

Lecture Notes

October 27, 2020

Working in a team: how to choose test inputs



Consider a simple calendar program where the user can enter an arbitrary date and the program returns the corresponding day of the week. The user is prompted to enter the month, the day, and the year in separate fields.

The calendar represents months as integers 1 to 12, days as integers 1 to 31, and years as integers 1 to 9999 (AD).

How should we choose test cases for this program?



What would be some good test inputs for “test to pass”?
(initial smoke test)

What would be some good test inputs for “test to fail”?
(most testing is intended to reveal bugs)

How do we know whether the test outputs are correct?

The entity that knows the correct outputs, and/or can check that the actual outputs are the same as or consistent with the expected outputs, is called the “test oracle”.

Test oracles will be discussed in a later lecture. For now, assume the human tester knows what the correct output should be for each input.

Do you know the day of the week for January 1, 1?

How about October 27, 3333?



Should we test the month, day, or year 0?

What about testing with negative integers?

Should we test with month 13?

What about day 32?

Is there anything special about the days 29, 30 or 31 that we should test?

Should we test with non-numeric input?



Now consider another calendar program that represents months as strings, “January”, “February”, and so on.

Focusing only on the months, what would be some good test inputs for “test to pass”?

What would be some good test inputs for “test to fail”?



Say the same calendar program that represents months as strings, “January”, “February”, and so on also allows *unambiguous* abbreviations, e.g., “F”, “Feb”, but gives an error message for *ambiguous* abbreviations, e.g., “J”, “Ju”.

Again focusing on the months, what would change about testing this program?

For the initial calendar with integer months, 0 and 13 are invalid inputs. What are some examples of *invalid* inputs for the calendar that represents months as strings?



Should we consider month inputs with valid vs. invalid *formatting*?

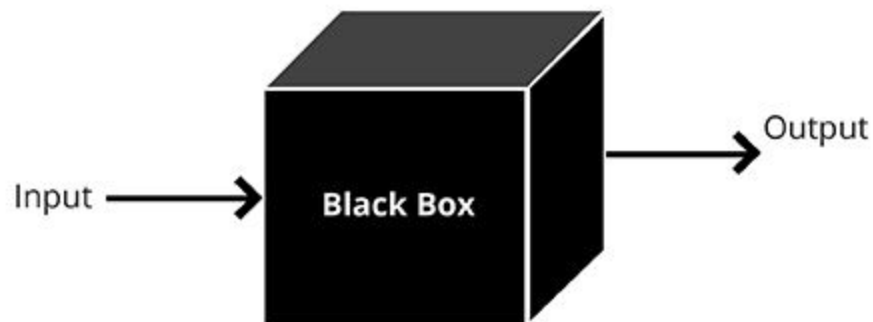
- Printing vs. non-printing characters
- Upper vs. lower case
- Space, tab, newline characters (whitespace)
- Any characters with special meaning to the application (or to the implementation, e.g., programming language, SQL)
- Null/empty string
- Any string above maximum buffer size (the buffer is the data storage for reading the string input).

What we have been doing is system testing.

How should we choose the inputs for unit testing?

Let's first review blackbox vs. whitebox testing: There are different testing techniques - and different approaches to choosing test inputs - for black box vs. white box testing. Gray box combines aspects of both.

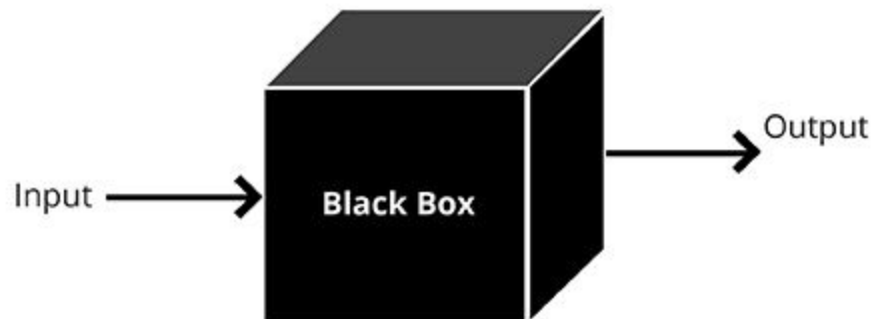
BLACK BOX TESTING APPROACH



Consider what the box is supposed to do, not how it does it. Don't look inside the box.

