Testing (Part 2/3)

Martin Kellogg

Testing (part 2)

Today's agenda:

- **Reading Quiz**
- Test quality
- Test suite quality
  - lens of logic: coverage
  - lens of statistics: testing on real users
  - lens of adversity: mutation testing

Reading Quiz: testing (part 2)

Q1: **TRUE** or **FALSE**: The author considers it unproductive to mutate the condition of an if statement if the statement's the body is arid (e.g., a logging statement).

Q2: When does the author's mutation testing tool surface unkilled mutants to the developer?

- **A.** whenever the tests run
- **B.** at code review time
- **C.** in the developer's IDE

Reading Quiz: testing (part 2)

Q1: **TRUE** or **FALSE**: The author considers it unproductive to mutate the condition of an if statement if the statement's the body is arid (e.g., a logging statement).

Q2: When does the author's mutation testing tool surface unkilled mutants to the developer?

- **A.** whenever the tests run
- **B.** at code review time
- **C.** in the developer's IDE

Reading Quiz: testing (part 2)

Q1: **TRUE** or **FALSE**: The author considers it unproductive to mutate the condition of an if statement if the statement's the body is arid (e.g., a logging statement).

Q2: When does the author's mutation testing tool surface unkilled mutants to the developer?

- **A.** whenever the tests run
- **B.** at code review time
- **C.** in the developer's IDE

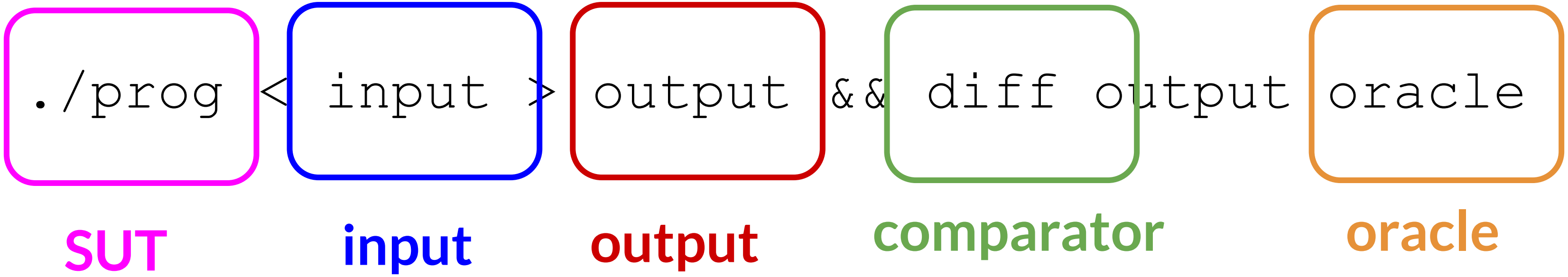
Testing (part 2)

Today’s agenda:

- Reading Quiz
- **Test quality**
- Test suite quality
  - lens of logic: coverage
  - lens of statistics: testing on real users
  - lens of adversity: mutation testing

Review: parts of a test

Definition: a *test* executes a **given input** on a program (the *system under test* or *SUT*) and **compares** the SUT's **output** to a given **oracle**



```
./prog < input > output && diff output oracle
```

SUT

input

output

comparator

oracle

Test quality: what makes a test good or bad?

Test quality: what makes a test good or bad?

Good	Bad

Test quality: what makes a test good or bad?

Good	**Bad**
isolated (only tests one thing)	brittle
runs quickly	slow
strong oracle	weak oracle
hermetic	redundant
easy to understand	hard to understand (“mystery”)
deterministic	non-deterministic (“flaky”)
etc.	etc.

Test quality: what makes a test good or bad?

Good

- isolated (only tests one thing)
- runs quickly
- strong oracle
- **hermetic**
- easy to understand
- deterministic
- etc.

Bad

- **brittle**
- slow
- weak oracle
- redundant
- hard to understand ("**mystery**")
- non-deterministic ("**flaky**")
- etc.

“Hermetic” tests

Definition: a ***hermetic*** test is fully self-contained: its behavior doesn’t depend on anything except the test itself and the SUT

“Hermetic” tests

Definition: a ***hermetic*** test is fully self-contained: its behavior doesn’t depend on anything except the test itself and the SUT

- avoid dependencies on the environment (e.g., software installed on the machine, environment variables, contents of other files, operating system behaviors, etc.)
- being hermetic is also important for builds generally (we’ll discuss more in our lecture on build systems later this semester)

Brittle tests

Definition: a ***brittle*** test fails for reasons unrelated to what it ostensibly tests

Brittle tests

Definition: a ***brittle*** test fails for reasons unrelated to what it ostensibly tests

- common causes:
  - not being hermetic
  - testing too much at once
  - comparator or oracle is too specific

Mystery tests

Definition: a ***mystery*** test fails for reasons that are not immediately clear

Mystery tests

Definition: a ***mystery*** test fails for reasons that are not immediately clear

- commonly **co-occurs with brittleness**: test is brittle because it is too complicated, and when it fails it’s not clear why
    - especially common for very large, end-to-end tests

Mystery tests

Definition: a ***mystery*** test fails for reasons that are not immediately clear

- commonly **co-occurs with brittleness**: test is brittle because it is too complicated, and when it fails it's not clear why
    - especially common for very large, end-to-end tests
- **best practice**: tests should give as much information as possible when they fail
    - **implication**: when writing tests, think about why they might fail in the future and document that in the test itself

Flaky tests

Definition: a ***flaky*** test fails non-deterministically: that is, they sometimes pass and sometimes fail

Flaky tests

Definition: a ***flaky*** test fails non-deterministically: that is, they sometimes pass and sometimes fail

- sometimes caused by brittleness (e.g., relying on the network)

Flaky tests

Definition: a ***flaky*** test fails non-deterministically: that is, they sometimes pass and sometimes fail

- sometimes caused by brittleness (e.g., relying on the network)
- sometimes caused by **non-determinism in the program** itself
  - e.g., relying on randomness, iteration order of hashtables, etc.

