Advanced Software Testing

Chapter 1

What is Testing?

- Method in which we compare actual and intended behaviors to show that the intended functionality is correctly implemented and to detect failures.
- Verification method

What it is not:

- Improvement of quality which is achieved by debugging
- Fault localization
- Static analysis of code, be it manual or automatic
- Testing can never show absence of errors only their presence

What is a test case?

Input and expected output along with environmental conditions

What is a good test case?

Ability to detect likely failures with good cost-effectiveness.

- Cost of writing/executing/evaluating the test
- Cost of failure in the field—severity
- Test suites: few and short tests
- Plus: easy track-down to potential faults

Why is testing difficult?

Because designing good test cases is a challenge. Also, we may not know the expected output. We don't know when to stop.

1. There may be millions of lines of code to test, many degrees of freedom. E.g. autonomous vehicles
2. We may not have a specification. E.g.: Google Search, neural networks
3. The systems to test may be very complicated e.g. autonomous vehicles, continuous controllers, cyber-physical systems

Test oracle problem

The challenge of distinguishing the corresponding desired, correct behaviour from potentially incorrect behaviour

Testing Selection can be based on

- equivalence classes
- requirements and knowledge of typical or hypothesized defects
- control flow and decisions
- data flow

- stochastic profiles (pick test cases at random or w.r.t. distribution)

Testing Selection Strategy

A test selection strategy is good if it is better than random testing.

Good strategies:

Limit Testing: Test on boundaries. Encodes defect hypothesis, an increased likelihood of defect

Stopping Criteria

- When time is up
- When no more defects are detected with existing test cases
- When a coverage criterion is met (Useful if there is a relationship with failure detection)
- When a specific number of failures was detected (Useful if we have good estimates)

Verification and Validation:

Verification is building the system right.
Validation is building the right system.

Verification can be viewed as a part of validation

Test-Driven Development:

Method in which test cases serve as specifications and drive development

Defects:

Failure:

- Deviation of actual I/O behaviour w.r.t. intended behaviour.
- Occurs at runtime.
- User-centered "interface" perspective

Error:

- Deviation of system's actual state w.r.t. its intended state. (When system is in invalid state)
- May or may not lead to failure.
- Internal "transition system" perspective

Fault:

- actual or hypothesized reason for the deviation
- can only be corrected at design time: fix & rebuild
- Developer-centered perspective

Fault –(may cause)-> error-(may cause)->failure

Different terminology: error is a mistake in the process, fault is a mistake in the code

Wrap-Up

- Testing is about detecting failures. No guarantees.
- Fundamental problem is that of test selection.
  ➢ But writing the test infrastructure isn't easy either – in particular, for embedded systems
  ➢ Cost! Many ways to organize the testing process in different development process models.
- Testing "functionality" is just one task.
- Take all numbers with a grain of salt!
- Types of defects found vary with the method used

Chapter 2

Software Development Activities:

1. Requirements Engineering
2. Elicitation, Analysis, Specification, Validation
3. Design
4. Implementation
5. Verification and Validation
6. Operation and Maintenance

Testing Activities:

- Planning
- Organization
- Documentation
- Test Case Derivation
- Execution
- Monitoring
- Evaluation

Testing in the development process:

- Unit Tests
- Integration Tests
- System Tests
- Acceptance tests

V-Model:

Business Case — Release Testing

Requirements — Acceptance Testing

System Specification — System Testing

System Design — Interface Testing

Component Design — Component Testing

Construct Component

Terminology

- Black-Box Testing: Unaware of the system. Test functionality only.
- White-Box Testing: Uses structure of system under test. Example- Code Coverage Testing
- Robustness Testing: System should not crash
- Performance Testing: Response Time
- Regression Testing: To test if changes to the system did not break the system i.e. the system is working as it was working earlier.
- Incremental Testing: Test modules one by one. Top-down or Bottom-up.
- Big-Bang Testing: Test all modules in isolation; then combine all of them
- Drivers: simulate callers
- Stubs: simulate callees
- Integration Testing: tests communication between 2 modules. Top-Down Integration Testing (requires only stubs, no drivers). Bottom-up Integration Testing (requires drivers, no stubs)
- Subsystem Testing: tests the overall functionality of the subsystem
- Continuous Integration: Macro process of frequently integrating code into a shared code repository and automatically building and testing the code base
  - Reduced risk of accumulating integration problems
  - Integration verified by an automated build, test and often even deployment (continuous
  - delivery/deployment)
  - Possibly, more suitable infrastructure or environment on CI server to execute tests (e.g. through virtualization)
- Requirement-Based Testing: Black-Box testing. Equivalence Partitioning.
- Equivalence Partitioning: Create a classification tree (Boxes are classifications, elements are classes).
- Control Flow-Based Testing:
  - C0: statement coverage: Coverage of all nodes in the CFG. Ratio= #executed statements / #statements
  - C1: branch coverage: make sure every edge of the CFG is executed at least once
  - C2: condition coverage: each literal of a condition to evaluate once to true and once to false
  - C3: condition / decision coverage: Condition coverage plus result must evaluate to both true and false
  - C4: modified condition / decision coverage (MC/DC): Independently swapping the value of each literal leads to a test case with different result. Recommended for validating requirements-based tests (i.e. not for test selection!)
  - C5: multiple condition coverage: All combinations of literals are tried. 2^n test cases.
  - C6: path coverage: Define test cases (test data) such that every path from the entry to the exit point in the CFG is executed once. Undecidable for loops without upper bound. Exhaustive path testing is, like exhaustive input testing, not realistic. Exhaustive path testing doesn't guarantee revelation of all failures.

Problems: Missing paths/statements/branches, dependence on data values not considered, specification may be wrong

Semantically “equivalent” programs require different test suites for the same coverage
For dependent parameters 100% coverage not possible.

For microservices prefer integration tests over unit tests

If you use statement coverage as your testing criteria, you are not better than random testing.

Control flow graphs are useless because data is not considered in the testing which possibly will leave out important test cases. Use these for assessment of test cases but not for selection of test cases.

Requirement based testing do not subsume fault-based testing and hence no better than random testing.

“Statement coverage” – like all selection criteria – is of dual use [Zhu et al. 1997]

➢A-posteriori: measure an existing test suite

➢A-priori: use as selection criterion

➢(use as stopping criterion)

Coverage criteria based on control flow used as

➢Selection criteria (before testing)

➢Quality indicators (after testing)

➢Stopping criteria (to end testing)

A good test case can detect likely failures with good cost effectiveness. Detecting it depends on the system.

Boundary Interior Testing:

Test cases such that:

1. You reach the loop but don’t iterate the loop.
2. You reach the loop and iterate a path of the loop but only once.
3. Iterate the loop more than once.

No conclusive evidence that coverage correlates with fault detection.

…Subsumptions..

Requirements based testing -> equivalence classes

Coverage based testing -> equivalance classes based on paths/statements etc.

Coverage criteria based on control flow used as

➢Selection criteria (before testing)

➢Quality indicators (after testing)

➢Stopping criteria (to end testing

Structured—provide methodological guidelines

➢Potential for automation (problem: f-1)

Measurable

➢May lead to optimization of the wrong parameter

Purely syntactic—no relationship with requirements

➢Always use in combination with black-box testing.

Don't use as test selection criterion (will get back to this)!

Combinatorial Testing:

Faults are based on interactions of few parameters

Chapter 3

Random Testing

- Black-Box testing technique
- Test input values are selected randomly and independently from the specified range of values (test domain). Can be selected uniformly or from a specific probability distribution (statistical testing).
- Tests are executed with these random input values.
- The results are compared against the expected output: → Possibly "exception/no exception" (robustness testing)
- General random testing not a good idea for AI systems

Statistical Testing

- Instead of uniformly picking test inputs from the complete test domain, we test the most frequent interactions with a system first.
- We use so-called usage or operational profiles as a basis for picking test inputs.

Operational Profiles

use cases plus occurrence rates. Take use cases from common interactions etc. but the most problematic operations would have the least probability of occurring. We cannot assess severity of failure from operational profiles.

GUI random testing

Problem: Some functionality might only be accessed after a sequence of events

Random walk on specification/environment models

Adaptive random testing

♣ Generate a set of candidate test cases by randomly selecting from the input domain.

♣ Choose the test case from the set of candidates that is farthest away from the already executed ones.

♣ Execute this test case and repeat if no failure was found.

Another way:

- Instead of randomly choosing input values, we randomly generate output values and find corresponding input values.
- The idea is to maximize the differences between the selected output values.
- [Alshahwan and Harman] show that output variety can make coverage information more useful for fault detection.

Fuzzing:

Black-box random testing

Checks for software vulnerabilities through dynamic testing i.e., by semi-randomly generating abnormal test cases.

Target program is monitored for security-related bugs

Software tools that perform fuzzing are called fuzzers:

Black-box fuzzers: Simply check if a fuzz test crashed the target program or not

White-box fuzzers: Use heavyweight program and runtime analyses to drive the fuzzing process

Grey-box fuzzers: Employ only lightweight analysis techniques such as code coverage

Test case generation:

Random

Model-based:

 Predefined model

 Test cases are generated from some format description, e.g. RFC, documentation, etc.

 Anomalies are added to each possible spot in the test inputs

 Knowledge of input format should give better results than random fuzzing

 Can deal with complex dependencies, e.g. checksums

¬ Inferred model: Input format of a black-box system is gradually derived during execution ("Specification Mining")

Mutation-based:

Little or no knowledge of the structure of the inputs is assumed

 Seeds / seed inputs = Set of (initial) inputs accepted by the target program

 Mutation operators are used to add anomalies to

 An initial corpus of valid seed inputs

 Existing semi-valid test inputs generated in previous iterations

 Mutations may be completely random or follow some heuristics (cf. following examples)

 Bit-flipping

 Arithmetic-, block-based-, and dictionary-based mutations

 Crossover (cf. „Genetic Algorithms“)

Execution Monitoring

Use sanitizers (error detectors) to detect vulnerabilities that do not lead to program crashes (but to incorrect

calculations)

 Address Sanitizer: Detect memory-related vulnerabilities (e.g. used-after-free vuls., buffer-overflows)

 UB Sanitizers: Detect undefined behavior (UB) in C/C++ programs

 Thread Sanitizers: Help to reveal concurrency-related bugs (e.g. data races)

Fuzzing - Bug Triage

 Analysis and reporting of fuzz tests that cause security violations

 Deduplication

 Only consider fuzz tests that lead to "unique" security violations

 Techniques: Stack backtrace hashing, coverage-based and semantics-aware deduplication

 Prioritization & Exploitability (a.k.a. "fuzzer taming problem")

 Ranking resp. grouping of violating fuzz tests according to their uniqueness and severity

 Techniques: Heuristics, machine learning, ...

 Test case minimization

 Identify the portion of a violating test case that is necessary to trigger the violation and create a test case that is

smaller and simpler than the original (but still causes the same violation).

White Box fuzzing: Symbolic execution

Grey Box fuzzing:

lightweight program instrumentation to trace runtime information for each input fed to a fuzz target. Based upon this information, grey-box fuzzers add „interesting“ test inputs (for further mutations) to the seed corpus and also prioritize them („Evolutionary Fuzzing“)

Random Testing:

Advantages:

 Saves time and effort when compared with other testing strategies.

 Generated tests are unbiased.

 Knowledge about the software's implementation is not necessary

Disadvantages:

 Many test cases are redundant or unrealistic.

 Does not solve the oracle problem → only robustness and reliability testing are possible

 Often only capable of finding basic bugs (because "deep" parts of the code cannot be reached, also for fuzzing) -

but state-of-the art for security testing

 Predictability/Stability? One would love to get similar results when applying random testing twice in a row.

Fault localization

Chapter 4

"detecting at least one failure" may or may not be an adequate measure for effectiveness

Weyuker and Jeng:

Similar (abstract) results for modification (moving elements) and intersection of blocks in [Weyuker&Jeng].

Partition testing can be better, worse, same as random testing, depending on the choice of blocks.

 W.r.t. the ability to detect at least one failure.

 How well is this suited to compare random and partition testing?

 Not worse if block sizes and # tests per block identical.

 Yet, even then no clear superiority of partition-based testing.

In particular, when the failure rates of the blocks are equal, then partition testing is as good as random testing.

Partition testing will be good when partitions are defect-based and hence have a high defect density.

Directed fuzzing targets specific typical memory faults, therefore can expect to lead to "good" test cases!

Coverage is not a good selection criterion, but from a practical perspective maybe a good assessment criterion – possibly necessary but certainly not sufficient!

These results do not tell us how good particular testing strategies are.

Same if failure cases equally distributes.

Better if all and only failure cases in one partition. for this one block θi = 1, hence Pp = 1

Worse if (8/99, 0/1)

Blocks of Equal Size Favor Partition Testing i.e. Same # of tests and elements in each block

Defect hypothesis tries to concentrate the failure causing inputs into one block

Gutjahr:

For independent failure rates with equal expected value θъ(that's not the same as identical failure rates) and

one test drawn from each of the k blocks:

 Partition testing is better or the same as random testing: P⊔$p \geq P$⊔

Weyuker&Jeng model: Random testing can be better, worse, and same as partition testing.

Criterion "detect at least one failure"

Identified special cases where partition testing was not worse (but not by much).

 In particular, equal failure rates and equal distribution among blocks.

Gutjahr: Deterministic assumptions on failure rates favor random testing.

 Effect particularly strong if a partition consists of few large and many small blocks.

 Independent failure rates with identical expected value in the blocks guarantee a higher or equal Pp .

 Seeing failure rates as random variables favors partition testing with a factor up to k.