For system testing, the box is the full program. We choose external inputs and check the externally visible outputs.

BLACK BOX TESTING APPROACH



The box might be a component in a larger system, used as a distinct process, library, service, etc. Then choose inputs according to how the box is used within the system.

We have been doing blackbox system testing with the calendar program, but the same tests might be appropriate for a calendar component - except the data would probably be provided in a different way, e.g., in network packets.

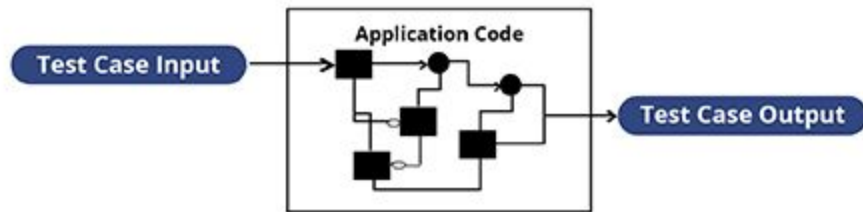
For unit testing, the box is a method (or function, procedure, subroutine). Choose inputs to the methods and check the outputs of the method. The method might contain one line of code or might invoke a complicated graph of method calls.

Note that “inputs” to methods may include parameters, global variables (e.g., static fields in classes), data entered by the user, data received from network/database/files/devices, return values and side-effects from API calls visible inside the code.

“Outputs” from methods may include return values from the code, global variables changed by the code, data displayed to the user, data sent to network/database/files/devices, data supplied to APIs that become visible outside the code.

Inputs/outputs to full systems can include most of these.

WHITE BOX TESTING APPROACH



Leverage full knowledge of what is inside the box

White box testing is necessary to achieve coverage - track what code has and has not already been exercised by previous tests, then choose test inputs specifically to force execution of previously unexercised code.







Intuition: If the test suite never executed a method, a statement or a branch, we should not have any confidence in what that code will do when it is eventually executed during operation. (If it will never be executed during operation, why is it there?)




During component and system testing, check any error status, exceptions, logs, network and device I/O, etc. produced by one box that are visible to other boxes or to human users (or administrators) and external systems.

Consider leftover side-effects, e.g., temporary files, files or database connections left open.

Integration testing is a special case of gray box testing

	Unit Testing	Integration Testing
		
		



Integration testing considers two or more boxes that communicate or depend on each other in some way (or where one box communicates with or depends on another box, does not need to be reciprocal).

Choose inputs aimed to exercise the communication paths or dependencies among boxes. Implicitly combines black box and white box, since we need to know what inputs to one box will lead to exercising the other box(es).

Class (module) testing is a special case of integration testing that is often conducted together with unit testing. The communication paths and dependencies of interest are only between units within the class (module).

How do we choose inputs for blackbox testing?

One potential approach to choosing black box test inputs is to select a set of white box inputs that exercises everything.

But coverage cannot detect *missing* code!

Another approach is to choose a set of inputs that we think would make a nice demo. (This usually does not address unit and integration testing.)

But what happens when the demo audience asks to “drive”? If this is higher level management in your company, your venture capital investor, or the instructor/TA who decides your grade, you cannot say No.

A third approach chooses a few inputs at random from among valid inputs.

However, users (and demo audiences) will invariably try invalid inputs, so we also need to pick a few inputs at random from among invalid inputs.

Will this be sufficient to ensure that all the user stories and/or use cases operate correctly?

Maybe... or maybe not... even if we choose a few inputs at random for every feature (at system level) and for every method (at unit level).

Yet another approach is to choose test inputs for each of the “conditions of satisfaction” accompanying the user stories and/or the individual steps of use cases’ basic and alternative flows.