Flaky tests

Definition: a *flaky* test fails non-deterministically: that is, they sometimes pass and sometimes fail

- sometimes caused by brittleness (e.g., relying on the network)
- sometimes caused by **non-determinism in the program** itself
    - e.g., relying on randomness, iteration order of hashtables, etc.
- are a **major problem in practice**
    - difficult to debug, so waste a lot of developer time
    - detecting them is an active research area

Testing (part 2)

Today's agenda:

- Reading Quiz
- Test quality
- **Test suite quality**
  - lens of logic: coverage
  - lens of statistics: testing on real users
  - lens of adversity: mutation testing

Test suite quality

- We’ve talked about what makes individual test cases good or bad

Test suite quality

- We’ve talked about what makes individual test cases good or bad
- However, programs typically have **more than one** test

Test suite quality

- We’ve talked about what makes individual test cases good or bad
- However, programs typically have **more than one** test

Definition: a ***test suite*** is a collection of tests for the same program

Test suite quality

- We’ve talked about what makes individual test cases good or bad
- However, programs typically have **more than one** test

Definition: a ***test suite*** is a collection of tests for the same program

Question: what makes one test suite **better or worse** than another?

Test suite quality

- We’ve talked about what makes individual test cases good or bad
- However, programs typically have **more than one** test

Definition: a ***test suite*** is a collection of tests for the same program

Question: what makes one test suite **better or worse** than another?

- not just the sum of the “goodness” of all the individual tests!

Test suite quality: who cares?

Why would we want to evaluate the quality of a test suite?

Test suite quality: who cares?

Why would we want to evaluate the quality of a test suite?

- testing is **expensive** (e.g., 35% of total IT spending according to Capgemini World Quality Report, 2015)

Test suite quality: who cares?

Why would we want to evaluate the quality of a test suite?

- testing is **expensive** (e.g., 35% of total IT spending according to Capgemini World Quality Report, 2015)
- we want to direct our resources **efficiently**
  - i.e., avoid writing new tests if we already have a good test suite

Test suite quality: who cares?

Why would we want to evaluate the quality of a test suite?

- testing is **expensive** (e.g., 35% of total IT spending according to Capgemini World Quality Report, 2015)
- we want to direct our resources **efficiently**
  - i.e., avoid writing new tests if we already have a good test suite
- we want to know how much confidence our tests give us
  - ideal world: all tests pass = software is 100% correct

Test suite quality: who cares?

Why would we want to evaluate the quality of a test suite?

- testing is **expensive** (e.g., 35% of total IT spending according to Capgemini World Quality Report, 2015)
- we want to direct our resources **efficiently**
  - i.e., avoid writing new tests if we already have a good test suite
- we want to know how much confidence our tests give us
  - ideal world: all tests pass = software is 100% correct
- sometimes, we may not even have enough resources to run all tests
  - we'll discuss test suite minimization next time

Ways to think about test suite quality

Today we’re going to consider three ways to think about test suite quality:

- test suite quality through the lens of **logic**
- test suite quality through the lens of **statistics**
- test suite quality through the lens of **adversity**

The Lens of Logic

Informally, we want the following property:

- The program passes the tests if and only if it does **all the right things** and **none of the wrong things**.

The Lens of Logic

Informally, we want the following property:

- The program passes the tests if and only if it does **all the right things** and **none of the wrong things**.
    - Pass all tests → program adheres to requirements
    - Each failing test → program behaves incorrectly

The Lens of Logic: intuition

- Suppose you were writing a sqrt program and one of the requirements was that it should abort gracefully on negative inputs.

The Lens of Logic: intuition

- Suppose you were writing a sqrt program and one of the requirements was that it should abort gracefully on negative inputs.
- Suppose further that your test suite does not include any negative inputs.

The Lens of Logic: intuition

- Suppose you were writing a sqrt program and one of the requirements was that it should abort gracefully on negative inputs.
- Suppose further that your test suite does not include any negative inputs.
- Can we conclude that passing all of the tests implies adhering to all of the requirements?

The Lens of Logic: coverage

- We desire all of the requirements to be covered (“checked”) by the test suite.

The Lens of Logic: coverage

- We desire all of the requirements to be covered (“checked”) by the test suite.
- For our purposes, ***X* coverage** is the degree to which ***X*** is executed/exercised by the test suite.

The Lens of Logic: coverage

- We desire all of the requirements to be covered (“checked”) by the test suite.
- For our purposes, **X coverage** is the degree to which **X** is executed/exercised by the test suite.
- **Code coverage** is the degree to which the source code is executed by the test suite.

The Lens of Logic: coverage

- We desire all of the requirements to be covered (“checked”) by the test suite.
- For our purposes, ***X* coverage** is the degree to which ***X*** is executed/exercised by the test suite.
- **Code coverage** is the degree to which the source code is executed by the test suite.
    - How do we actually **measure** code coverage?

Coverage: statement coverage

Definition: ***Statement coverage*** is the fraction of source statements that are executed by the test suite.

Coverage: statement coverage

Definition: ***Statement coverage*** is the fraction of source statements that are executed by the test suite.

- **Key Logical Observation**: If we **never test** line X then testing **cannot rule out** the presence of a bug on line X

Coverage: statement coverage

Definition: ***Statement coverage*** is the fraction of source statements that are executed by the test suite.