 Fundamental assumptions: equal expected failure rates and one test per block

Gutjahr's results also hold for different measures for "better":

 At least one failure detected

 # of faults detected

 Weighted # of faults detected

Adaptive Partition Testing:

If we disover a fault in a block of a partition, we should take a deeper look at this block to possibly discover other faults.

 If no fault is detected, take a look at other blocks.

Main idea of adaptive partition testing:

 1. Split input domain into blocks ci

(with i = 1, 2, … , m), which have a probability pi

to get selected.

 2. Next, a block ci

is randomly selected and then a test case from this block is randomly selected.

 3. The probabilities pi are adjusted depending on the outcome of the test case.

 If the test case fails, we want to take a deeper look at this block and increase its probability pi

.

 Otherwise, we assume this block to include "fewer" faults and decrease its probability pi

.

 4. Repeat steps beginning from 2.

 Possible termination conditions:

• Testing resource has been exhausted.

• A certain number of test cases has been executed.

• A certain number of faults has been detected.

Comparison with the methods "Dynamic Random Testing" (DRT) and "Random Partition Testing" (RPT)

 RPT divides the input domain into blocks that have a specific probability to be chosen. A block is randomly selected based on

these probabilities. Then, a concrete test case is randomly selected from the chosen block.

 DRT is similar to RPT, but the probabilities of the blocks are not static, but instead changed dynamically. If a test case from a

block reveals a fault, the probability of this block is increased by a constant. Otherwise, it is decreased by another constant.

Chapter 5

Testing Concurrent Programs:

Deadlock: mutual exclusion, no preemtion, circular wait, hold and wait

Non-deadlock bugs: Starvation, livelock, suspension, race conditions

Hard to test because occur rarely, hard to reproduce and analyze, lead to violation of functional and quality requirements

Soln:

Test all possible interleavings, random testing, defect hypothesis

Testing CP systems:

No requirements/oracle available. System works in dynamic open context. X-in-the-Loop test setups rely on a simulation environment

Testing autonomous driving:

Test against a safe operating envelope. Test for violations of the safe operating envelope.

In a "good" test scenario, a

➢ correct system approaches

➢ faulty system violates

the limits of the safe operating envelope.

Testing ML models:

No specification

Unforeseen input

Test the machine learning model iteratively and retrain with increased training set

Metamorphic testing: reverse engineers a part of the specification

Watchdogs during runtime

Robustness testing: Check if a machine-learned model outputs the correct classification despite adversarial perturbations of normal resp. correctly classified inputs. Symbolic execution(pixel attack)

Test Ending Criteria

Fault injection:

Mutation testing there can be Equivalent Mutants

➢ Competent Programmer Hypothesis: "Programmers' faults are of a syntactical nature only"

➢ Coupling hypothesis: "Syntactic faults correlate with other kinds of faults"

Regression Testing:

Retesting of a system or component to verify that changes, such as source code modifications, have not caused unintended effects and that the system or component still complies with its specified requirements. A regression is a new version of the code with failing tests that formerly passed.

Co-evolving with code base of SUT

Test Case Prioritization for early fault detection

Chapter 6

Model based testing

Model: Abstraction of SUT/environment. There should be minimum loss of info. Ensures separation of concerns. Cost of validating model < cost of validating SUT. Drivers bridge levels of abstractions.

Model-based testing in the hardware industry

 Need for redundancy is acknowledged

 Reluctance in the SW industry!

reliability engineering

Models primarily built for test case generation

Search based technique for CPS:

Create model, create error state, create search space(all possible inputs), create fitness function, run search

Structural tests complement functional tests

Chapter 7

Fault models in state charts:

State of an object is modified by operations; methods can be modeled as state transitions

Test cases are sequences of method calls that traverse the state machine model

All transitions, combinations of transitions can be tested for:

Missing or incorrect transition or event

Missing or incorrect action

Extra, missing or corrupt state

Illegal message failure

Trap door

Inheritance:

Incorrect initialization, conflict between superclass and subclass, naked access, naughty children, wormholes, spaghetti inheritance

Tested by flattening

Chapter 8

Delta debugging for test case minimization

Fault localization: use syntactic blocks as granularity level

Use dynamic call graphs

> **A-Prioiri: code metrics and code churn**
> **Complexity metrics correlate with defects**

Failure clustering:

Ex1:

Ans1

Quality Assurance Activity.

Examining the behaviour of a system in order to discover potential faults

Used to verify a system dynamically against its specification with a finite set of test cases.

Purpose: check requirements and find failures.

Issues:

Testing can never show absence of errors only their presence.

Test oracle problem – the challenge of distinguishing the corresponding desired, correct behaviour from potentially incorrect behaviour (**determining the correct output for a given input**). An oracle for a software program might be a second program that uses a different algorithm to evaluate the same mathematical expression as the product under test. This is an example of a pseudo-oracle, which is a derived test oracle. During Google search, we do not have a complete oracle to verify whether the number of returned results is correct. We may define a metamorphic relation[17] such that a follow-up narrowed-down search will produce fewer results. This is an example of a partial oracle, which is a hybrid between specified test oracle and derived test oracle.

A statistical oracle uses probabilistic characteristics,[18] for example with image analysis where a range of certainty and uncertainty is defined for the test oracle to pronounce a match or otherwise. This would be an example of a quantitative approach in human test oracle.

A heuristic oracle provides representative or approximate results over a class of test inputs.[19] This would be an example of a qualitative approach in human test oracle.

Test selection and stopping criteria

To find faults in the system that could possibly lead to failure. Developer can't test outside her perspective. Difficult to formulate test cases and costly to test each possible case.

Improving quality and fault localization

Test case: test input, expected output, environment conditions. Manual, automatic/semi-automatic.

Ans 2:

Walkthroughs, reviews and inspections

Ans 3:

After implementation

Ans 4:

Test case: input, output and environment conditions. Test case execution: manually, automatic, semi-automatic.

Ans 2

Failure: Deviation of actual I/O behavior w.r.t. intended behavior

Error: Deviation of system's actua l state w.r.t. its intended state (invalid state)

Fault: actual or hypothesized reason for the deviation

Fault –(may cause)-> error-(may cause)->failure

after the component has been developed, after it has reached the client

Ans 3

Good test case: ability to detect failure with good cost-effectiveness

Quality of a test case depends on the program/system and the distribution of its faults.

Why inject faults into a program?

-To assess the quality of a test suite i.e. if it is able to detect that.

-Eg: Mutation testing, make a small change in the program and see if it affects the test suite.

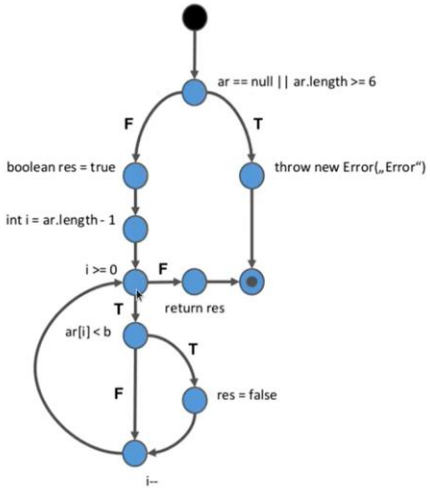
To evaluate a given test selection criteria we should compare it to random testing. Or we can use mutation testing.

During testing: as stopping criteria

After testing: Find the coverage of the tests

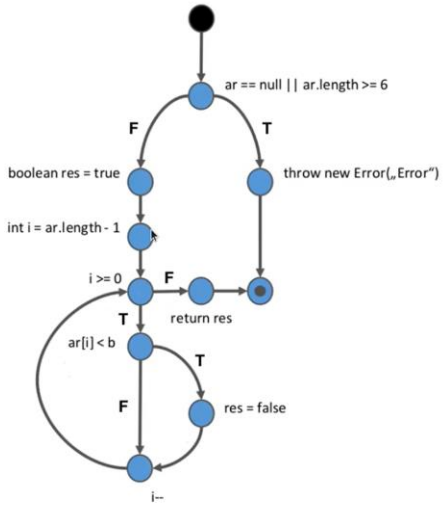
Don't use the same test selection criteria for during and after testing.

Exercise sheet 1 – Exercise 4: Control-Flow Based Testing




```
public static boolean check (int[] ar, int b) {
   if ( ar == null || ar.length >= 6 )
      throw new Error("Error");
   else {
      boolean res = true;
      int i = ar.length - 1;
      while ( i >= 0 ) {
         if ( ar[i] < b )
            res = false;
         i--;
      }
      return res;
   }
}
```


- Number of paths:
  - $1 + 1 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 = 64$



Exercise sheet 1 – Exercise 4: Control-Flow Based Testing

Nr.	A	B	C	D	B || C	A && (B || C)	B && D	(A && (B || C)) || (B && D)
1	0	0	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0
3	0	0	1	0	1	0	0	0
4	0	0	1	1	1	0	0	0
5	0	1	0	0	1	0	0	0
6	0	1	0	1	1	0	1	1
7	0	1	1	0	1	0	0	0
8	0	1	1	1	1	0	1	1
9	1	0	0	0	0	0	0	0
10	1	0	0	1	0	0	0	0
11	1	0	1	0	1	1	0	1
12	1	0	1	1	1	1	0	1
13	1	1	0	0	1	1	0	1
14	1	1	0	1	1	1	1	1
15	1	1	1	0	1	1	0	1
16	1	1	1	1	1	1	1	1

Ans 4

(A&&(B||C))||(B&&D)

Condition Coverage: ABCD = 0110 -> 0, 1001->0

Multiple Condition Coverage: All possible combinations = 16

Condition/Decision: ABCD = 0100 ->0,1011->1

Modified Condition/Decision C:

Coverage(A): ABCD=0011->0, 1011->1

Coverage(B):0011->0, 0111->1

Coverage(C):1000->0, 1010->1

Coverage(D):0100->0, 0101->1

C/D > branch

Exercise sheet 1 – Exercise 4: Control-Flow Based Testing

- A coverage criterion C_1 subsumes a coverage criterion C_2 iff all test suites that satisfy C_1 also satisfy C_2
  - We use the relational operator $\succ$ (preorder), i.e. $C_1 \succ C_2$, to state coverage criterion C_1 subsumes C_2
- Subsumption relationships
  - Path coverage $\succ$ boundary-interior coverage $\succ$ branch coverage $\succ$ statement coverage
  - Multiple condition coverage $\succ$ MC/DC $\succ$ C/D coverage $\succ$ condition coverage
- Check the truth tables to see if this is correct ;-)

Boundary Interior Testing:

Test cases such that:

4. You reach the loop but don't iterate the loop.
5. You reach the loop and iterate a path of the loop but only once.
6. Iterate the loop more than once.

```
Public void func(int v){

Int index=0;

Int result=0;

While(index<v){

Result=index/(v-3);

Index++;

}

}
```

Success test suite ={0,1,2}

Failing test suite ={0,1,3}

Combinatorial Testing:

d1,n1,m1

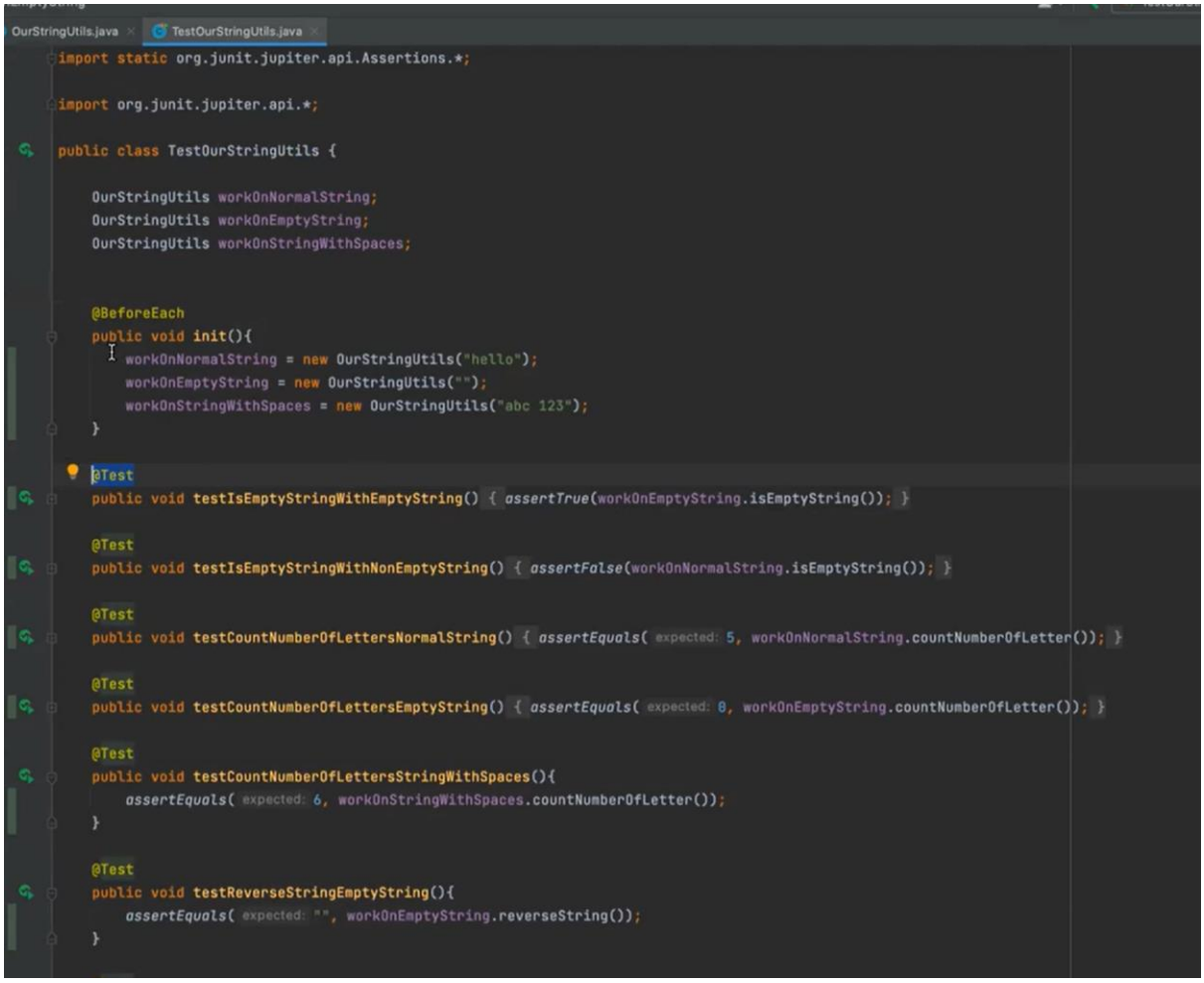
d1,n2,m2

d1,n1,m3

d2,n1,m2

d2,n2,m1

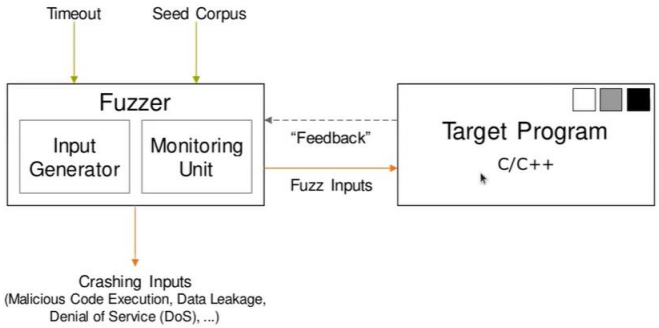
d2,n2,m3



Black box fuzzing – we are unable to test deeply

White box fuzzing/Symbolic execution – testing in each branch of a control flow graph

