creating extremely large files.
  - For a web server we could stress test it by making many requests, or even better, by making many requests, all of which require he database to communicate with its backend.
- Stress testing is typically done to assess the robustness and reliability of the SUT.

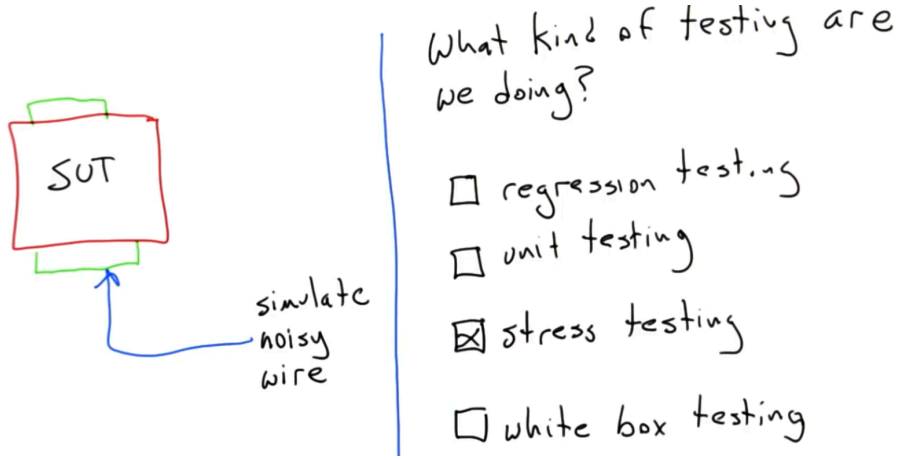
- In **random testing** we use the results of a pseudo-random number generator to randomly create test inputs, and we deliver those to the SUT.
- Very often this can be much more subtle and more

sophisticated than just throwing random bits at the software. And random testing is very often useful for finding corner cases in software systems, and the crashme program for Linux and other Unix kernels is a great example of a random tester.

- **Regression testing** always involves taking inputs that previously made the system fail and replaying them against the system.

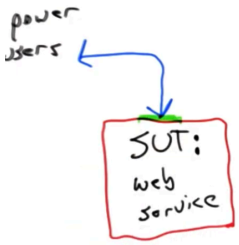
35. Testing car software

- We have a piece of SUT that is some sort of a critical embedded system, for example, it's controlling airbags on a vehicle. And this vehicle is going to be out in the field for many years, and so cars get subjected to fairly extreme conditions.
  - They have a lot of heating and cooling cycles.
  - They get exposed to salt and mud and water, and so the cars are in a fairly harsh environment.
- We want to know how the software responds to defects that get introduced into the automobile's wiring as this vehicle ages in the field.
- We're going to use some sort of a simulation module to simulate a noisy wire, a wire whose insulation has rubbed off and that is not perfectly conducting signals anymore.
- What kind of testing are we doing?



36. Testing a web service

- SUT is some sort of a web service, exposed to some small fraction of users who have been selected, based on their willingness and desire to use new features.
- What kind of testing are we doing?



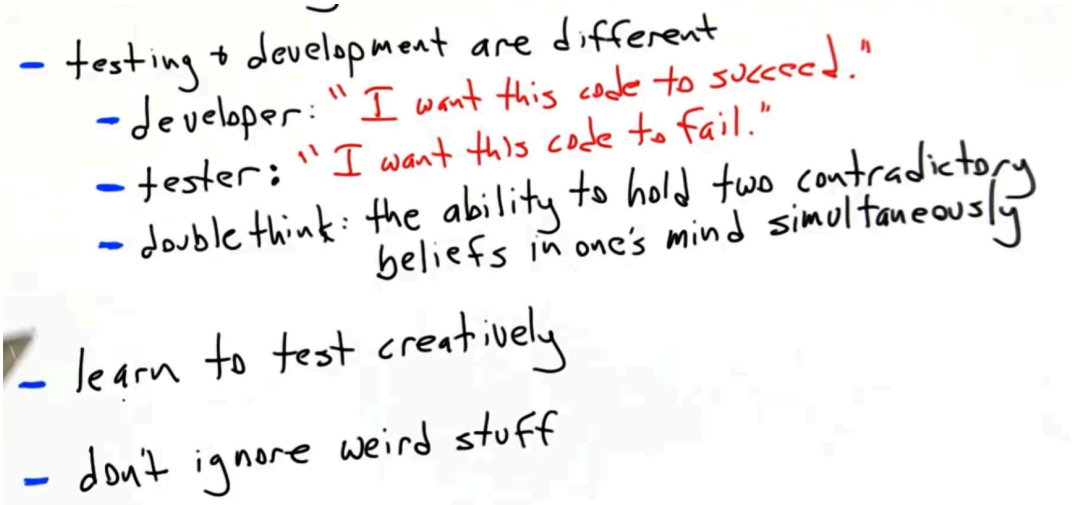
- [ ] integration testing
- [ ] differential testing
- [ ] random testing
- [x] system/validation testing

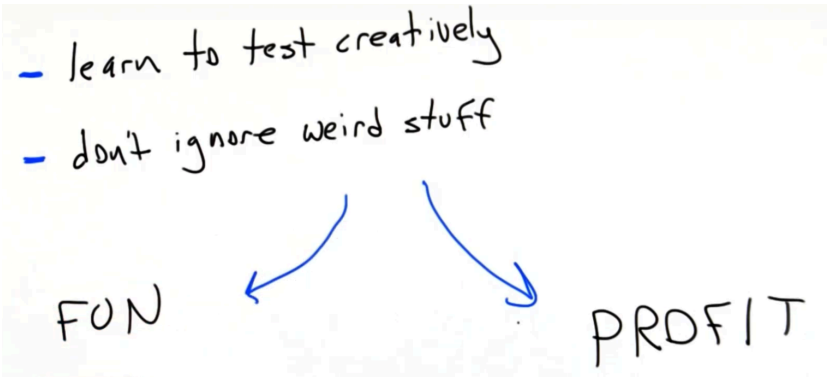
37. Testing a new library

- In this scenario we have some large SUT, and part of it is a library that has been giving us problems.
- Let's say this library is implementing numerical functions, and these numerical functions have been sometimes throwing floating point exceptions and crashing our system.
- The vendor has given us a new version and we're going to spend some time checking and trying to make sure that even if it's not that great it's at least not worse than the previous version.
- What kind of testing are we doing?

- [x] unit testing
- [ ] white box testing
- [ ] black box testing
- [ ] stress testing

38. Being great at testing




- learn to test creatively
- don't ignore weird stuff
FUN
PROFIT