- **Key Logical Observation**: If we **never test** line X then testing **cannot rule out** the presence of a bug on line X
- Example: if our test executes lines 1 and 2, but there is a bug on line 3, there is **no way** that our test will find the bug!

Aside: “don’t do bad things”

- We can test that programs **do not do certain bad things**
    - e.g., “don't segfault”, “don't send my password to Microsoft”, “on this one particular input, don't get the wrong answer”
- Note that “I never do bad things” is not the same as “I always/eventually do good things”
    - For more information, take a class on *Modal Logic* or read about *Liveness* vs. *Safety properties*

Coverage: statement coverage

Implication for statement coverage: you could test line X and still have a bug on line X

- e.g., foo(a,b) { return a/b; }
- test: foo(6,2) does not throw DivideByZeroException

Coverage: statement coverage

Implication for statement coverage: you could test line X and still have a bug on line X

- e.g., foo(a,b) { return a/b; }
- test: foo(6,2) does not throw DivideByZeroException

But testing line X gives us some **small but non-zero confidence** in the correctness of line X

Coverage: statement coverage: assumptions

We’ve made some **assumptions** in our discussion of statement coverage so far:

Coverage: statement coverage: assumptions

We’ve made some **assumptions** in our discussion of statement coverage so far:

- We gain the same amount of confidence (or information) for each visited line
- The amount of confidence (or information) we gain per visited line is positive
- …

Coverage: computing statement coverage

Coverage: computing statement coverage

- At its simplest, this is just **print-statement debugging**

Coverage: computing statement coverage

- At its simplest, this is just **print-statement debugging**
- Put a print statement before every line of the program
    - Run all the tests, collect all the printed information, remove duplicates, count

Coverage: computing statement coverage

- At its simplest, this is just **print-statement debugging**
- Put a print statement before every line of the program
    - Run all the tests, collect all the printed information, remove duplicates, count
- Practical concern: the **observer effect** (from physics) is the fact that simply observing a situation or phenomenon necessarily changes that phenomenon.

Coverage: computing statement coverage

- At its simplest, this is just **print-statement debugging**
- Put a print statement before every line of the program
  - Run all the tests, collect all the printed information, remove duplicates, count
- Practical concern: the **observer effect** (from physics) is the fact that simply observing a situation or phenomenon necessarily changes that phenomenon.
  - Implication for computing statement coverage: program might depend on timing info, amount of I/O, etc.

Coverage: computing statement coverage

Definition: ***Coverage instrumentation*** modifies a program to record coverage information in a way that minimizes the observer effect.

Coverage: computing statement coverage

Definition: ***Coverage instrumentation*** modifies a program to record coverage information in a way that minimizes the observer effect.

- This can be done at the source or binary level.
- Don't actually print to stdout/stderr
- Don't slow things down too much
    - Pre-check before printing a duplicate?
- Don't introduce infinite loops
    - Instrument "print" with a call to "print"?

Coverage: computing statement coverage

Definition: ***Coverage instrumentation*** mo
coverage information in a way that minin

- This can be done at the source or bin
- Don't actually print to stdout/stderr
- Don't slow things down too much
    - Pre-check before printing a duplicate?
- Don't introduce infinite loops
    - Instrument "print" with a call to "print"?

Good news: coverage instrumentation is a "solved" problem:

- e.g., Jest does it automatically

Coverage: limitations of statement coverage

- As we’ve seen, executing every line doesn’t guarantee no bugs

Coverage: limitations of statement coverage

- As we’ve seen, executing every line doesn’t guarantee no bugs
- Not only that, but executing every line doesn’t even guarantee that we cover all of the program’s **behaviors**

Coverage: limitations of statement coverage

- As we’ve seen, executing every line doesn’t guarantee no bugs
- Not only that, but executing every line doesn’t even guarantee that we cover all of the program’s **behaviors**
    - many behaviors are dependent on data that causes particular **control flows**: that is, that cause different branches of conditionals to be executed

Coverage: limitations of statement coverage

- As we've seen, executing every line doesn't guarantee no bugs
- Not only that, but executing every line doesn't even guarantee that we cover all of the program's **behaviors**
    - many behaviors are dependent on data that causes particular **control flows**: that is, that cause different branches of conditionals to be executed
- Informally, the problem of ensuring that we cover interesting data values may **reduce** to the problem of ensuring that we **cover all branches** of conditionals

Aside: reductions

Quick poll: raise your hand if you have ever **reduced** one problem to another

- examples: reducing something to the halting problem to show that it is not computable; reducing something to satisfiability to show that it is NP-hard
- should be covered in a theory of computation class (likely near the end of the semester)

Aside: reductions

Quick poll: raise your hand if you have ever **reduced** one problem to another

- examples: reducing something to the halting problem to show that it is not computable; redu
show that it is NP-hard
- should be covered in a theory
the end of the semester)

Reduction is a **powerful tool** for thinking about problems: it lets you solve difficult problems indirectly by re-using solutions for other, related problems.

Coverage: branch coverage

Definition: ***Branch coverage*** is a test suite quality metric that counts the total number of conditional branches exercised by that test suite (i.e., if true and if false are counted separately)

Coverage: branch coverage

Definition: ***Branch coverage*** is a test suite quality metric that counts the total number of conditional branches exercised by that test suite (i.e., if true and if false are counted separately)

Note that branch coverage can subsume line coverage:

```
foo(a):
  if a > 5:
    print “x”
  print “y”
```

Coverage: branch coverage

Definition: ***Branch coverage*** is a test suite quality metric that counts the total number of conditional branches exercised by that test suite (i.e., if true and if false are counted separately)

Note that branch coverage can subsume line coverage:

```
foo(a):
  if a > 5:
    print “x”
  print “y”
```

Test Suite { `foo(7)` } has 100% line coverage but 50% branch coverage.