Fuzzing- security testing activity in which a target program is executed with semi-valid inputs in order to find security-related bugs.



Random input generation:

Adv:

Easy and fast to generate

Dis:

Often rejected by program

Doesn't reach deeper code regions

Executes same path again and again

Model based input generation:

Test inputs are generated from a formal description, eg: documentation, RFC etc.

Adv:

Can deal with complex dependencies and input checks

Dis:

Generation is costly

Only applicable for certain programs where specifications are formally defined

Mutation based:

Inputs generated by applying mutations to seed inputs(set of inputs accepted by the target program)

Mutations:

Bit flipping, arithmetic, block based, dictionary based

Adv:

Little or no knowledge of input format required

Easy and fast to generate

Dis:

Effectiveness of fuzzing strongly depends on the selected seed inputs

How to detect non-crashing vulnerabilities:

To ensure non-crashing vulnerabilities:

Sanitizers- Error detectors.

- Address Sanitizer: memory related vulnerabilities eg: used after free vulnerabilities, buffer overflows
- Undefined Behaviour:
- Thread Sanitizers: concurrency related bugs eg data races

Black box fuzzing:

Target program simply checked for observable program crashes without the help of sanitizers

Used when no sources are available or no instrumentation is possible. Eg 3rd party software

White box fuzzing:

Use symbolic execution engines to detect new program paths, each path is then fuzzed to find vulnerabilities with a fuzzer for a certain period of time

Requires heavy weight program and runtime analysis

Does not scale for large software systems

Grey box fuzzing:

Lightweight program instrumentation to trace runtime instrumentation for each input fed to a fuzz target.

Add 'interesting' test inputs to the seed corpus for further mutations(Evolutionary fuzzing) using fitness metrics like code coverage, distance to potentially vulnerable locations(guided fuzzing) etc.

Program instrumentation: profiling, logging program crashes, inserting timers in source code

Defect hypothesis 1:

Unexpected inputs i.e. input bombs, maybe problematic(random)

Defect hypothesis 2:

Similar to legal inputs i.e. semi-valid inputs, maybe problematic (mutation based)

Defect hypothesis 3:

Guided fuzzing, are problematic eg: memcpy,strcpy

GUI Random Testing:

Android Debug bridge

Run Monkey program

Adv:

We can test a large amount of the operations in our application in a random manner.

Useful for robustness testing where the expected outcome is no crash/no exception

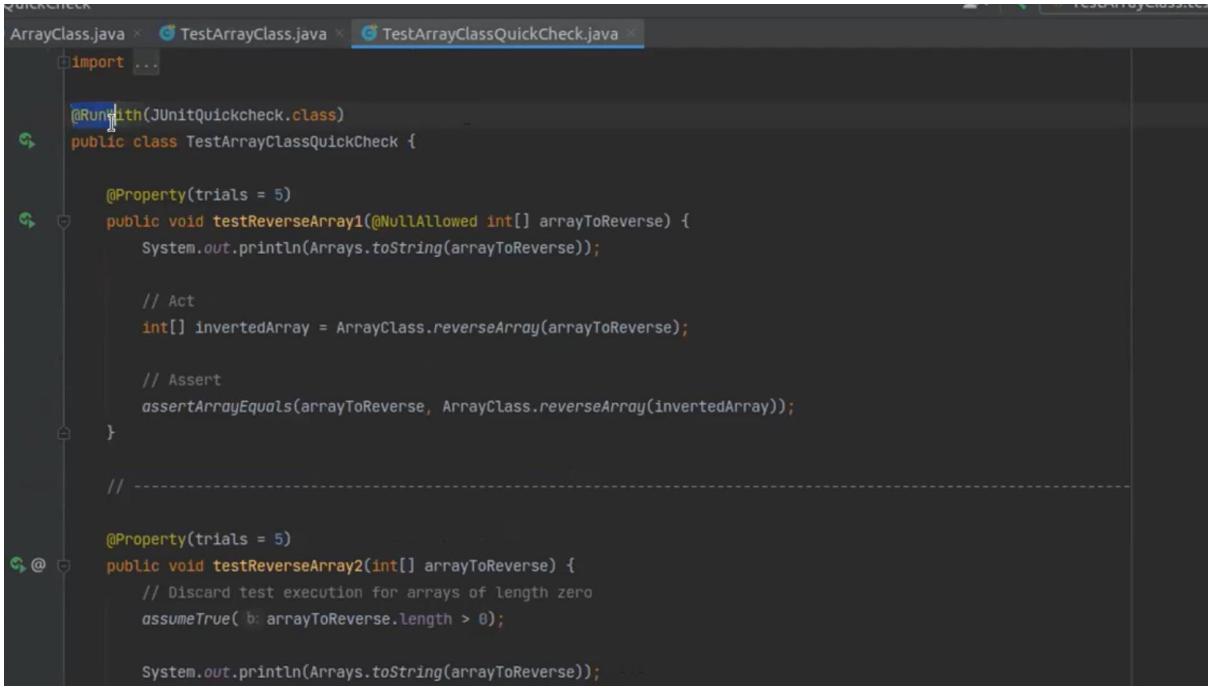
Dis:

We do not know the expected outcome(oracle problem)

Some functionality will only be accessed after a sequence of events

Quick-Check

Property based testing framework



Heart-bleed bug: buffer over-read,[5] a situation where more data can be read than should be allowed.[6]

Statistical Testing:

Operational Profiles:

➢ Essentially use case descriptions with probabilities

➢ Prioritization of "important" interactions → risk-based testing

➢ Reliability assessment of the system under test → stopping criterion

➢ Assessment of the development process (“Cleanroom Softwrae Engineering”)

➢ Operational profiles can also be represented with Markov Chains:

▪ Markov chains can be used for analysis purposes and for computing test cases.

▪ There exists tool support for both intentions, e.g. JUMBL or MaTeLo.

What is problematic about this method?

➢Fundamentally problematic operations often have a low probability to occur, e.g. accident at nuclear power plant.

➢ We can not assess the severity of an operation with operational profiles!

General Random Testing Approaches :

- Random walk on specification

    Advantages and disadvantages of random walks:

    + New perspective by randomly walking on the

    specification / environment model.

    - Takes long to cover the complete specification / model

    (expected # required tests can be computed).

- Adaptive random testing:

    ▪ Generate a set of candidate test cases by randomly selecting from the input domain.

    ▪ Choose the test case from the set of candidates that is farthest away from the already executed ones.

    ▪ Execute this test case and repeat if no failure was found.

    Choice of distance measure is crucial

- Randomly generate output values and find corresponding input values. Try to maximize difference between output values.

3.2. Fuzzing - Bug Triage

- Analysis and reporting of fuzz tests that cause security violations
- Deduplication
  - Only consider fuzz tests that lead to "unique" security violations
  - **Techniques:** Stack backtrace hashing, coverage-based and semantics-aware deduplication
- Priorization & Exploitability (a.k.a. "fuzzer taming problem")
  - Ranking resp. grouping of violating fuzz tests according to their uniqueness and severity
  - **Techniques:** Heuristics, machine learning, ...
- Test case minimization
  - Identify the portion of a violating test case that is necessary to trigger the violation and create a test case that is smaller and simpler than the original (but still causes the same violation). Will consider at the end of the semester !

3.2. Fuzzing - Fuzzing rules of thumb

- Protocol specific knowledge very helpful
  - Generational tends to beat random, better spec's make better fuzzers
- More fuzzers is better
  - Each implementation will vary, different fuzzers find different bugs
- The longer you run, the more bugs you find
- Use different seed inputs
- Best results come from guiding the process
  - Notice where your getting stuck, use profiling!

Analysis Random Testing vs. Partition-Based Testing

Partition based- equivalence partitioning. Equivalence can be based on whether the divided blocks trigger the same failures or if they cover the same part of the program or if they correspond to the same requirement etc.

Operational profile

Weyuker and Jeng:

Partition testing can be better, worse, same as random testing, depending on the choice of blocks.

➢ W.r.t. the ability to detect at least one failure. ▪ How well is this suited to compare random and partition testing?

➢ Not worse if block sizes and # tests per block identical. ▪ Yet, even then no clear superiority of partition-based testing.

➢ Confirmed by experiments and empirical studies In particular, when the failure rates of the blocks are equal, then partition testing is as good as random testing.

Partition testing will be good when partitions are defect-based and hence have a high defect density.

Directed fuzzing targets specific typical memory faults, therefore can expect to lead to "good" test cases!

Pp Good if we have defect hypothesis.

In [1]:
```
import random
```

In [2]:
```
def fuzz_generator(max_length=100, char_start=32, char_range=32):
    string_length = random.randrange(0, max_length + 1)

    out = ""
    for i in range(0, string_length):
        out += chr(random.randrange(char_start, char_start + char_range))

    return out
```

In [7]:
```
import os
import tempfile
```

In [8]:
```
def write(file, text, mode = 'w'):
    with open(file, mode) as fh:
        fh.write(text)
```

In [9]:
```
file = os.path.join(tempfile.mkdtemp(), "fuzz_input.txt")
print(file)
```

```
/tmp/tmpc6shh_03/fuzz_input.txt
```

In [10]:
```
import subprocess as sp
```

In [11]:
```
program = "bc" # or /usr/bin/bc
fuzz_input = fuzz_generator()

write(file, fuzz_input)

result = sp.run([program, file], stdin=sp.DEVNULL, stdout=sp.PIPE,
                stderr=sp.PIPE, universal_newlines=True)
```

In [16]:
```python
from datetime import datetime
```

In [17]:
```python
def current_timestamp():
    return int(datetime.timestamp(datetime.now()))
```

In [18]:
```python
def fuzz_target(timeout, program):
    result_list = []

    file = os.path.join(tempfile.mkdtemp(), "fuzz_input.txt")

    timestamp = current_timestamp()
    while (current_timestamp() - timestamp) < timeout:
        fuzz_input = fuzz_generator()

        write(file, fuzz_input)

        result = sp.run([program, file], stdin=sp.DEVNULL, stdout=sp.PIPE,
                        stderr=sp.PIPE, universal_newlines=True)

        result_list.append((fuzz_input, result))

    return result_list
```

Afl-cmin: tries to minimize the seed corpus. Removes all redundant seed inputs that trigger the same range of instrumentation data points.