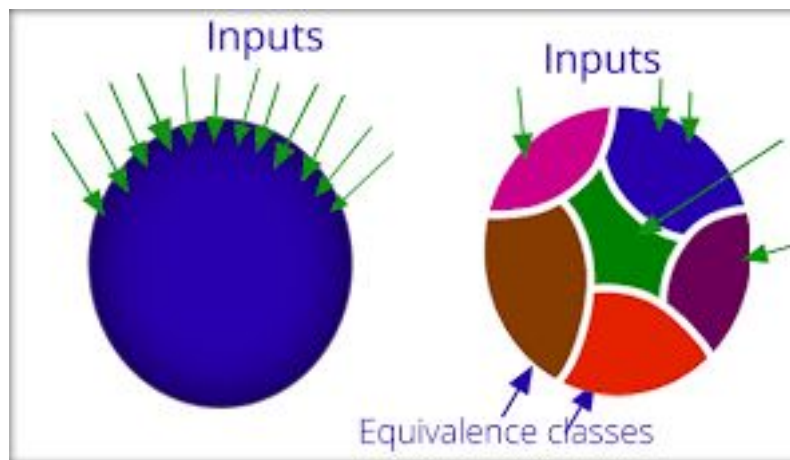
This *may* provide sufficient testing of the full system, but does not address unit and integration testing.

How about testing *all* possible inputs?

It is not feasible to test *all* inputs for non-trivial programs. Even for trivial programs, like the weekday calendar, such testing would take a long time.

To maximize the chances of finding and fixing bugs before the users (or demo audiences) do, we need to choose a practically small number of inputs that, collectively, 1. represent all possible inputs and 2. are *most likely* to reveal bugs. That is, the inputs lead to actual outputs that do not match the expected outputs.

The simplest approach for representing all possible inputs to a “box” with a practically small number of examples is to divide the input space into *equivalence partitions* (also known as equivalence classes) based on **both** application/domain knowledge (e.g., the user stories and/or use cases) and software engineering knowledge.



An equivalence partition is a set of alternative input values where the software can reasonably be expected to behave equivalently (similarly).

This does **not** mean literally the same output for every input in the class.

Instead it means that if the “software under test” (SUT) behaves correctly on one example input chosen from that equivalence class, it is reasonable to believe it will behave correctly on all the other inputs in the same class.

For example, if we are testing a “box” intended to accept numbers from 1 to 1000, then there is no point in writing a thousand test cases for all 1000 valid input numbers - and also no point in writing many more test cases for all invalid input numbers between MinInt and MaxInt.

Instead, divide the test cases into three sets:

#1) Input partition with all valid inputs. Pick a single value from range 1 to 1000 as a valid test case. If we select other values between 1 and 1000 the result “should” be the same.

#2) Pick a single value for the input partition with all values below the lower limit of the range, i.e., any value below 1.

#3) Pick a single value for the input partition with all values greater than the upper limit, i.e., any value above 1000.

Equivalence partitions are typically extended with *boundary analysis* because input values at the extreme ends of the valid and invalid partitions are more likely to trigger bugs, e.g., “off by one”, “corner case”. Recall we are trying to find bugs. Tests cannot prove the absence of bugs, only their presence.

#1) Test cases with input exactly at the boundaries of the valid partition, i.e., values 1 and 1000.

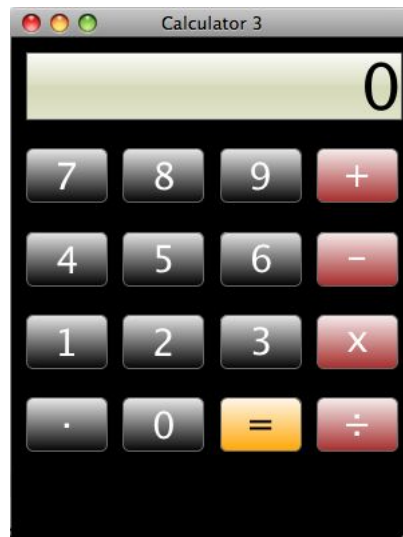
#2) Test data with values just below the extreme edges, i.e., values 0 and 999.

#3) Test data with values just above the extreme edges, i.e., values 2 and 1001.

#4) For numerical domains, make sure to include zero, positive and negative examples, as well as the underlying primitive extremes (integers). In this case, we already have 0 and some positive values, so we need to add some negative value, e.g., -1, plus MinInt and MaxInt.