Coverage: branch coverage

Definition: *Branch coverage* is a test suite quality metric that counts the total number of conditional branches exercised by that test suite (i.e., if true and if false are counted separately)

Note that branch coverage can subsume line coverage:

```
foo(a):
  if a > 5:
    print “x”
  print “y”
```

Test Suite { `foo(7), foo(4)`} has 100% line coverage and 100% branch coverage.

Coverage: branch vs statement coverage

Coverage: branch vs statement coverage

- Branch coverage typically gives us **more confidence** than line coverage

Coverage: branch vs statement coverage

- Branch coverage typically gives us **more confidence** than line coverage
    - Typically, 100% branch coverage **implies** 100% line coverage

Coverage: branch vs statement coverage

- Branch coverage typically gives us **more confidence** than line coverage
    - Typically, 100% branch coverage **implies** 100% line coverage
- However, branch coverage is "**more expensive**" in the sense that it is harder for a test suite to have high branch coverage than to have high line coverage

Coverage: branch vs statement coverage

- Branch coverage typically gives us **more confidence** than line coverage
    - Typically, 100% branch coverage **implies** 100% line coverage
- However, branch coverage is “**more expensive**” in the sense that it is harder for a test suite to have high branch coverage than to have high line coverage
    - Note: quality isn't really “more expensive”, you were just fooling yourself before by thinking line coverage was OK. Being correct is expensive.

Coverage: other kinds of coverage

- **Function Coverage**: what fraction of functions have been called?
- **Condition Coverage**: what fraction of boolean subexpressions have been evaluated to both true and also (e.g., on another run) to false?
  - Comparing this to branch coverage is a not-uncommon test question …
- **Modified Condition / Decision Coverage** (MC/DC): function coverage + branch coverage (this is a simplification)
  - Used in mission critical (e.g., avionics) software

Ways to think about test suite quality

Today we're going to consider three ways to think about test suite quality:

- test suite quality through the lens of **logic**
- test suite quality through the lens of **statistics**
- test suite quality through the lens of **adversity**

The Lens of Statistics: intuition

The Lens of Statistics: intuition

- The bugs experienced **by users** are the ones that matter.

The Lens of Statistics: intuition

- The bugs experienced **by users** are the ones that matter.
    - Dually, bugs never experienced by users do not matter.

The Lens of Statistics: intuition

- The bugs experienced **by users** are the ones that matter.
  - Dually, bugs never experienced by users do not matter.
- If user-experienced bugs are the ones that matter, testing should be devoted to **sampling** those inputs that users will provide

The Lens of Statistics: intuition

- The bugs experienced **by users** are the ones that matter.
  - Dually, bugs never experienced by users do not matter.
- If user-experienced bugs are the ones that matter, testing should be devoted to **sampling** those inputs that users will provide
- Two views:

The Lens of Statistics: intuition

- The bugs experienced **by users** are the ones that matter.
    - Dually, bugs never experienced by users do not matter.
- If user-experienced bugs are the ones that matter, testing should be devoted to **sampling** those inputs that users will provide
- Two views:
    - Sample what users do **most commonly**

The Lens of Statistics: intuition

- The bugs experienced **by users** are the ones that matter.
    - Dually, bugs never experienced by users do not matter.
- If user-experienced bugs are the ones that matter, testing should be devoted to **sampling** those inputs that users will provide
- Two views:
    - Sample what users do **most commonly**
    - Sample what causes the **most harm** if users do it

The Lens of Statistics: intuition

- The bugs experienced **by users** are the ones that matter.
    - Dually, bugs never experienced by users do not matter.
- If user-experienced bugs are the ones that matter, testing should be devoted to **sampling** those inputs that users will provide
- Two views:
    - Sample what users do **most commonly**
    - Sample what causes the **most harm** if users do it
- Compare:
    - Risk = (Probability of Event) * (Damage if Event Occurs)

Example: limited input domain

- Suppose you are writing a point-of-sale cashier application that makes change for a dollar. Given any price between 1 and 100 cents, you must indicate the coins to give out as change.
    - e.g., 23 → return 3 quarters and 2 pennies

Example: limited input domain

- Suppose you are writing a point-of-sale cashier application that makes change for a dollar. Given any price between 1 and 100 cents, you must indicate the coins to give out as change.
    - e.g., 23 → return 3 quarters and 2 pennies
- In this scenario, you can **exhaustively test** all 100 inputs that will occur to real users in the real world
    - In some sense, it does not matter if that is 100% statement or code coverage (e.g., dead code): your testing is still exhaustive of the inputs that will matter in the real world

Limitations on users in the real world

- Usually, in the real world, your input domain isn’t so limited
- But, you might still be able to say:

Limitations on users in the real world

- Usually, in the real world, your input domain isn’t so limited
- But, you might still be able to say:
    - Suppose users will only ever cause lines 1, 2 and 3 of your program to be executed

Limitations on users in the real world

- Usually, in the real world, your input domain isn’t so limited
- But, you might still be able to say:
    - Suppose users will only ever cause lines 1, 2 and 3 of your program to be executed
    - If you can be sure of this, then there is no need to test line 4

Limitations on users in the real world

- Usually, in the real world, your input domain isn’t so limited
- But, you might still be able to say:
    - Suppose users will only ever cause lines 1, 2 and 3 of your program to be executed
    - If you can be sure of this, then there is no need to test line 4
        - Aside: why do you have line 4?

