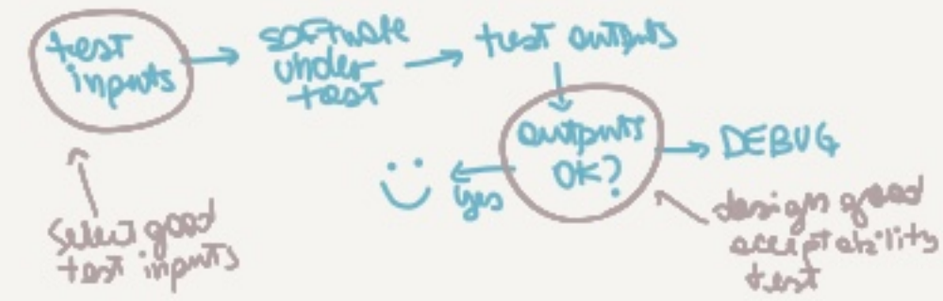
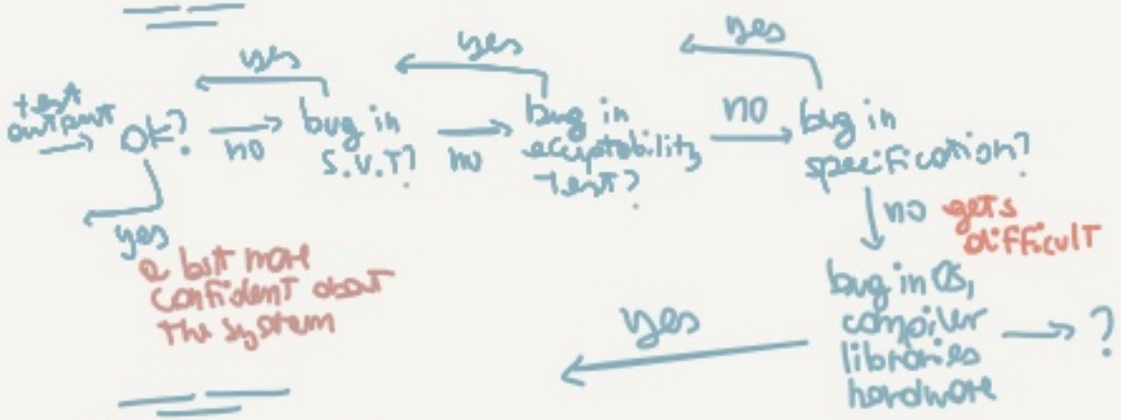
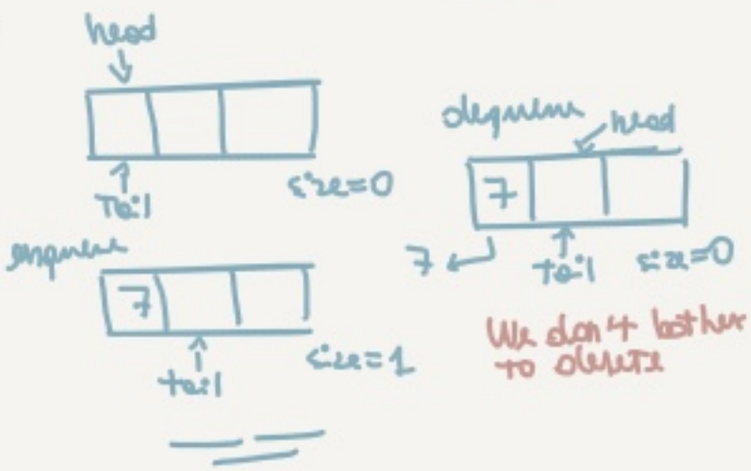




Fixed-size queue
- enqueue
- dequeue
- FIFO order



We don't bother to dequeue

enqueue(7)
x = dequeue()
if x==7:
success

↓

A single testcase, represents a whole class of actual executions of the system we're testing

Arguments about equivalent testcases depend on the system

"Cover all possible branches when writing test-cases"

Creating Testable Software
- clean code
- refactor (when clean isn't possible)
- should always be able to describe what a module does and how it interacts with other code

- no extra threads
- no swamp of global variables
- no pointer soup
- when applicable, support fault injection
- assertions, assertions, assertions

Assertions

Executable check, for a property, that must be true

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

- assertions are not for error handling (eg not for arg>0)
- NO SIDE EFFECTS (eg assert foo()==0 where foo changes a global variable)
- No silly assertions

We want to check a non-trivial property, that could be wrong if we actually made a mistake in our logic

Disabling assertions
- code runs faster (python -O disables them)
- code keeps going

- What if code relies on side effecting assertions? (wrong one of them)
- Even in production code, may be better to fail early !!!