Afl-tmin: test case minimizer

Exercise sheet 3 – Exercise 3: Model of Weyuker & Jeng

1. What is the quality criterion for the assessment of random testing and partition-testing in this model' sensible criterion?
   - Finding at least one failure causing input.
   - Usually, one would like to find all faults by finding respective failure-causing inputs!
2. Which formulas are used for random testing and partition-based testing?
   - Formula for random testing:

$$P_r = 1 - (1 - \theta)^n$$

   - Formula for partition-based testing:

$$P_p = 1 - \prod_{i=1}^{k} (1 - \theta_i)^{n_i}$$

   - How do we get to these formulas? Let's take a look at the one for random testing:
     - Idea: Compute the probability P to find no failure-causing input and then invert it with $1 - P$
     - Probability to find no failure-causing input in 1 test case: $1 - \theta$
     - Probability to find no failure-causing input in n test cases: $(1 - \theta)^n$
     - Probability to find at least one failure-causing input in n test cases: $1 - (1 - \theta)^n = P_r$

3. What is the result of the analysis of random testing vs. partition-based testing in the model?
   - Partition-based testing can be better, worse, or the same as random testing!
   - It depends on the partitioning of the input domain:
     - Example: $d = 99, m = 9, n = 3 \rightarrow P_r = 1 - (1 - \theta)^n = 1 - (1 - \frac{9}{99})^3 = 0.25$

Result	PT is better than RT	PT is worse than RT	PT is the same as RT
When does it happen?	If one of the blocks includes only failure-causing inputs.	If one block includes zero failure-causing inputs.	If the domain is divided equally and we have equal failure rates.
Possible partitioning:	1/42 | 1/50 | 7/7	0/10 | 5/50 | 4/39	3/33 | 3/33 | 3/33
Compute P_p:	$P_p = 1 - \prod_{i=1}^{k}(1 - \theta_i)^{n_i}$ $= 1 - \left(1 - \frac{1}{42}\right)^1 * \left(1 - \frac{1}{50}\right)^1 * \left(1 - \frac{7}{7}\right)^1$ $= 1 - 0.976 * 0.98 * 0 = 1$	$P_p = 1 - \prod_{i=1}^{k}(1 - \theta_i)^{n_i}$ $= 1 - \left(1 - \frac{0}{10}\right)^1 * \left(1 - \frac{5}{50}\right)^1 * \left(1 - \frac{4}{39}\right)^1$ $= 1 - 1 * 0.9 * 0.897 = 0.19$	$P_p = 1 - \prod_{i=1}^{k}(1 - \theta_i)^{n_i}$ $= 1 - \left(1 - \frac{3}{33}\right)^1 * \left(1 - \frac{3}{33}\right)^1 * \left(1 - \frac{3}{33}\right)^1$ $= 1 - 0.91 * 0.91 * 0.91 = 0.25$

4. What is the underlying assumption of the model?
   - We work with fixed failure rates!
   - Tester has to know these failure rates, which he or she can't know in advance.
   - But, they might have an idea about them → Gutjahr's model

5. Would you recommend using coverage-based testing on the grounds of Weyuker & Jeng's model?
   - A coverage criterion induces a partitioning of the input space.
   - Weyuker & Jeng's model shows us that without any underlying fault model, such a partition-based testing may perform better, the same, or worse than random testing.
   - → one can also perform random testing instead

8. One may argue that requirements-based testing is not useful according to Weyuker & Jeng's analysis. W could / should one do it after all?
   - Requirements of a system describe the desired functionality of the system.
   - Test cases that are derived from the objectives and conditions of the requirements are based on a fault model.
   - If we base the partitions on such a fault model, we can perform better than random testing.
   - Also, remember that the purpose of testing is to check the requirements.

Exercise sheet 3 – Exercise 4: Idea of Gutjahr's Model

1. How does Gutjahr change the assumption of Weyuker & Jeng?
   - He proposes the use of random variables instead of fixed failure rates.
   - This means that we work with an expected failure rate and the formulas change accordingly.

2. What is the result of the analysis of random testing vs. partition-based testing in Gutjahr's model?
   - Same quality criterion as Weyuker & Jeng: finding at least one failure causing input
   - Partition-based testing can be better or the same as random testing!

3. What is Gutjahr's underlying assumption?
   - Gutjahr assumes equal expected failure rates.
   - But, this is a critical assumption!

4. Which other quality criteria can be used for measuring different testing methods?
   - Number of faults detected
   - Weighted number of faults detected
   - Number of test cases until the first fault is detected

Exercise sheet 4 – Exercise 1: Statistical Testing

1. What is the idea behind statistical testing?
   - Test the most frequent interactions first instead of uniformly picking inputs from the complete domain.
   - We can use operational profiles as a basis for picking test inputs.
   - Enables us to measure the reliability of a system.

2. What can we assess when we test our system based on operational profiles?
   - How often each module (e.g. class, method) is executed.

3. What can we not assess with this approach?
   - The severity of the modules.
   - E.g. the correct working of a shutdown for a nuclear power plant is very important, but has a low occurrence probability.

4. Is statistical testing a good strategy to test libraries?
   - Libraries are used in different ways by different programs.
   - You may not have an idea of how often functions are used and which functions are critical.
     - Testing of libraries is program-specific!
   - Proposed strategy for a developer of a library: use a different strategy!
   - Proposed strategy for a developer of a program depending on a library: you can use usage profiles!

Exercise sheet 4 – Exercise 2: Adaptive Partition Testing

1. Why do the authors propose a new technique for testing programs?
   - Random testing and partition-based testing both have advantages and disadvantages:
     - Only if the partitions are based on a fault model, partition-based testing performs better than random testing.
     - Random testing might miss faults that can be easily revealed by partition-based testing.
   - Idea: Combine these two approaches and observe the performance.

2. Explain how the two proposed algorithms in general work.
   - Markov-chain based adaptive partition testing (MAPT):
     - Uses a Markov matrix to dynamically adapt the probabilities of the partitions to get selected for testing.
     - The Markov matrix represents for each partition the probability to select each of the partitions for the next test.
     - If a test case reveals a fault, the probability to change to any other partition is decreased and the probability to stay in the current partition is increased.
     - Otherwise, the probability to stay in the same partition is decreased and the other probabilities are increased.
   - Reward-punishment based adaptive partition testing (RAPT):
     - Uses a reward and punishment mechanism to adjust the selection probabilities of the partitions.
     - If a test case reveals a fault, the reward variable of this partition is incremented by 1 and the punishment variable is set to 0.
     - Otherwise, the reward variable of this partition is set to 0 and the punishment variable is incremented by 1.

3. Which fault-detection effectiveness metrics are described?
   - P-measure: displays the probability of detecting at least one fault
   - E-measure: describes the expected number of faults that are detected by a test suite
   - F-measure: talks about the expected number of test cases that are necessary to detect the first fault
     - In the experiments, this measure is used.

4. Summarize the two research questions of the empirical studies and the main findings.
   - RQ1: How effective are the two proposed methods in detecting software faults?
     - Both variants need significantly fewer test cases to detect the first fault than Dynamic Random Testing (DRT) and Random Partition Testing (RPT).
   - RQ2: What is the test selection overhead for the two proposed algorithms?
     - RAPT has a significantly lower overhead than DRT and RPT.
     - The overhead of MAPT's test selection strategy is only marginally lower than the one of DRT and RPT.
   - In the experiments, the authors could not observe a strong correlation between different levels of granularity or different types of initial probability profiles with the performance of the algorithms.

5. Shortly summarize the threats to the validity of the study.
   - Internal validity: The testing method was implemented by the authors themselves.
   - External validity: Only three programs are used in the evaluation. Although the partitioning is done with regard to the requirements, it is still a subjective process.
   - Construct validity: Threats are small since common metrics are used.
   - Conclusion validity: Threats are small since a large number of trials were executed to guarantee statistical reliability.

6. Which areas/ problems do the authors want to research in future work?
   - Study the best partitioning and granularity level for the proposed method.
   - How can we design a "good" probability profile that tries to optimize the effectiveness of testing?
   - Investigate the appropriate setting of the parameters of the proposed method.
   - Compare Adaptive Partition Testing with other testing techniques.

- **Livelock:** A thread repeatedly obtains a requested resource but never completes its execution before losing that resource
- **Starvation:** A thread is ready to execute, but is indefinitely delayed because other threads have the necessary resource(s)
- **Suspension:** A thread is forced to wait too long before it can access a shared resource (i.e. it eventually obtains the resource but too late for proper execution)
- **Order Violation:** A failure condition that exists when the nondeterministic ordering of multiple interleaved operations results in a different, unexpected, and potentially incorrect behavior
  - Race conditions (a.k.a. data races) are a special form of order violations in which shared resources are not protected from simultaneous access
- **Atomicity Violation:** The execution of one thread interrupts the execution of another thread, which must run to completion without disruption

Why concurrent systems are hard to test?

- Intermittent/nondeterministic failures that only occur rarely, often only after prolonged execution
- Concurrency-related failures are hard to both reproduce and analyze to identify their root causes
  - Tests may produce different results due to the non-deterministic nature of concurrent systems
- Concurrency bugs often cause violations of functional and quality requirements (e.g., performance, reliability, and safety)
  - E.g. violations in real-time systems can lead to violations of response time, jitter and latency requirements
- Correcting concurrency bugs often requires a redesign of the software rather than simple software code changes

How to detect concurrency bugs?

- **Long-duration / Random Testing**
  - Rely on the fact that concurrency errors occur very rarely
- **Model-based Testing**
  - Use MBT tools that support modeling of concurrent behavior
- **Manual Delays**
  - Use delays (e.g. Thread.sleep()) to enforce specific schedules

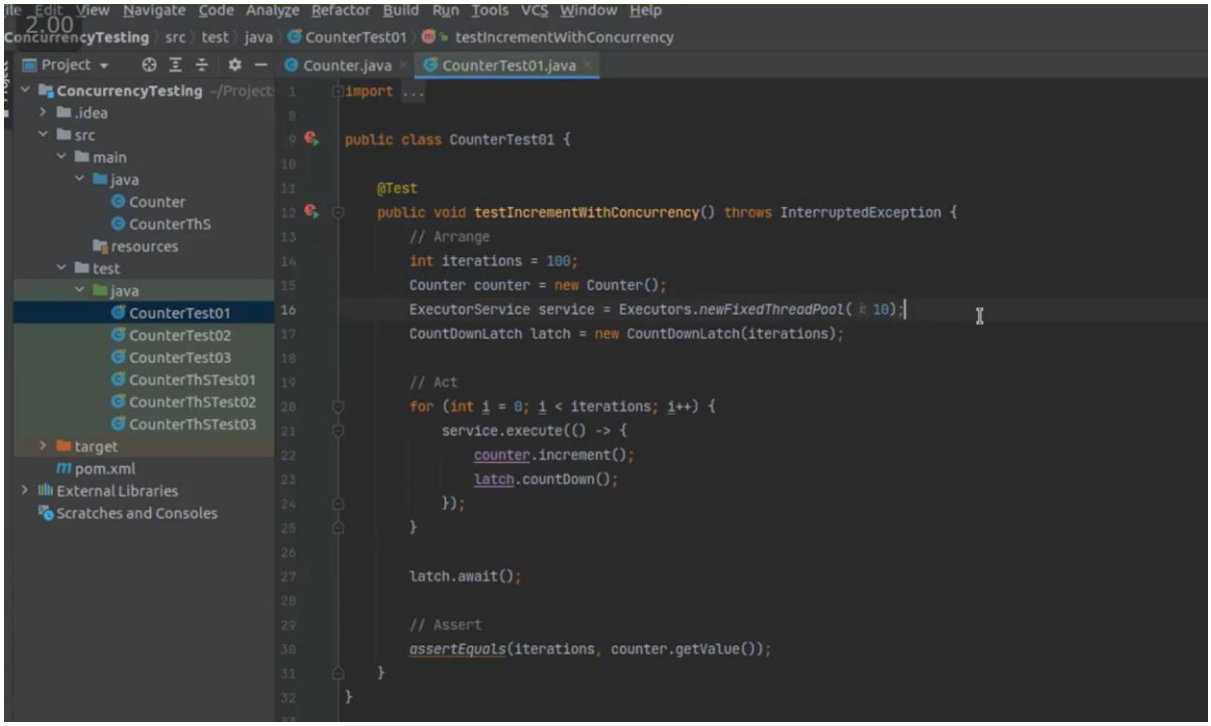
1. Why is the class Counter not thread-safe? Provide an example of an interleaving that shows this. What kind of concurrency bug is this?

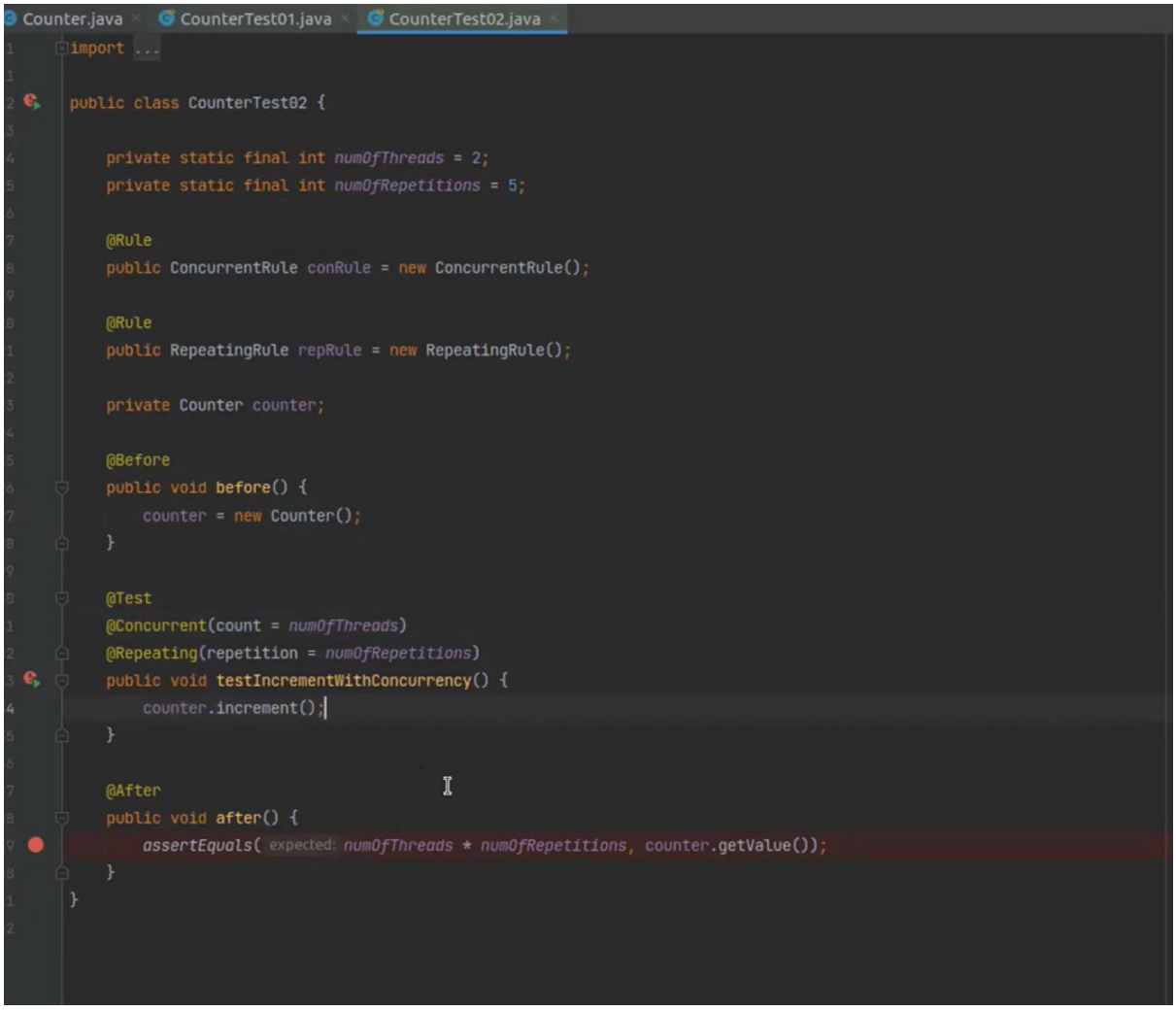
- A *data race* could occur, leading to an incorrect (final) computation.