Limitations on users in the real world

- Usually, in the real world, your input domain isn’t so limited
- But, you might still be able to say:
  - Suppose users will only ever cause lines 1, 2 and 3 of your program to be executed
  - If you can be sure of this, then there is no need to test line 4
    - Aside: why do you have line 4?
  - Even if line 4 has a bug, users will **never** encounter it

Limitations on users in the real world

- Usually, in the real world, your input domain isn't so limited
- But, you might still be able to say:
  - Suppose users will only ever cause lines 1, 2 and 3 of your program to be executed
  - If you can be sure of this, then there is no need to test line 4
    - Aside: why do you have line 4?
  - Even if line 4 has a bug, users will **never** encounter it
- Note "will": this either requires a **prediction of the future** or a **finite input domain**

The Lens of Statistics

- **Key idea:** Sample test inputs from the population of inputs users will actually provide in the real world

The Lens of Statistics

- **Key idea:** Sample test inputs from the population of inputs users will actually provide in the real world
    - This approach inherits both advantages and disadvantages from other kinds of statistical techniques

The Lens of Statistics

- **Key idea:** Sample test inputs from the population of inputs users will actually provide in the real world
    - This approach inherits both advantages and disadvantages from other kinds of statistical techniques

Key advantages:

- confidence that tests are indicative of the real world
- can use statistical techniques to estimate the chance that our tests don't cover some important behavior

The Lens of Statistics: disadvantages

The Lens of Statistics: disadvantages

- In statistics, **sampling error** is incurred when the statistical characteristics of a population are estimated from a subset, or sample, of that population.

The Lens of Statistics: disadvantages

- In statistics, **sampling error** is incurred when the statistical characteristics of a population are estimated from a subset, or sample, of that population.
  - “Our test suite is a sample of inputs that could occur in the real world. Our program behaves well on our test suite.”

The Lens of Statistics: disadvantages

- In statistics, **sampling error** is incurred when the statistical characteristics of a population are estimated from a subset, or sample, of that population.
    - “Our test suite is a sample of inputs that could occur in the real world. Our program behaves well on our test suite.” → later →

The Lens of Statistics: disadvantages

- In statistics, **sampling error** is incurred when the statistical characteristics of a population are estimated from a subset, or sample, of that population.
  - "Our test suite is a sample of inputs that could occur in the real world. Our program behaves well on our test suite." → later → "Our program behaves badly on some other untested real input. Sampling error!"

The Lens of Statistics: disadvantages

- In statistics, **sampling error** is incurred when the statistical characteristics of a population are estimated from a subset, or sample, of that population.
  - “Our test suite is a sample of inputs that could occur in the real world. Our program behaves well on our test suite.” → later → “Our program behaves badly on some other untested real input. Sampling error!”
- Testing gives confidence the same way sampling (or polling) gives confidence.

The Lens of Statistics: disadvantages

- In statistics, **sampling bias** is a bias in which a sample is collected in such a way that some members of the intended population are less likely to be included than others.

The Lens of Statistics: disadvantages

- In statistics, **sampling bias** is a bias in which a sample is collected in such a way that some members of the intended population are less likely to be included than others.
  - Suppose you are conducting a poll to see who will win the next election, but you only poll republicans.

The Lens of Statistics: disadvantages

- In statistics, **sampling bias** is a bias in which a sample is collected in such a way that some members of the intended population are less likely to be included than others.
  - Suppose you are conducting a poll to see who will win the next election, but you only poll republicans.
  - Suppose you are creating tests to see if your program will crash, but you only poll nice, small, inputs.

The Lens of Statistics: disadvantages

- Possible solution: there are a number of **well-established sampling techniques** in the field of statistics to help address such biases

The Lens of Statistics: disadvantages

- Possible solution: there are a number of **well-established sampling techniques** in the field of statistics to help address such biases
    - Unfortunately, they often require knowing something about the **distribution** of the full population from which you want to sample a subpopulation

The Lens of Statistics: disadvantages

- Possible solution: there are a number of **well-established sampling techniques** in the field of statistics to help address such biases
    - Unfortunately, they often require knowing something about the **distribution** of the full population from which you want to sample a subpopulation
- The basic problem in SE is that the underlying distribution of real user inputs is **not known**

The Lens of Statistics: practical options

The Lens of Statistics: practical options

Definition: ***Beta testing*** is testing done by external users (often using a special beta version of the program).

The Lens of Statistics: practical options

Definition: ***Beta testing*** is testing done by external users (often using a special beta version of the program).

- in contrast to ***alpha testing***, which is usually performed by developers or a quality assurance team

The Lens of Statistics: practical options

Definition: ***Beta testing*** is testing done by external users (often using a special beta version of the program).