#5) For numerical domains, also try some non-numerical input, e.g., “abcde” (if it is possible to provide such inputs).

Consider a calculator implemented in software, with a GUI as shown. The user clicks the buttons with a mouse, but cannot enter text into the display.



In mathematics, the “+” operation takes two numerical operands.

Say this software is tested by an external testing team separate from the developers. Based on *domain knowledge*, a tester might reasonably expect the “+” operation to behave similarly on all pairs of numbers $\langle \text{first operand}, \text{second operand} \rangle$.

So the tester tries an arbitrary pair of numbers:
 $\langle 3.141592653, 42 \rangle$.

The calculator flashes 00000000. What does this mean?

Based on *software engineering knowledge*, the tester might realize the flashing 00000000 could indicate:

- An error message because the software cannot handle numbers with more than N digits (e.g., 8).
- An error message because the software cannot handle mixed operands, i.e., one operand is floating point and the other is integer.
- A bug.
- Something else (e.g., the user has reached the limit of their “free trial” and now has to pay to continue using the calculator software).



Let's ignore the "." key (planned for version 2.0) and assume the "+" operator is currently intended only for integer operands.

A tester with software engineering knowledge, not just domain knowledge, might realize that a calculator implementation could behave differently for positive integers, zero, and negative integers.

Thus they test with five different equivalence classes:

- < pos int, pos int >
- < pos int, neg int >
- < pos int, 0 >
- < neg int, neg int >
- < 0, 0 >

Is this sufficient?

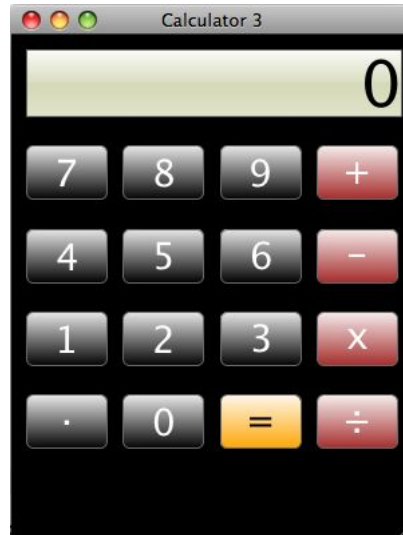
Although the “+” operator is mathematically commutative, that does not mean that the implementation (which could be buggy) is commutative.

Better to test with:

- < pos int, pos int>
- < pos int, neg int >
- < pos int, 0 >
- < neg int, pos int >
- < neg int, neg int >
- < neg int, 0 >
- < 0, pos int >
- < 0, neg int >
- < 0, 0 >

Or, more generally, define equivalence partitions separately per parameter, and then extend to the cross-product.

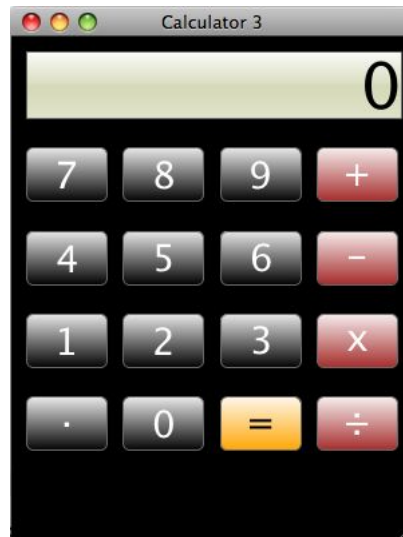
But notice the scaling problem with large numbers of parameters!



Let's say the calculator is indeed intended to handle operands only up to 8 digits (or to any other fixed number of digits).

This means there is a `MaxInt` and a `MinInt` (negative), probably specific to what the calculator GUI was designed to display rather than `MaxInt/MinInt` of the programming language, compiler, operating system, etc.