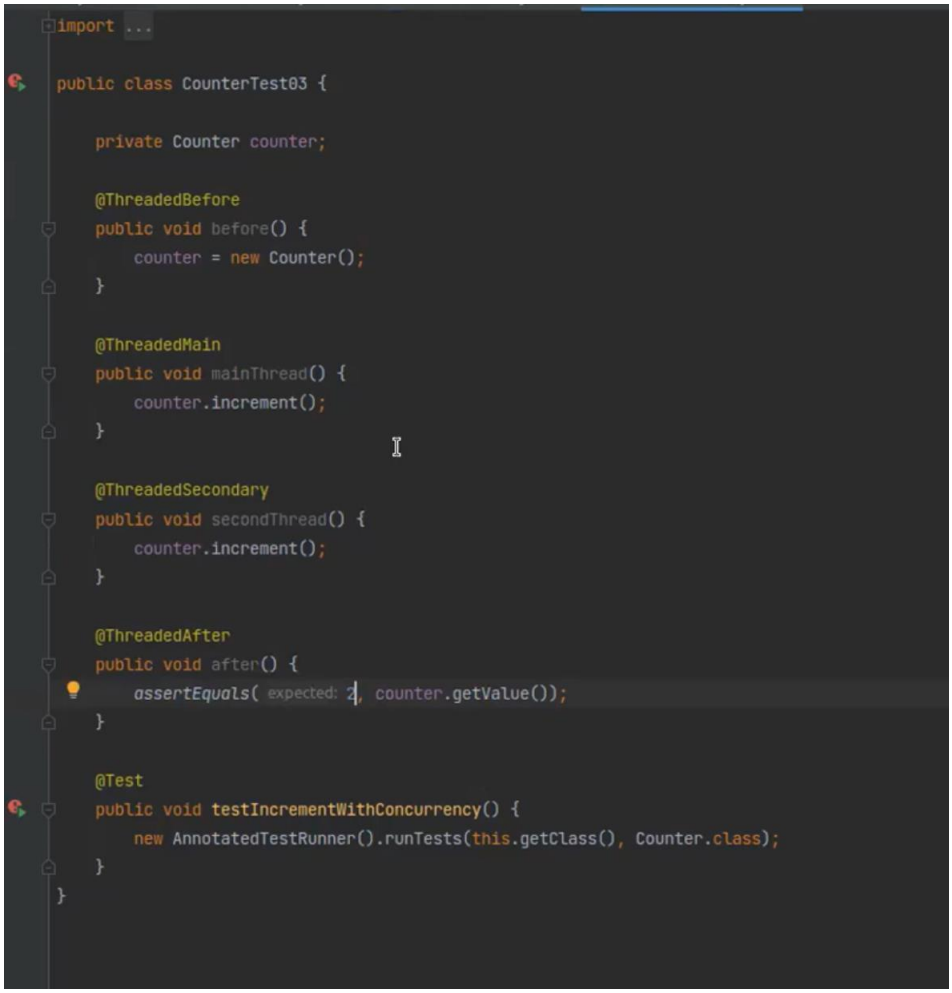
```
public class Counter {
   private int value;
   public int getValue() {
      return value;
   }
   public void increment() {
      int temp = value;
      value = temp + 1;
   }
}
```

1. Justify your answer by giving an example of an erroneous interleaving $l = s_1, \ldots, s_n$ with steps $s \in T \times Op \times Z$ that shows that the value $v \in Z$ of the counter variable is 1 (and not 2 as expected) after it was incremented (atomic operations $read, update, write \in Op$) simultaneously by the threads $thd_1, thd_2 \in T$.

- **Increment operation:**
  - $(thd, read, n)$, $(thd, update, n)$, $(thd, write, n+1)$
- **Valid interleaving:**
  - $(thd_1, read, 0)$, $(thd_1, update, 0)$, $(thd_1, write, 1)$, $(thd_2, read, 1)$, $(thd_2, update, 1)$, $(thd_2, write, 2)$
- **Invalid interleaving:**
  - $(thd_1, read, 0)$, $(thd_1, update, 0)$, $(thd_2, read, 0)$, $(thd_1, write, 1)$, $(thd_2, update, 0)$, $(thd_2, write, 1)$


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
2.00
ConcurrencyTesting › src › test › java › CounterTest01 › testIncrementWithConcurrency
Project
ConcurrencyTesting ~/Project
.idea
src
main
java
Counter
CounterThS
resources
test
java
CounterTest01
CounterTest02
CounterTest03
CounterThSTest01
CounterThSTest02
CounterThSTest03
target
pom.xml
External Libraries
Scratches and Consoles
Counter.java
CounterTest01.java
import ...
public class CounterTest01 {
@Test
public void testIncrementWithConcurrency() throws InterruptedException {
// Arrange
int iterations = 100;
Counter counter = new Counter();
ExecutorService service = Executors.newFixedThreadPool(n: 10);
CountDownLatch latch = new CountDownLatch(iterations);
// Act
for (int i = 0; i < iterations; i++) {
service.execute(() -> {
counter.increment();
latch.countDown();
});
}
latch.await();
// Assert
assertEquals(iterations, counter.getValue());
}
}


Counter.java
CounterTest01.java
CounterTest02.java
import ...
public class CounterTest02 {
private static final int numOfThreads = 2;
private static final int numOfRepetitions = 5;
@Rule
public ConcurrentRule conRule = new ConcurrentRule();
@Rule
public RepeatingRule repRule = new RepeatingRule();
private Counter counter;
@Before
public void before() {
counter = new Counter();
}
@Test
@Concurrent(count = numOfThreads)
@Repeating(repetition = numOfRepetitions)
public void testIncrementWithConcurrency() {
counter.increment();
}
@After
public void after() {
assertEquals(expected: numOfThreads * numOfRepetitions, counter.getValue());
}
}

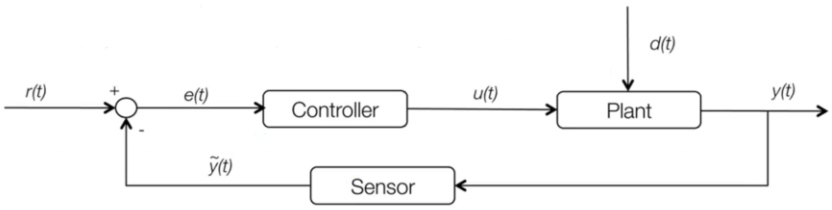

import ...
public class CounterTest03 {
private Counter counter;
@ThreadedBefore
public void before() {
counter = new Counter();
}
@ThreadedMain
public void mainThread() {
counter.increment();
}
@ThreadedSecondary
public void secondThread() {
counter.increment();
}
@ThreadedAfter
public void after() {
assertEquals(expected: 2, counter.getValue());
}
@Test
public void testIncrementWithConcurrency() {
new AnnotatedTestRunner().runTests(this.getClass(), Counter.class);
}
}

```java
import java.util.concurrent.Semaphore;

public class CounterThS {

    private int value;

    //private Semaphore mutex = new Semaphore(1);

    public int getValue() { return value; }

    public synchronized void increment() {
        int temp = value;
        value = temp + 1;
    }

//    public void increment() {
//        try {
//            mutex.acquire();
//            int temp = value;
//            value = temp + 1;
//        } catch (InterruptedException e) {
//            e.printStackTrace();
//        } finally {
//            mutex.release();
//        }
//    }
}
```

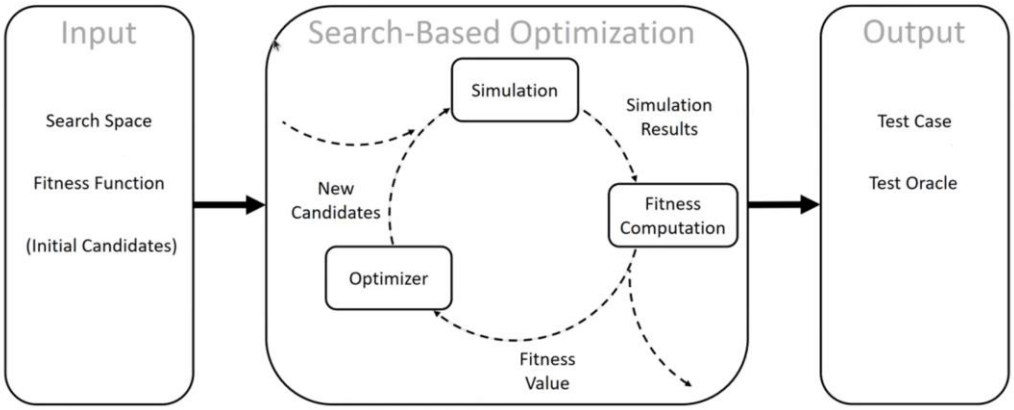
Ex 5:

1. Describe the difficulties when it comes to testing CPS.
   - Usually no useful requirements available
     - E.g. autonomous driving: "Should be able to safely handle any traffic situation" … not helpful …
   - Usually no clear oracle available
     - "Autonomous driving system needs to keep sufficient safety distance"
     - For some traffic situations such safety distances can be defined, e.g. for one car following another on a single lane
     - How to define safety without being to restrictive?
   - Usually system operates in a dynamic open context
     - How to make sure that the system behaves correctly in situations we do not even know up to now?

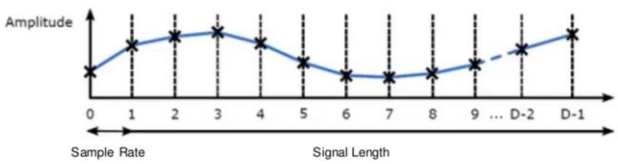
2. Describe in your own words the concept of a controller.



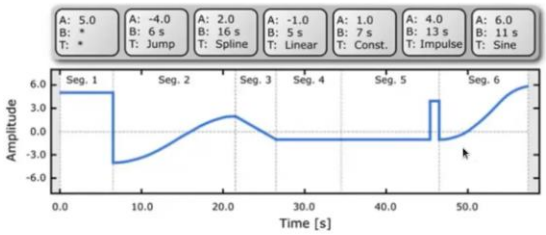
3. Describe in your own words the idea of search-based test generation.



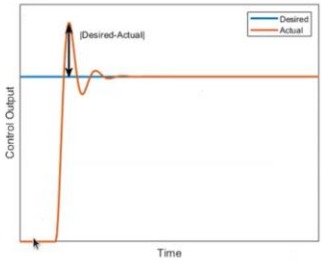
1. What other quality criteria besides stability can you think of that we could use as fault models for test case generation for testing controller?
   - Precision, responsiveness, smoothness, steadiness, …
2. How can those fault models be used for search-based testing?
   - Search for test inputs that either exceed a threshold defined by a domain expert or are very close to that threshold.
3. How would you choose the search space?
   - Search space: "all possible input signals" (simpler: "all possible jumps from one desired value to another")



3. How would you choose the search space?
   - Search space: "all possible input signals" (simpler: "all possible jumps from one desired value to another")
   - Implementation:
     - "all possible input signals": define simple basic signals (constant, polynomial, exponential, sinosoidal,..); each having a certain amount of parameters to vary them; the final input signal is a concatenation of multiple basic signals;
     - "all possible jumps ..": one parameter for start and one for the end;



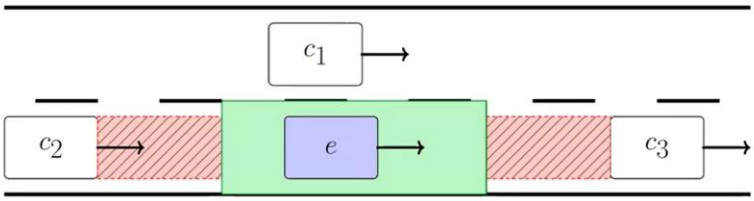
4. The following plot shows the concept of smoothness. What does it mean and how can we formulate a fitness function for it?