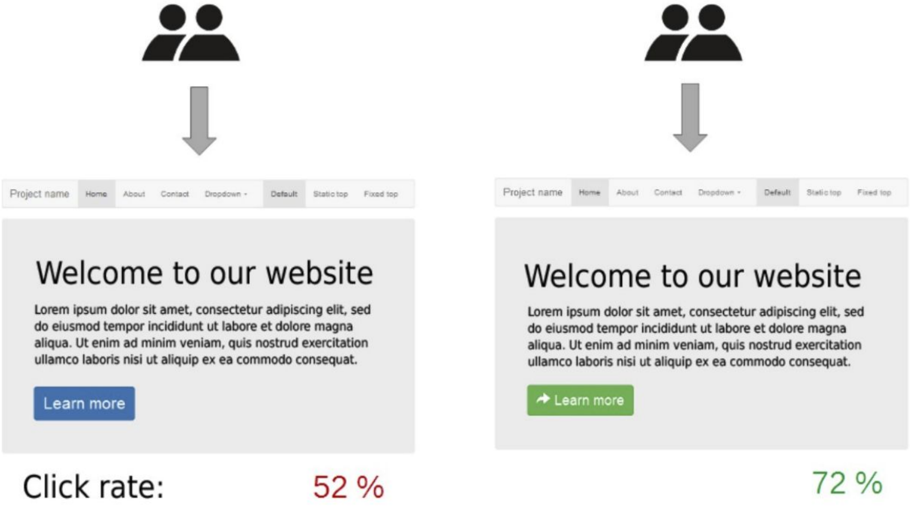
- in contrast to ***alpha testing***, which is usually performed by developers or a quality assurance team
- Beta testing can be viewed as directly sampling the space of user inputs

The Lens of Statistics: practical options

Definition: ***A/B testing*** involves two variants of your software, A and B, which differ only in one feature. Different users are shown different variants and responses are recorded.

The Lens of Statistics: practical options

Definition: ***A/B testing*** involves two variants of your software, A and B, which differ only in one feature. Different users are shown different variants and responses are recorded.



Welcome to our website

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

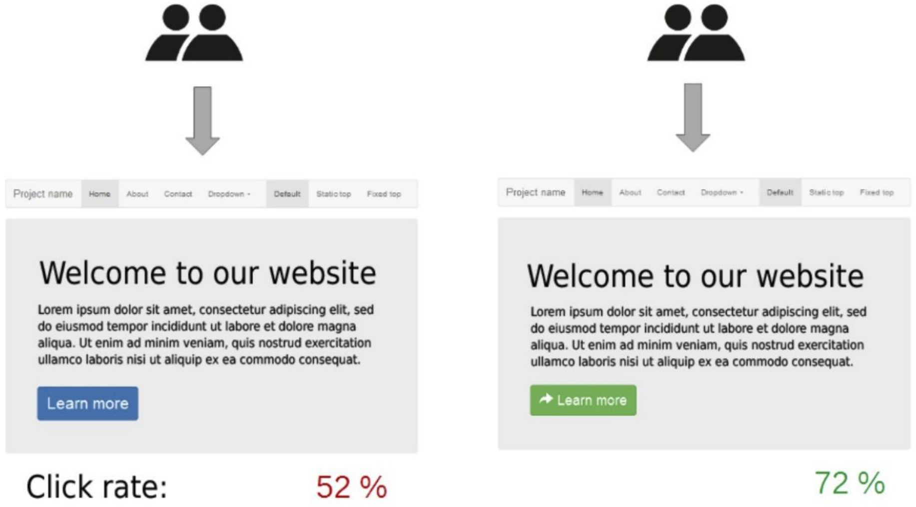
Welcome to our website

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

The Lens of Statistics: practical options

Definition: ***A/B testing*** involves two variants of your software, A and B, which differ only in one feature. Different users are shown different variants and responses are recorded.

- A/B testing is an instance of two-sample hypothesis testing, like you'd encounter in a statistics class.



Welcome to our website

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Welcome to our website

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

The Lens of Statistics: practical options

- Recall two guiding approaches:
    - Sample what users will do **most commonly**
    - Sample what will cause the **most harm**

The Lens of Statistics: practical options

- Recall two guiding approaches:
  - Sample what users will do **most commonly**
  - Sample what will cause the **most harm**
- The former is sometimes called ***workload generation***
  - Common for databases, webservers, etc.

The Lens of Statistics: practical options

- Recall two guiding approaches:
    - Sample what users will do **most commonly**
    - Sample what will cause the **most harm**
- The former is sometimes called ***workload generation***
    - Common for databases, webservers, etc.
- The latter often relates to **computer security**
    - E.g., exploit generation, penetration testing, etc.

The Lens of Statistics: practical options

- Recall two guiding approaches:
    - Sample what users will do **most commonly**
    - Sample what will cause the **most harm**
- The former is sometimes called ***workload generation***
    - Common for databases, webservers, etc.
- The latter often relates to **computer security**
    - E.g., exploit generation, penetration testing, etc.
- **Damage** can also be in other forms
    - e.g., for Amazon, “damage” might be “customer doesn’t complete the purchase”

Ways to think about test suite quality

Today we’re going to consider three ways to think about test suite quality:

- test suite quality through the lens of **logic**
- test suite quality through the lens of **statistics**
- test suite quality through the lens of **adversity**

The Lens of Adversity: finding bugs

- Suppose you wanted to evaluate the quality of two truffle-sniffing pigs

The Lens of Adversity: finding bugs

- Suppose you wanted to evaluate the quality of two truffle-sniffing pigs
    - **Intuition**: test whether they can actually find truffles!

The Lens of Adversity: finding bugs

- Suppose you wanted to evaluate the quality of two truffle-sniffing pigs
    - **Intuition**: test whether they can actually find truffles!
- **Test idea**: hide some truffles in your backyard and see how many each pig finds

The Lens of Adversity: finding bugs

- Suppose you wanted to evaluate the quality of two truffle-sniffing pigs
  - **Intuition**: test whether they can actually find truffles!
- **Test idea**: hide some truffles in your backyard and see how many each pig finds
  - The pig that finds more of the hidden truffles in your backyard is assumed to find more real truffles in the wild

The Lens of Adversity: finding bugs

- Suppose you wanted to evaluate the quality of two truffle-sniffing pigs
  - **Intuition**: test whether they can actually find truffles!
- **Test idea**: hide some truffles in your backyard and see how many each pig finds
  - The pig that finds more of the hidden truffles in your backyard is assumed to find more real truffles in the wild
- Suppose you wanted to evaluate the quality of two bug-finding test suites …

The Lens of Adversity: mutation testing

Definition: ***Mutation testing*** (or ***mutation analysis***) is a test suite adequacy metric in which the quality of a test suite is related to the number of intentionally-added defects it finds

The Lens of Adversity: mutation testing

Definition: ***Mutation testing*** (or ***mutation analysis***) is a test suite adequacy metric in which the quality of a test suite is related to the number of intentionally-added defects it finds

- Informally: “You claim your test suite is really great at finding security bugs? Well, I'll just **intentionally add a bug** to my source code and see if your test suite finds it!”