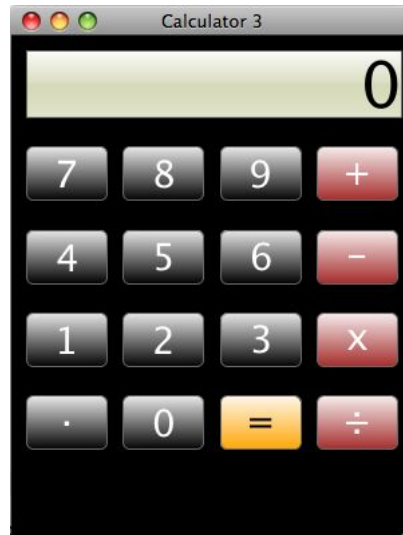
What additional tests are needed?



- < xxx, MaxInt >
- < MaxInt, xxx >
- < xxx, MinInt >
- < MinInt, xxx >
- < MaxInt, MinInt >
- < MinInt, MaxInt >

Any more?

What if the user could enter text, using a conventional keyboard, into the display window in addition to mousing the keys.



Does that change the set of equivalence partitions that need to be tested?

Design-your-burger

Please make your selection



Burger:

Cooked: ☐ rare ☐ medium ☐ well

Cheese: ☐

Lettuce: ☐

Tomato: ☐

Onion: ☐

Ketchup: ☐

Mustard: ☐

Mayo: ☐

Secret sauce: ☐



[Nutrition facts](#)

Reset

Submit

[About the chef](#)

What are the equivalence partitions?

How would we test this program?

Equivalence classes need to address required vs. optional inputs (with or without default values)

Get

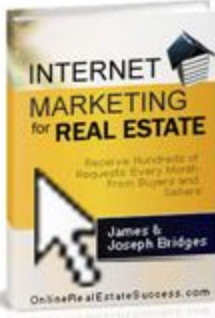
"Internet Marketing for Real Estate EBook"

via e-mail

Name*:

Email*:

Phone*:



* Indicates a required field

* Indicates an optional field

What are the equivalence partitions?

How would we test this program?

Distinct parameters may be related to each other in forming equivalence classes

Let's say we had a function that took two variables (blood sugar level and level of frustration) to calculate "mood"