   - Smoothness: There should be no (or a not too big) overshoot when transitioning to the new desired value
   - Fitness function: $maximize\ f\ =\ \max(\ |Actual(t) - Desired(t)|)$
   - Have a look at:
     - Holling, Dominik; Stanescu, Alvin; Beckers, Kristian; Pretschner, Alexander; Gemmar, Matthias: Failure Models for Testing Continuous Controllers. 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), 2016
     - Holling, Dominik; Pretschner, Alexander; Gemmar, Matthias: 8Cage: Lightweight Fault-Based Test Generation for Simulink. Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, Association for Computing Machinery (ACM), 2014

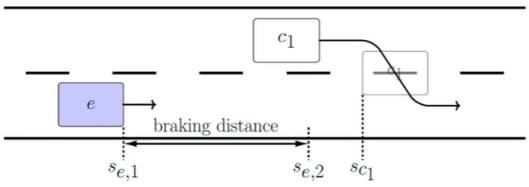
1. What may be interesting test goals? Can you give an abstract example of a search space and a fitness function for them?
   - They have to provide their functionality in a safe and comfortable way:
     - Search space: curvey road
     - Fitness function: "search for the strongest lateral force" (this will annoy the passengers a lot if too large)
   - While not disrupting the traffic flow:
     - Search space: vehicle in front brakes in various ways
     - Fitness function: "search for the braking of the driving system that causes perturbations of the traffic flow" (called "string unstable" driving: causes traffic jams)

2. What does “good” mean for test cases here?



- *A “good” test case can reveal* potential *faulty system behavior.*
- In a “good” test scenario, a
  - correct system approaches
  - faulty system violates
- the limits of the safe operating envelope.

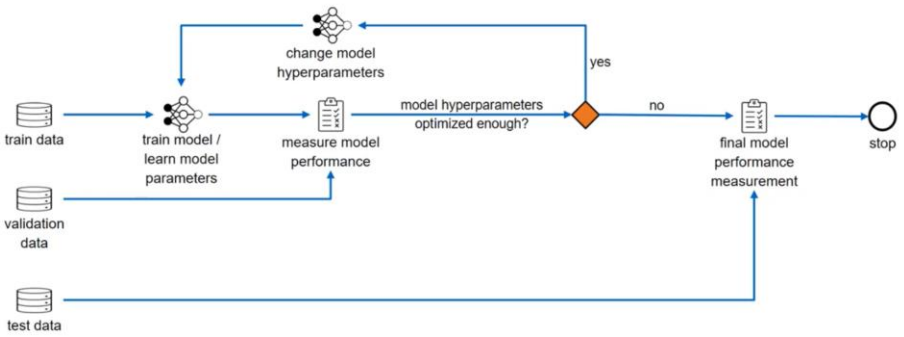
d) How may the fitness function look like if we cannot compute the breaking distance?



- Make use of the templates that will be presented in the lecture:
  - Cut-in needs to happen in front of the ego vehicle.
  - Minimum distance throughout the test case shall be as small as possible (will lead to collision).
  - When collision happens, assign bad fitness value.
- Will lead to a lot of test cases which are barely a collision (show them to an expert and let him/her decide if ok).

Ex 6:

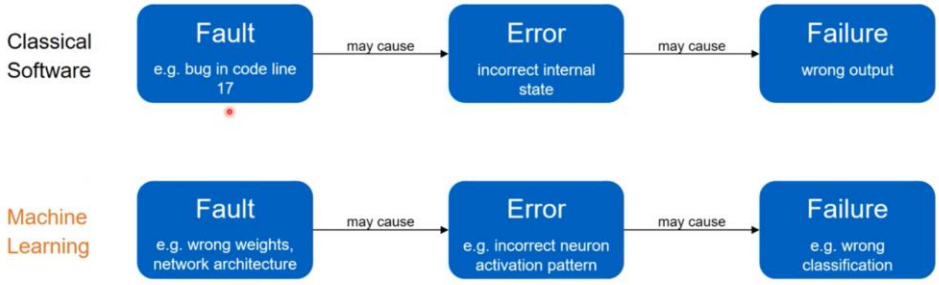
- **Motivation:** Why testing neural networks (NNs)?
  - Benchmark data (BD): After training, the NN yields a perfect prediction score on the test data (50% of the BD)
  - Real-world data: Misclassification of some inputs that are "new" to the NN



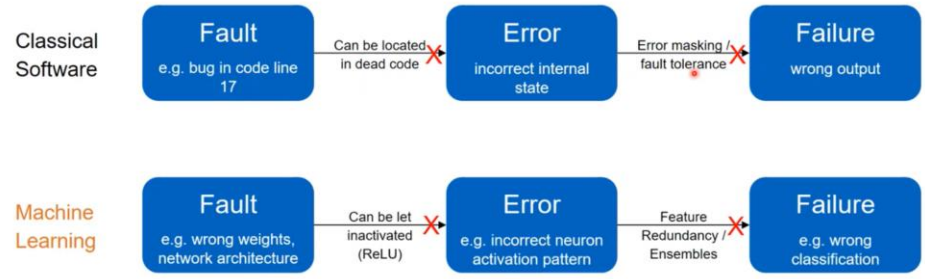
Exercise sheet 6 – Exercise 1: Testing ML-Based Models

- **Motivation:** Why testing neural networks (NNs)?
  - Benchmark data (BD): After training, the NN yields a perfect prediction score on the test data (50% of the BD)
  - Real-world data: Misclassification of some inputs that are "new" to the NN
- **Problem of AI-testing:** No functional specification of the NN
  - How to check if a NN input resp. test case passes or fails without a proper test orcale?
  - How to find test cases that identify missing "functionality" of the NN?
  - Missing specification, test oracle

- The distinction can be generalized to ML models:
  - Example for ML Fault: Wrong weight(s), Wrong connection between Neurons, Wrong Network Architecture
  - Example for ML Error: Incorrect Neuron Activation / Neuron Activation Pattern, (also: neuron should be activated but isn't)
  - Example for ML Failure: Wrong Classification, Wrong Detection, Wrong Bounding Box, Wrong Output



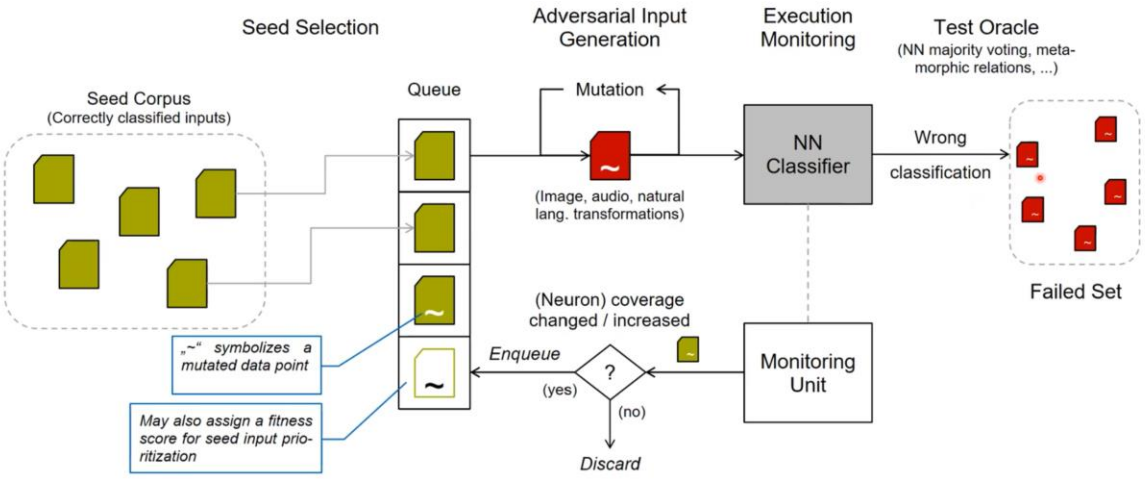
- The interrelationship is transferrable to ML models:
  - **Example for ML Fault → Error:** A Fault does not necessarily lead to an Error as the Fault (e.g. wrong weight) does not always manifest: The Neuron, in which the weights contributes, can be let inactivated due to the ReLU activation function. In other words: it doesn't matter whether the neuron output is -1 or -10 before the activation function
  - **Example for ML Error → Failure:** An Error (wrong neuron activation) does not necessarily lead to a failure (wrong classification) as there exists feature redundancy: Many neurons detect similar things so if one is let inactivated the others might correctly get activated and lead to a correct classification. Also using ensembles instead of a single NN.



- **Step 1:** Translate a neural network NN (image classifier) into an imperative program
- **Step 2:** Detect most important pixels of a labelled image
  - Compute a linear expression (e.g. $c_1x_1 + \cdots + c_nx_n$) in terms of the input variables x (pixels) for each output label
  - Use the coefficients c of the input pixels (summed weights) to assign an importance score for that pixel
    - **Idea:** The higher the importance score, the higher the impact of the pixel on the final classification
  - Sort the pixels according to their importance
- **Step 3:** Create an adversarial image from the original image (1- and/or 2-pixel attacks)
  - Select one or two pixels with a high importance score and make them symbolic (the remaining pixels keep their concrete values)
  - Create a constraint-solution problem (path condition) for the symbolic values so that a solution (concrete values)
    - executes the same program path up to the output layer (ensures that the adv. image resembles the original image) and
    - changes the output label from the originally predicted label

Exercise sheet 6 – Exercise 2: Testing Approaches for ML Models

- **Step 1:** Partition the input-output space of the NN using neuron coverage
  - **Assumption:** Neuron coverage correlates with NN behavior
    (i.e. changes in neuron coverage changes the NN predictions)
  - Tian et al. showed correlation for their use case; does this generalize???
- **Step 2:** Select a seed image (driving situation) and apply synthetic image transformations to simulate rain, fog, camera pollution and -rotation, ...
  - **Goal:** Increase neuron coverage resp. find a representative (with transformations) that triggers new NN behavior and thus also increases test coverage



Exercise sheet 6 – Exercise 2: Testing Approaches for ML Models

- Majority voting by multiple NNs with the same functionality
- Domain-specific metamorphic relations
  - For example, a relationship that defines that a car should behave similarly under different lighting conditions (cf. driving situation above)

Exercise sheet 6 – Exercise 3: Mutation Testing

1. Describe in your own words how mutation testing works.
   - Modify the program by introducing a small syntactic change.
   - This modified program is called a mutant.
   - Then, run the existing test suite and see if the mutant(s) were detected.
   - In this case the mutant is killed.
   - If a test suite can not kill all mutants, this may hint at deficiencies of the test suite and we should add further tests.

2. What are the assumptions for mutation testing? Are they justified?
   - Competent Programmer Hypothesis
   - Coupling Hypothesis
   - They may be justified:
     - There is some empirical evidence.
     - However, programmers sometimes make logical mistakes.

3. How is the mutation score calculated? Why is it problematic to use this score in practice?
   - $mutation\ score = \frac{\#\ mutants\ killed}{\#\ non{-}equivalent\ mutants}$
   - What is problematic?
     - It is undecidable if a mutant is equivalent to the original program.

```java
private int number;

public NumberUtils(int number) { this.number = number; }

public boolean isPalindromic(){
    if(this.number >= 0) {
        int numberToCheck = this.number;
        int reversedNumber = 0;
        int currentLastDigit;
        while (numberToCheck > 0) {
            currentLastDigit = numberToCheck % 10;
            reversedNumber = reversedNumber * 10 + currentLastDigit;
            numberToCheck /= 10;
        }
        return reversedNumber == this.number;
    } else {
        return false;
    }
}
```

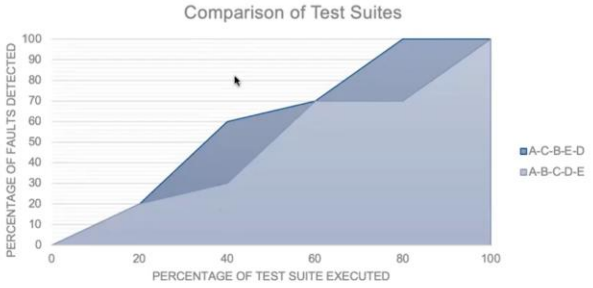
```java
        assertFalse(numUtils.isPalindromic());
    }

    @Test
    public void testIsPalindromicLowNumberPalindromic(){
        numUtils = new NumberUtils(1);
        assertTrue(numUtils.isPalindromic());
    }

    @Test
    public void testIsPalindromicLowNumberNotPalindromic(){
        numUtils = new NumberUtils(10);
        assertFalse(numUtils.isPalindromic());
    }

    @Test
    public void testIsPalindromicHighNumberPalindromic(){
        numUtils = new NumberUtils(313313);
        assertTrue(numUtils.isPalindromic());
    }

    @Test
    public void testIsPalindromicHighNumberNotPalindromic(){
        numUtils = new NumberUtils(212112);
        assertFalse(numUtils.isPalindromic());
```

Exercise sheet 7 – Exercise 1: Regression Test Optimization – Test Case Prioritization

1. Name three examples for properties optimized by TCP.
   - Value / Cost-based objectives:
     - Value-based: (Early) fault-detection capability, coverage
     - Cost-based: Execution Time (could differ due to more / less efficient ordering), Setup Costs

2. For T (A-B-C-D-E) and the prioritized test suite T' with order A-C-B-E-D, draw a diagram with the test suite fraction in percent on the x-axis and the percent of detected faults on the y-axis.

Test / Fault	1	2	3	4	5	6	7	8	9	10
A	x		x							
B	x		x	x						
C	x	x	x				x	x	x	
D			x	x						
E					x	x				x



3. Compare the two prioritized test suites T' (A-C-B-E-D) and T'' (C-A-B-E-D) regarding their APFD.
   - $APFD = 1 - \frac{TF_1 + \ldots + TF_m}{n * m} + \frac{1}{2 * n}$

Test/Fault	1	2	3	4	5	6	7	8	9	10
A	x		x							
B	x		x	x						
C	x	x	x				x	x	x	
D			x	x						
E					x	x				x

   - We have a set of $m = 10$ faults and our test suite contains $n = 5$ test cases.
   - $T' = A - C - B - E - D = 1 - \frac{1+2+1+3+4+4+2+2+2+4}{10 * 5} + \frac{1}{2*5} = 0.6$
   - $T'' = C - A - B - E - D = 1 - \frac{1+1+1+3+4+4+1+1+1+4}{10 * 5} + \frac{1}{2*5} = 0.68$

Exercise sheet 7 – Exercise 2: Regression Test Optimization – Regression Test Selection

1. Evaluate how TS' performs w.r.t. fault-detection capabilities compared to the retest-all strategy by calculating its average recall.
   - TS': select test cases that either failed in the previous 2 runs or have not been executed in the previous 2 runs.