Mutation testing: verisimilitude

- In the truffle-pig example, if every truffle I hide in my backyard is next to a smelly red flower, a pig that finds them all may not actually do well in the real world

Mutation testing: verisimilitude

- In the truffle-pig example, if every truffle I hide in my backyard is next to a smelly red flower, a pig that finds them all may not actually do well in the real world
  - The truffle placements I made up were **not indicative** of real-world truffles

Mutation testing: verisimilitude

- In the truffle-pig example, if every truffle I hide in my backyard is next to a smelly red flower, a pig that finds them all may not actually do well in the real world
  - The truffle placements I made up were **not indicative** of real-world truffles
- Similarly, if I add a bunch of defects to my software that are not the sort of defects real humans would make, then mutation testing is **uninformative**

Mutation testing: verisimilitude

- In the truffle-pig example, if every truffle I hide in my backyard is next to a smelly red flower, a pig that finds them all may not actually do well in the real world
  - The truffle placements I made up were **not indicative** of real-world truffles
- Similarly, if I add a bunch of defects to my software that are not the sort of defects real humans would make, then mutation testing is **uninformative**
  - **Implication**: mutation testing requires us to know what real bugs look like

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

- The defect introduced is typically intentionally similar to defects introduced by real developers.

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

- The defect introduced is typically intentionally similar to defects introduced by real developers.
- The seeding is typically done by changing the source code.

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

- The defect introduced is typically intentionally similar to defects introduced by real developers.
- The seeding is typically done by changing the source code.
- For mutation testing, defect seeding is typically done automatically (given a model of what human bugs look like)

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

- The defect introduced is typically intentionally similar to defects introduced by real developers.
- The seeding is typically done b ode.
- For mutation testing, defect se automatically (given a model c like)

This is **exactly** how our "fault injection" system for testing your IP1&2 tests works.

Mutation testing: mutation operators

Definition: A ***mutation operator*** systematically changes a program point. In mutation testing, the mutation operators are modeled on historical human defects.

Mutation testing: mutation operators

Definition: A ***mutation operator*** systematically changes a program point. In mutation testing, the mutation operators are modeled on historical human defects.

- Example mutations:
    - `if (a < b)` → `if (a <= b)`
    - `if (a == b)` → `if (a != b)`
    - `a = b + c` → `a = b - c`
    - `f(); g();` → `g(); f();`
    - `x = y` → `x = z`

Mutation testing: mutants

Definition: A ***mutant*** (or ***variant***) is a version of the original program produced by applying one or more mutation operators to one or more program locations.

Mutation testing: mutants

Definition: A ***mutant*** (or ***variant***) is a version of the original program produced by applying one or more mutation operators to one or more program locations.
Definition: The ***order*** of a mutant is the number of mutation operators applied.

Mutation testing: mutants

Definition: A ***mutant*** (or ***variant***) is a version of the original program produced by applying one or more mutation operators to one or more program locations.

Definition: The ***order*** of a mutant is the number of mutation operators applied.

```
// original
if (a < b):
x = a + b
print(x)
```

→

```
  // 2nd-order mutant
if (a <= b):
  x = a - b
print(x)
```

Mutation testing: competent programmers

- The **competent programmer hypothesis** holds that program faults are syntactically small and can be corrected with a few keystrokes.

Mutation testing: competent programmers

- The **competent programmer hypothesis** holds that program faults are syntactically small and can be corrected with a few keystrokes.
    - Programmers write programs that are largely correct. Thus the mutants simulate the likely effect of real faults.

Mutation testing: competent programmers

- The **competent programmer hypothesis** holds that program faults are syntactically small and can be corrected with a few keystrokes.
    - Programmers write programs that are largely correct. Thus the mutants simulate the likely effect of real faults.
    - Therefore, **if the test suite is good at catching the artificial mutants, it will also be good at catching the unknown but real faults** in the program.

Mutation testing: competent programmers

- The **competent programmer hypothesis** holds that program faults are syntactically small and can be corrected with a few keystrokes
  - P . Thus t
  - T **ificial r n but r**

Is the competent programmer hypothesis true?

- Yes and no.
- It is true that humans often make simple typos (e.g., + vs -)
- But it is also true that some bugs are much more complex than that.

Mutation testing: competent programmers

- The **competent programmer hypothesis** holds that program faults are syntactically small and can be corrected with a few keystrokes
  - P . Thus t
  - T **ificial r n but r**

Is the competent programmer hypothesis true?

- Yes and no.
- It is true that humans often make simple typos (e.g., + vs -).
- But it is also true that some bugs are much **more complex** than that!

Mutation testing: coupling effect

- The **coupling effect hypothesis** holds that complex faults are “coupled” to simple faults in such a way that a test suite that detects all simple faults in a program will detect a high percentage of the complex faults.

Mutation testing: coupling effect

- The **coupling effect hypothesis** holds that complex faults are “coupled” to simple faults in such a way that a test suite that detects all simple faults in a program will detect a high percentage of the complex faults.
- Is this true?