Assume valid inputs are

- Blood sugar 70-100
- Frustration is 50% .. 100%

For $50 \leq F \leq 90$

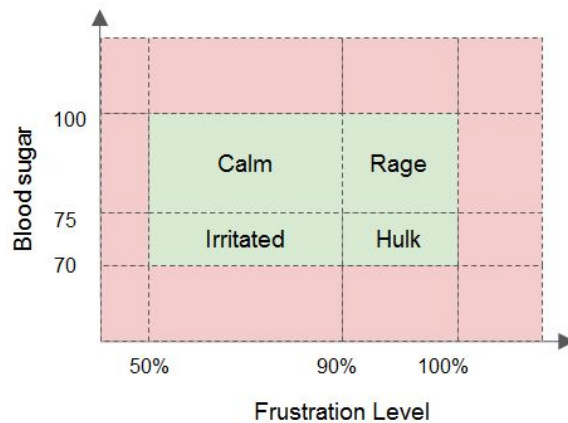
75 \leq BS \leq 100: Calm

BS < 75: Irritated

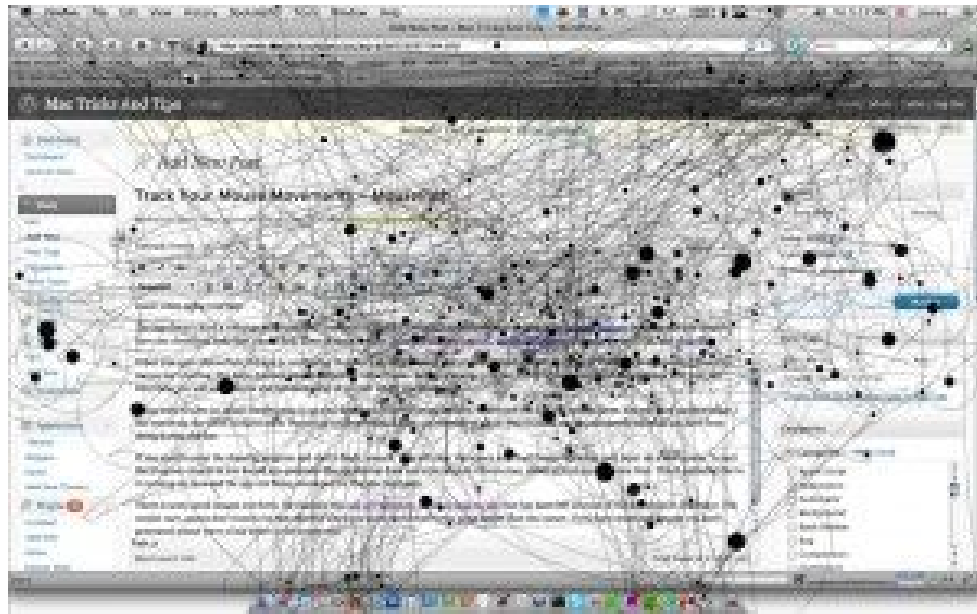
For $F > 90$

75 \leq BS \leq 100: Rage

BS < 75: Hulk Smash

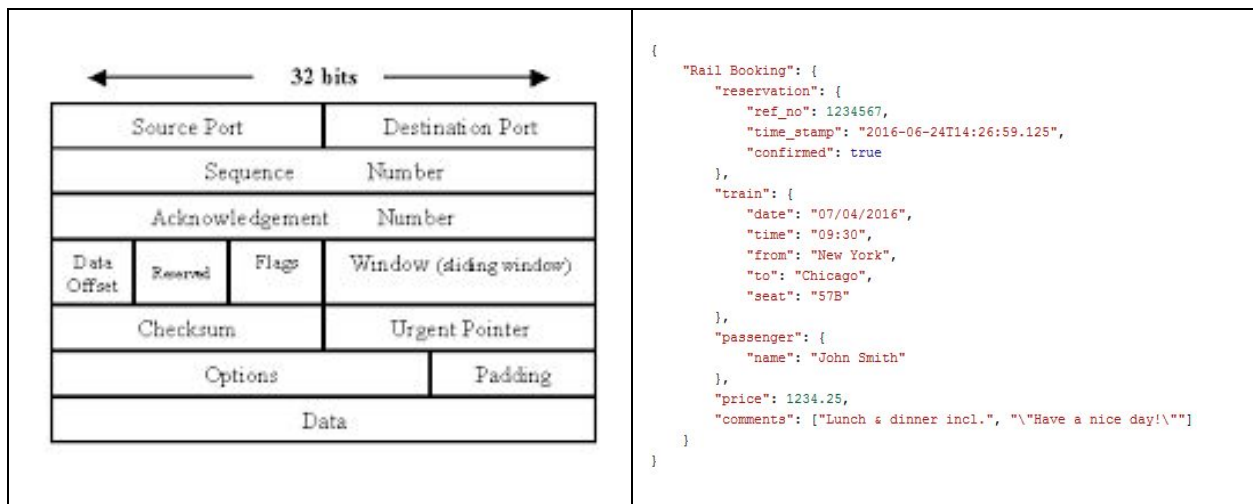


For GUI applications, inputs may also be mouse movements, locations or clicks, manual (touch-sensitive) gesture, camera images, gps coordinates, etc.



What might be some valid and invalid equivalence classes for mouse activity?

System-to-system inputs (or among system components) are likely to be specially formatted, e.g., network packets, JSON



Each field has its own equivalence partitions, as well as higher-level correctly formatted vs. not (or multiple correct formats if alternatives supported)

Equivalence classes may cut across multiple inputs:

- Sometimes the value of one input restricts appropriate values for other inputs
- For example, the checksum in the network packet

At the full system and component levels, inputs are generally going to be numbers, strings (text), tuples (packets), and files.

File inputs can:

- Exist or not
- Be below min or above max size
- Be readable, writeable, executable by current user (permissions)
- Correct, corrupted, garbled, etc. data and metadata formats - images, audio, video, ...

“Assignment T1: Preliminary Project Proposal” due tonight
<https://courseworks2.columbia.edu/courses/104335/assignments/486922>

“Assignment T2: Revised Project Proposal” due next week
<https://courseworks2.columbia.edu/courses/104335/assignments/486956>

You must meet with your primary IA mentor *before* submitting! The entire team should try to attend, but it’s better for a subset of your team to meet with your mentor asap than to wait for more convenient scheduling.

First team meetings with mentors will be during class time on Thursday