CI run/Test case	A	B	C	D	E
1	*P*	*P*	*F*	*F*	*P*
2	*P*	*P*	*F*	*P*	*F*
3	*P*	*P*	*P*	*F*	*P*
4	*P*	*P*	*P*	*F*	*P*
5	*F*	*P*	*F*	*P*	*P*

   - Selection with TS': [{A, B, C, D, E}, {C, D}, {C, D, E}, {A, B, C, D, E}, {D}]
   - Recall for TS': $recall = [\frac{2}{2+0}, \frac{1}{1+1}, \frac{1}{1+0}, \frac{1}{1+0}, \frac{0}{0+2}]$
   - Average recall for TS': $average\ recall = \frac{1+0.5+1+1+0}{5} = 0.7$

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives}$$

2. Evaluate how TS' performs w.r.t. fault-detection capabilities compared to the retest-all strategy by calculating its average precision.
   - TS': select test cases that either failed in the previous 2 runs or have not been executed in the previous 2 runs.

CI run/Test case	A	B	C	D	E
1	P	P	F	F	P
2	P	P	F	P	F
3	P	P	P	F	P
4	P	P	P	F	P
5	F	P	F	P	P

   - Selection with TS': [{A, B, C, D, E}, {C, D}, {C, D, E}, {A, B, C, D, E}, {D}]
   - Precision for TS': $precision = [\frac{2}{2+3}, \frac{1}{1+1}, \frac{1}{1+2}, \frac{1}{1+4}, \frac{0}{0+1}]$
   - Average precision for TS': $average\ precision = \frac{0.4+0.5+0.33+0.2+0}{5} = 0.29$

$$Precision = \frac{true\ positives}{true\ positives + false\ positives}$$

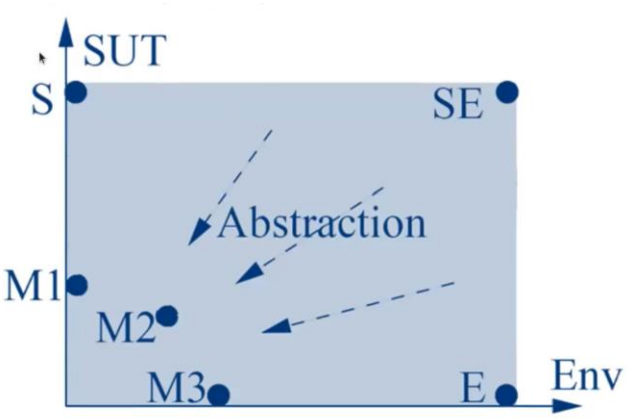
1. What are the general advantages and disadvantages of abstraction?
   - Advantages of abstraction:
     - Abstraction gives a high-level overview, it reduces the complexity.
     - It makes systems comprehensible for non-technical people.
     - Abstraction allows focusing on certain aspects (e.g. safety).
   - Disadvantages of abstraction:
     - Bridging the gap between abstraction and implementation may be hard.
     - Abstraction can omit important aspects.
     - It enhances the risk of forgetting about details.

2. Explain the difference between abstraction by encapsulation and abstraction by omission.
   - Abstraction by encapsulation:
     - We hide the details, but keep them accessible.
     - Frequently used in architecture & design.
   - Abstraction by omission:
     - Details are not known.
     - Enables us to focus on a certain aspect.
     - Frequently used in requirements engineering.

3. What are the different roles of models of the system under test and models of the environment for testi

- Model of the system under test:
  - This model provides an oracle (i.e. the expected outcome for a specific input).
  - Its structure can be used for automated test case generation.
- Model of the environment:
  - This model restricts the possible inputs to the system under test.
  - It acts as a test selection criterion.
  - The model can also describe typical interaction patterns.

3. What are the different roles of models of the system under test and models of the environment for testi



S: models all details of the SUT
- no knowledge about expected environment
- no sanity constraints on the input
- high complexity

E: complete model of the ENV
- specifies all legal test inputs
- no information about the SUT and thus the expected output
- we can use it for robustness testing

SE: we model SUT and ENV completely
- too much detail to be practical
- high effort when creating the model

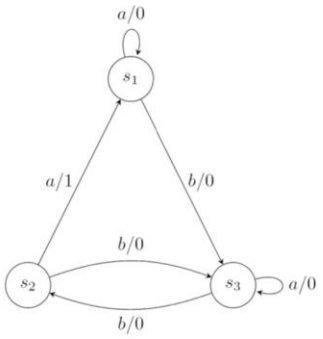
→ needs to be reduced to M1, M2, M3

4. Think about the differences of test selection criteria at the level of code and at the level of models.
   - Test case selection criteria for code:
     - Specification-based (functional) criteria
     - Code-based (structural) criteria
   - Test case selection criteria for models:
     - Model coverage criteria
     - Requirements-based criteria (if we have requirements models)

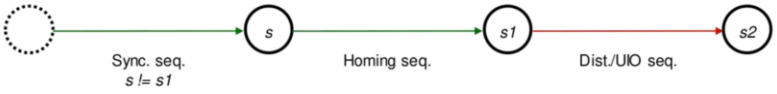
5. Is it easier to test programs that are written in Java or those which are written in Java byte code?
   - Program written in Java:
     - These programs are readable by the tester.
     - Programmer's intention might become clear.
     - White-box testing is possible.
   - Program written in Java Bytecode:
     - These programs are (normally) not readable by the tester.
     - Programmer's intention is probably unclear.
     - Only black-box testing is possible.
     - Tester has to rely on the specification for test case selection.

Exercise sheet 7 – Exercise 4: Black-Box Testing with Mealy Machines

- Test if a black-box (BB) system conforms to its specification, i.e., verify that "each" transition of the spe ification is implemented correctly, using *Mealy machines* (output BB system == output sepc. ???)
- *Mealy machine* = Finite-state machine (FSM) whose output values are determined by both its current st and the current inputs



- **Sequences for Mealy machines:**
  1. **Synchronizing sequence:** Sequence of input values that brings the Mealy machine from any of its states to a guaran state s.
  2. **Homing sequence:** Sequence of input values that leads to different end-states in the Mealy machine, which can be determined by the output sequences.
  3. **Distinguishing sequence:** Sequence of input values that produces different output sequences when we apply it to each state of the Mealy machine.
  4. **Unique-Input-Output (UIO) sequence:** Sequence of input values that enables us to determine from the output sequences from which state we started.



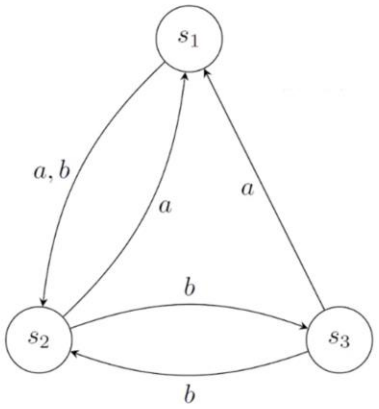
1. Find a synchronizing sequence for Mealy machine 1.

- Synchronizing sequence: Sequence of input values that brings the Mealy machine from any of its states to a guaranteed state s.

Start state	Input sequence	End state
s_1	ba	s_1
s_2	ba	s_1
s_3	ba	s_1

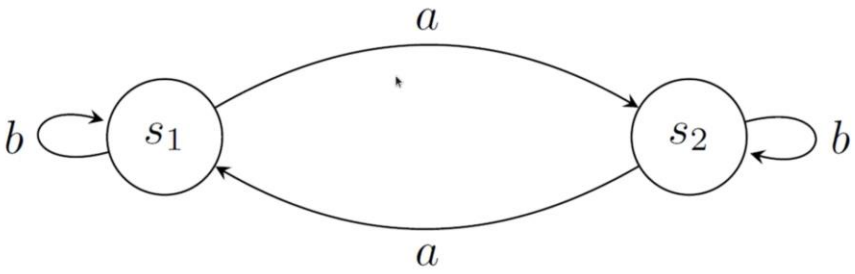
- Sequence "ba" is a synchronizing sequence.
- It brings us always to state s_1.

Mealy machine 1



2. Provide a Mealy machine for which no synchronizing sequence exists.

- Synchronizing sequence: Sequence of input values that brings the Mealy machine from any of its states to a guaranteed s



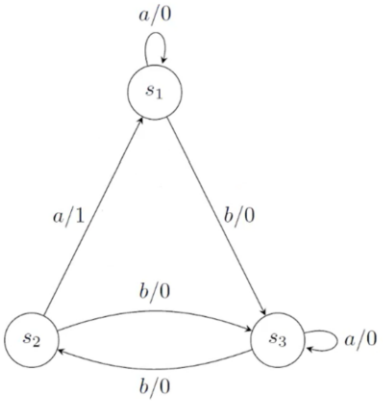
3. Detect a homing sequence for Mealy machine 2.

- Homing sequence: Sequence of input values that leads to different end-states in the Mealy machine, which can be determined by the output sequences.

Start state	Input sequence	Output sequence	End state
s_1	ba	00	s_3
s_2	ba	00	s_3
s_3	ba	01	s_1

- Sequence "ba" is a homing sequence.
- It leads to different end states for different output sequences, but to the same end state for equal output sequences.

Mealy machine 2



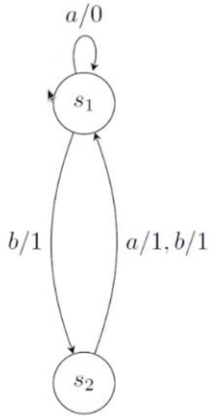
4. What is a distinguishing sequence for Mealy machine 3?

- Distinguishing sequence: Sequence of input values that produces different output sequences when we apply it to each state of the Mealy machine.

Start state	Input sequence	Output sequence
s_1	a	0
s_2	a	1

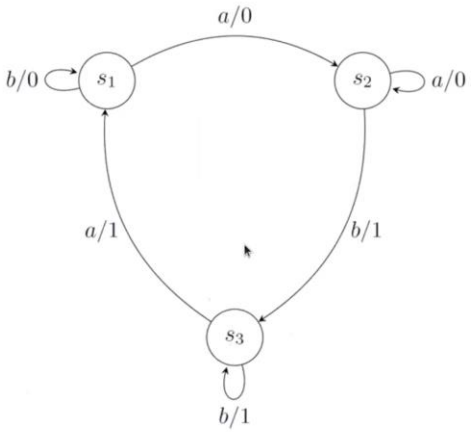
- Sequence "a" is a distinguishing sequence.
- It produces different output sequences for different start states.

Mealy machine 3



5. Provide a Mealy machine for which no distinguishing sequence exists.

- Distinguishing sequence: Sequence of input values that produces different output sequences when we apply it to each sta the Mealy machine.



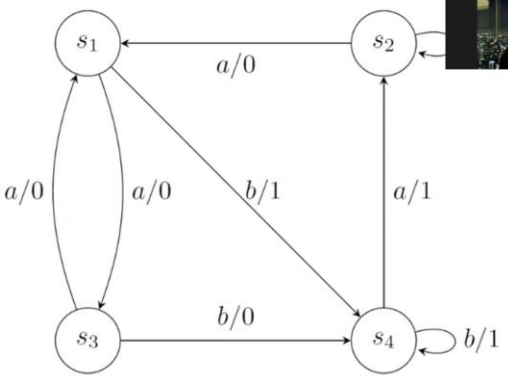
6. Find UIO sequences for each of the states of Mealy machine 4.

- Unique-Input-Output (UIO) sequence: Sequence of input values that enables us to determine from the output sequences from which state we started.

Input sequence	Output sequence for s_1	Output sequence for s_2	Output sequence for s_3	Output sequence for s_4
a	0	0	0	1
b	1	1	0	1
aa	00	00	00	10
ab	00	01	01	11
ba	11	10	01	11

- Sequence "a", "b", "ab", and "ba" are UIO sequences for the 4 states.
- We can determine the start state from the output sequences.
- Every dist. sequence is also a UIO sequence!

Mealy machine 4



7. Under which circumstances can these sequences be used for practical model-based testing of implementations?
   - If a system can be modeled as a mealy machine!

Abstraction:

1. **Encapsulation: hiding info**
2. **Loss of information: removing info**

Driver: bridges the gap between the abstracted model and the concrete SUT.

Model based testing: used for hardware testing, reliability testing

Models for test case generation: for autonomous driving

2 distinct models: too expensive, redundant

Selection criteria:

Functional: we test functionality by encoding how a system can go wrong

Ad-hoc: We ask experts to come up with good test cases

Structural: Based on coverage

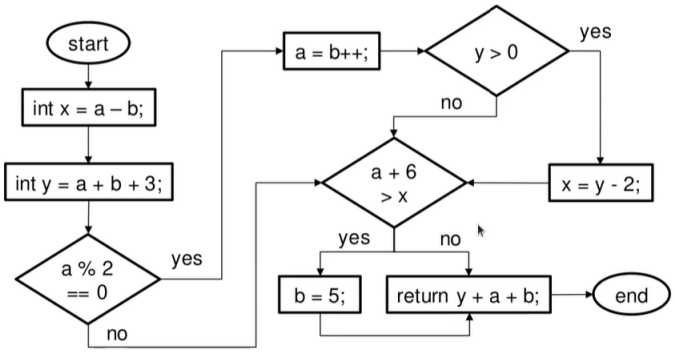
Stochastic:

Fault-based:

Structural criteria is problematic because test cases based on coverage are not better than random testing.