Mutation testing: coupling effect

- The **coupling effect hypothesis** holds that complex faults are “coupled” to simple faults in such a way that a test suite that detects all simple faults in a program will detect a high percentage of the complex faults.
- Is this true?
    - Tests that detect simple mutants were **also** able to detect over 99% of second- and third-order mutants historically

[A. J. Offutt. Investigations of the software testing coupling effect. ACM Trans. Softw. Eng. Methodol., 1(1):5–20, Jan. 1992.]

Mutation testing: putting it all together

- A test suite is said to ***kill*** (or ***detect***, or ***reveal***) a mutant if the mutant fails a test that the original passes.

Mutation testing: putting it all together

- A test suite is said to ***kill*** (or ***detect***, or ***reveal***) a mutant if the mutant fails a test that the original passes.
- ***Mutation testing*** (or ***mutation analysis***) of a test suite proceeds by making a number of mutants and measuring the fraction of them killed by that test suite. This fraction is called the ***mutation adequacy score*** (or ***mutation score***).

Mutation testing: putting it all together

- A test suite is said to ***kill*** (or ***detect***, or ***reveal***) a mutant if the mutant fails a test that the original passes.
- ***Mutation testing*** (or ***mutation analysis***) of a test suite proceeds by making a number of mutants and measuring the fraction of them killed by that test suite. This fraction is called the ***mutation adequacy score*** (or ***mutation score***).
  - A test suite with a **higher score is better**.

Mutation testing: putting it all together

- A test suite is said to ***kill*** (or ***detect***, or ***reveal***) a mutant if the mutant fails a test that the original passes.
- ***Mutation testing*** (or ***mutation analysis***) of a test suite proceeds by making a number of mutants and measuring the fraction of them killed by that test suite. This fraction is called the ***mutation adequacy score*** (or ***mutation score***).
    - A test suite with a **higher score is better**.
- (Sorry for all of the vocabulary!)

Mutation testing: pros and cons

Mutation testing: pros and cons

- Has the potential to subsume other test suite adequacy criteria (it **can be very good**)

Mutation testing: pros and cons

- Has the potential to subsume other test suite adequacy criteria (it **can be very good**)
- **Difficult** to do well:
  - Which mutation operators do you use?
  - Where do you apply them? How often do you apply them?
    - Typically done at random, but how?

Mutation testing: pros and cons

- Has the potential to subsume other test suite adequacy criteria (it **can be very good**)
- **Difficult** to do well:
    - Which mutation operators do you use?
    - Where do you apply them? How often do you apply them?
        - Typically done at random, but how?
- It is **very expensive**. If you make 1,000 mutants, you must now run your test suite 1,000 times!
    - We started by saying testing (1x) was expensive!

Mutation testing: equivalent mutant problem

- Suppose you have “`x = a + b; y = c + d;`” and you swap those two statements.

Mutation testing: equivalent mutant problem

- Suppose you have "`x = a + b; y = c + d;`" and you swap those two statements.
- The resulting program is a mutant, but it is **semantically equivalent** to the original.

Mutation testing: equivalent mutant problem

- Suppose you have “`x = a + b; y = c + d;`” and you swap those two statements.
- The resulting program is a mutant, but it is **semantically equivalent** to the original.
  - So it will pass and fail all of the tests that the original passes and fails.

Mutation testing: equivalent mutant problem

- Suppose you have "`x = a + b; y = c + d;`" and you swap those two statements.
- The resulting program is a mutant, but it is **semantically equivalent** to the original.
    - So it will pass and fail all of the tests that the original passes and fails.
    - So it will dilute the mutation score

Mutation testing: equivalent mutant problem

- Suppose you have "`x = a + b; y = c + d;`" and you swap those two statements.
- The resulting program is a mutant, but it is **semantically equivalent** to the original.
  - So it will pass and fail all of the tests that the original passes and fails.
  - So it will dilute the mutation score
- Detecting these "***equivalent mutants***" is a big deal. How hard is it?

Mutation testing: equivalent mutant problem

- Suppose you have "`x = a + b; y = c + d;`" and you swap those two statements.
- The resulting program i **ally** **equivalent** to the origi
  - So it will pass and fa ginal passes and fails.
  - So it will dilute the mutation score
- Detecting these "***equivalent mutants***" is a big deal. How hard is it?

Remember when I mentioned **reductions** earlier? Now is a good time to do one!

Mutation testing: equivalent mutant problem

- Detecting these “***equivalent mutants***” is a big deal. How hard is it?

Mutation testing: equivalent mutant problem

- Detecting these "***equivalent mutants***" is a big deal. How hard is it?
- It is **undecidable**! (= there is no algorithm for it that can always give the correct answer)

Mutation testing: equivalent mutant problem

- Detecting these "***equivalent mutants***" is a big deal. How hard is it?
- It is **undecidable**! (= there is no algorithm for it that can always give the correct answer)
    - by direct reduction to the **Halting Problem** (or by **Rice's theorem**)

```
def foo():           # foo halts if and only if
if p1() == p2():     # p1 is equivalent to p2
    return 0
foo()
```

Takeaways

- Individual tests should be hermetic and focused
  - avoid flaky and brittle tests
- Three lenses for test suite quality: logic, statistics, and adversity
- Lens of **Logic**: “no visit X → no find bug in X”
  - leads to statement and branch coverage.
- Lens of **Statistics**: “sample the inputs the users will make”
  - leads to beta testing, A/B testing.
- Lens of **Adversity**: “poke realistic holes in the program and see if you find them”
  - leads to mutation testing.