As soon as we start testing a piece of software, we start to really think about what it is supposed to be doing
Often when testing, we're helping to refine the specification

Domains and ranges
We can think of a software as a mathematical object with a domain and a corresponding range

set of inputs the software is designed to handle — possible outputs

sqrt: $F \to F^+ \cup$ exception

Some higher level software artifacts, don't have the luxury of limiting their domains (such as an OS)

When testing these systems, we want to test all possible values of the parameters to the call that it supports

If we have an interface or an API that represents a trust boundary, then we have to test that API with all possible representative values

Testing Principle

eg DOMAIN
GUI: clicks, keyboard, swipes → GUI application states

You must check all possible combinations of events since the domain is gigantic

A software does not just provide APIs, it also uses them

- We could not worry about the APIs that it uses.
- We'd like the APIs to do the right thing, but we need to test this
- But don't have direct control, sometimes we can simulate what we want, but it's not always the case

A way to deal with this is using low level programming languages or interfaces that are predictable and return friendlier error code when we have the option.
(eg between Unix system calls and python libraries, you almost always use the python libraries)

Another way is to use fault injection

```
file=open("/tmp/foo",'w')
File=my_open("/tmp/foo",'w')
```

you may have a hard time testing where the open call fails. /tmp is in most machines

a stub function, almost functionally equivalent to open, write by us. Conditionally, inside that function's code, we can cause the call to open to fail

A fault injection should not make the code fail to often

Timing

All those we talked about do not represent all possible inputs and outputs that we might care about. A really common example is timing of inputs.

To know if you need to test timing think of the specification, about what it's trying to do, about its requirements. And also, look at the source code for timeout, sleeps or values that depend on the time.

Non-Functional inputs

Inputs that affects the operations of a software under test, that have nothing to do with the APIs provided and used by the software we're testing

- context-switches ← depending on thread scheduling, bugs can be either concealed or revealed.

The timing is not controlled by our application. This makes testing multithreaded software difficult

Short survey

White box testing. The tester is using detailed knowledge about the internals of the system in order to construct better testcases.

Black box. We're testing based on how it is supposed to respond

Unit testing. Look at small software modules at a time, and testing it in a isolated fashion. Can be either white box or black box testing.
The goal is to find defects in the internal logic of the S.U.T. as early as possible in order to create more robust software modules.
At this level, we have no hypothesis about the patterns of the S.U.T., we try to the the unit with inputs from all different parts of the domain.
There are many frameworks for unit testing

System testing, we want to know whether the system as a whole meets his goal. Often at this point we're doing black box testing as
- the system is now large enough that a knowledge of its internals does not really help creating good test cases

Stress testing, the system is tested at or beyond its normal usage limits. To assess robustness and reliability of the SUT

Random testing, randomly create test inputs. Useful for finding corner cases

Integration testing refers to taking multiple software modules, already unit-tested, and testing them in combination with each other.
Are the interfaces between modules specified tightly enough so that the different people implementing them were able to make compatible assumptions.
Usually software modules don't survive integration testing

Differential testing, we take the same test input, deliver it to two different implementations of the SUT, and compare them for equality.
It is extremely powerful in the case where we have multiple versions.

Being great at testing

- developer: "I want this code to succeed"
  tester: "I want this code to fail" ← as the tester and goal is creating stronger code which later on doesn't fail

doublethink: invited to be both a great tester and developer.

The contradictory nature of those beliefs is apparent as they both have the end goal of creating great software

- Learn to test creatively (does a much better job than rote testing. Even worse rote testing with obvious inputs)
- Don't ignore weird things. We usually get little hints of things wrong software that would've lead us to find issues

COVERAGE TESTING

You can think of problems in released software as coming from things that people forgot to test. That means that testing was inadequate but developers were unaware of that

Code coverage is a collection of techniques where automated tools can tell us places where our testing strategy is not doing a good job

How much testing is enough?

our testing is confined to some small part of the input domain.
Even the small part, may contain an infinite # of test cases. There are other parts of the input domain we didn't think to test, that may result in outputs that are not okay.
It depends on how you've broken up the input domain

We want some sort of automated system that looks at our testing effort and gives us a score.
This helps us find parts of the input domain that needs more testing.
Also, we might be able to argue that we've done enough testing, or that the system hasn't been tested enough yet.

It would be nice if we could take a large test suite, one that maybe takes several days to run, and identify parts that are completely redundant, that have roughly the same testing effect on the system. Assigning a score lets us do that as well.

Partitioning the input domain

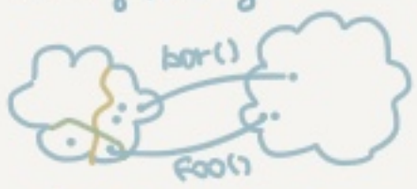
You want to partition the input domain into a number of different classes, so that all of the points within each class are treated the same by the SUT.
While finding those classes we could look at specs, implementation or use our vague intuition

Sometimes, what we thought was a class of inputs that are all equivalent, isn't really, and there is a different class hiding within this class which triggers a bug, even though the original test case didn't

So in practice what we ended up with is not this idea of coming up with a good partitioning for the input domain. Rather the notion of test coverage.

Test coverage is trying to accomplish the same thing that partitioning was accomplishing, but it does it in a different way. It is an automatic way of partitioning the input domain, based on observed features of the source code

For example, a particular kind of test coverage is function coverage, which is achieved if every function in our source code is executed during testing



We end up with a score, called test coverage metrics
eg 1811/2500 functions

that would not be a good score, we can look at those functions and come up with a test input that will cause to execute them.

If we cannot find those inputs it either can't be called or we don't know the system well enough

Test coverage, a measure of the proportion of the program exercised during testing

+ gives us an objective score
+ when coverage is <100%, we are given meaningful tasks.

- not very helpful in filling errors of omission ...since it's whitebox.
things that we should've implemented
- difficult to interpret scores <100%.
For large, complex software systems, achieving this is often difficult.
- 100% coverage doesn't mean all bugs were found

"When our test coverage fails, it is giving us a bit of evidence, that our test suite is poorly thought out, meaning it is failing to exercise functionality present in our code."

" We should not complete coverage and a number of successful testcases fool us into thinking that a piece of code is right. Often deeper analysis is necessary

COVERAGE METRICS

there are a lot of them!

Statement coverage,
Must execute all statements

Line coverage SIMILAR TO PREVIOUS
Must execute every line in the source code (a line can have more lines)

Branch coverage, a branch in the code is covered if it executes both ways

```
if x==0:
    y+=1
if y==0:
    x+=1
```

← to achieve 100% branch coverage here, you want to execute the if's and the else's

Other coverage metrics are interesting, even though you may not use them in practice, because they form part of the answer to the question "how should we come up with good test inputs, in order to find bugs?"

loop coverage, executes each loop zero, one more than one times
loop boundary conditions, are extremely frequent source of bugs in real codes.

Modified condition decision coverage → MCDC, branch coverage + conditions take all possible values + every condition independently affects its outcome

For safety critical software

path coverage, cares how you got to a certain piece of code.

```
def foo(x,y):
    for i in range(x):
        something()
    if y:
        somethingElse()
```

As x increases, there are unlimited branches. But it gives something to think about when generating test-cases

Boundary coverage, when a program depends on some numerical range, and when a program has different behaviours based on numbers within that range, then we should test numbers close to the boundary

What about concurrent software?

```
xfer(amount)
```

Shared between different calls → a_1 += amount
a_2 -= amount
return OK

As long as we transfer money between different accounts, probably everything is alright

since the function does not use locks while it manipulates the accounts, if multiple threads operate on the same account concurrently, then it's going to be a problem

Our coverage metrics make sure that threads T1 and T2, both call the function at the same time while transfering between the same amount. T1 get part way into the function and stops running, and then we start T2 and resume T1 after it finishes