Symbolic Execution:

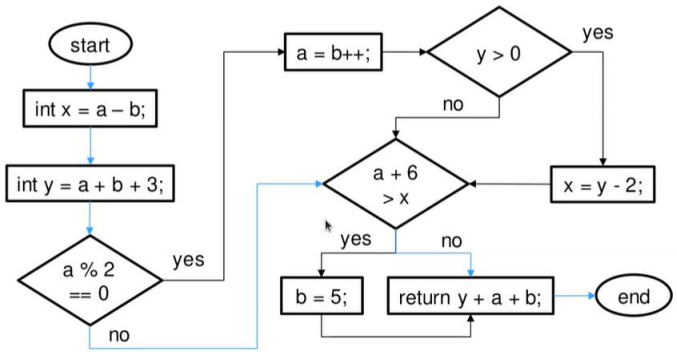
1. Draw the control flow graph for the function *fun*.




```
int fun(int a, int b){
    int x = a - b;          (1)
    int y = a + b + 3;      (2)
    if(a % 2 == 0){         (3)
        a = b++;            (4)
        if(y > 0)           (5)
            x = y - 2;      (6)
    }                       (7)
    if(a + 6 > x)           (8)
        b = 5;              (9)
    return y + a + b;       (10)
}
```

Exercise sheet 8 – Exercise 1: Understand Symbolic Execution

2. How many paths are possible for this function?



6 possible paths

3. Use symbolic execution to determine the path condition for covering all statements 1-10.

After statement	a	b	x	y	Condition
Initial	A	B			
int x = a – b (1)			A - B		
int y = a + b + 3 (2)				A + B + 3	
if(a % 2 == 0){ (3)					(A % 2 == 0)
a = b++; (4)	B	B + 1			
if(y > 0) (5)					(A + B + 3 > 0)
x = y – 2; (6)			A + B + 1		
} (7)					
if(a + 6 > x) (8)					(B + 6 > A + B + 1)
b = 5; (9)		5			
return y + a + b; (10)					

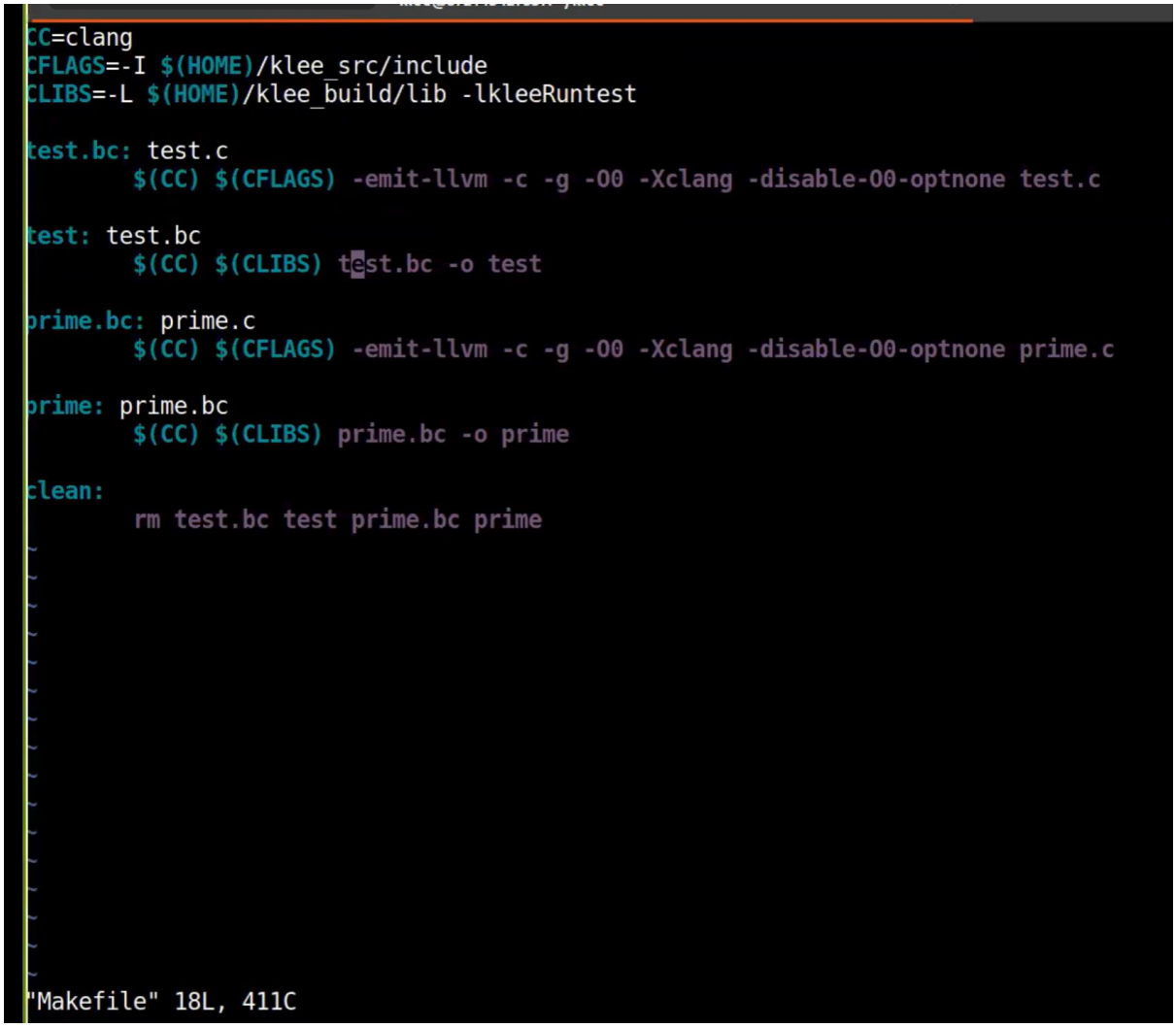
3. Use symbolic execution to determine the path condition for covering all statements 1-10.
   - Path condition: (A % 2 == 0) && (A + B + 3 > 0) && (B + 6 > A + B + 1)
4. What is the symbolic return value for this path?
   - (A + B + 3) + B + 5 = A + 2 * B + 8
5. Create three test cases that satisfy the path condition.
   - TC1 = { *fun*(**a** = 2, **b** = 1), 12 }
   - TC2 = { *fun*(**a** = -2, **b** = 5), 16 }
   - TC3 = { *fun*(**a** = 0, **b** = 0), 8 }
6. What problems can arise when trying to find test cases that fulfill a path condition for a given program?
   - Path conditions may be to hard to be solved (e.g. including cryptographic hash functions)
   - How often should we run loops?
7. Which test selection criterion can we use?
   - We can use e.g. path coverage when we do not have an infinite number of paths.

Exercise sheet 8 – Exercise 2: Symbolic Execution with KLEE

1. Install KLEE.
2. Create a C file that contains the function *fun* and make the function arguments symbolic.
3. Compile the file and execute KLEE on the generated bit-code file.
4. Check which variables assignments of *a*, *b* satisfy which path condition by using *ktest-tool*.
5. Re-compile the file with the library *libkleeRuntest.so* and replay the generated test cases with the environment variable *KTEST_FILE*.
6. Use KLEE to symbolically execute the function *is_prime*. Which problem did occur?

→ We will look at this in the terminal …

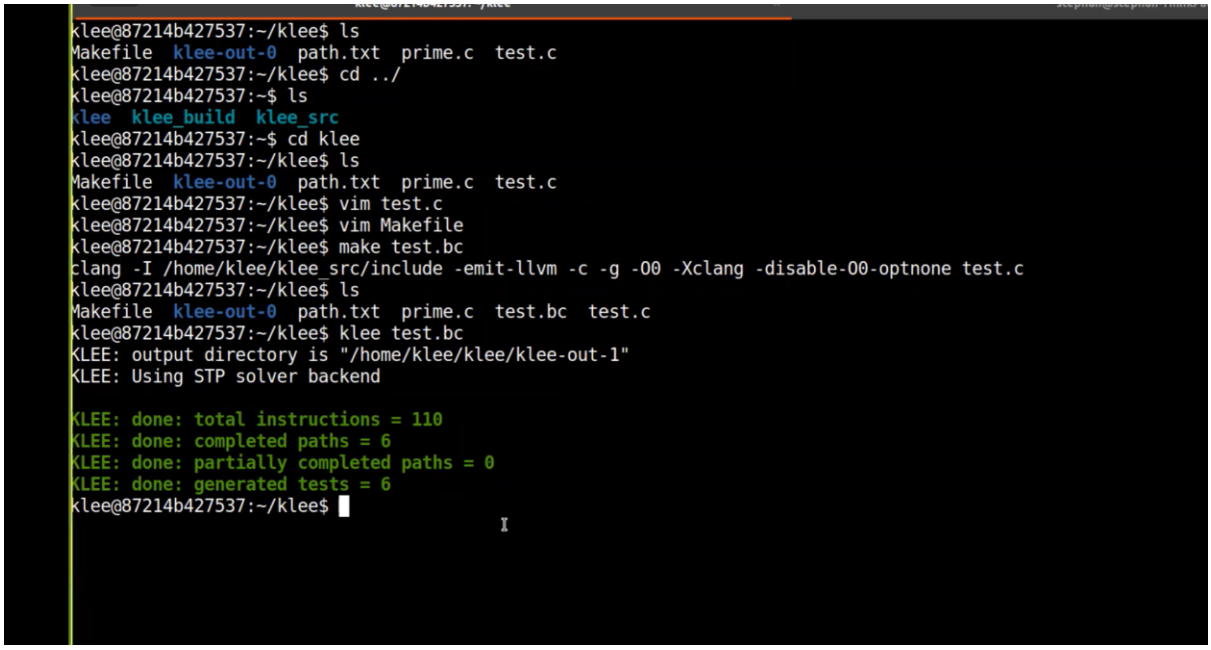
```c
#include "klee/klee.h"


int fun(int a, int b)
{
    int x = a - b;
    int y = a + b + 3;

    if (a % 2 == 0)
    {
        a = b++;
        if (y > 0)
            x = y - 2;
    }

    if (a + 6 > x)
        b = 5;

    return y + a + b;
}

int main(void) {
    int a, b;

    klee_make_symbolic(&a, sizeof(a), "a");
    klee_make_symbolic(&b, sizeof(b), "b");

    return fun(a, b);
}
~
~
~
~
~
"test.c" 29L, 389C
```

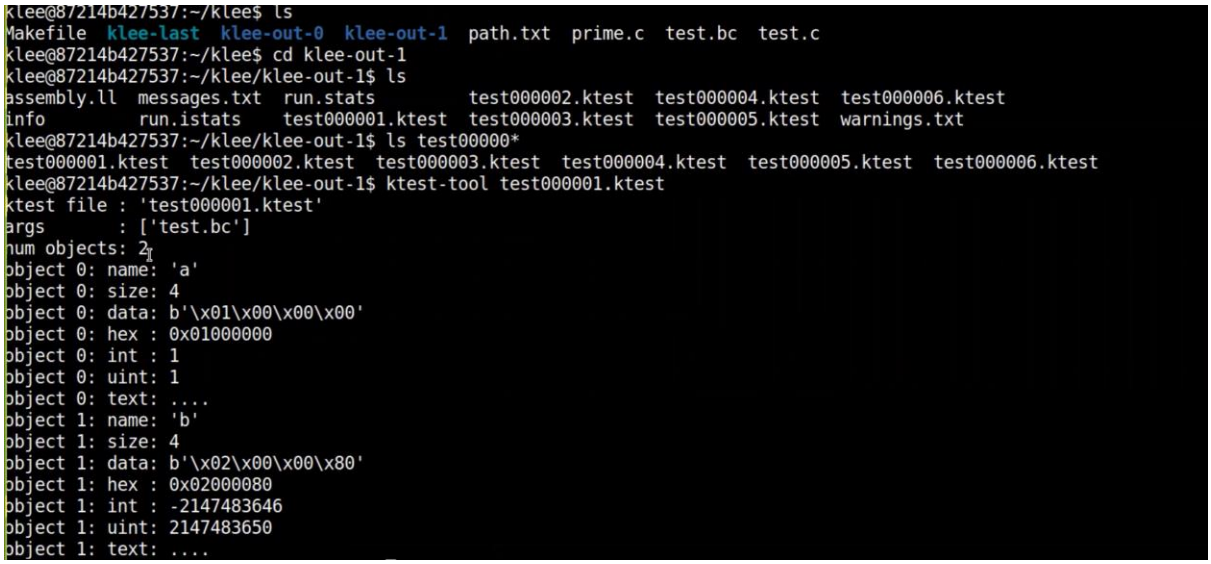


Because even if you test all possible *paths*, you still haven't tested them with all possible *values* or all possible *combinations of values*.

program testing may convincingly demonstrate the presence of bugs but can never demonstrate their absence.

Coverage does *not* tell you what code was tested, it tells you what code was *executed*.





Probs with symbolic execution:

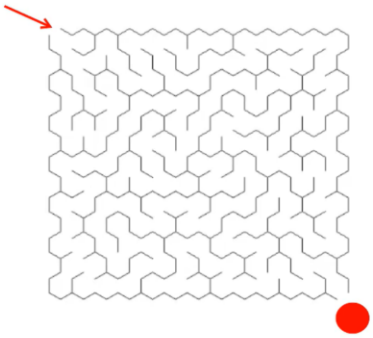
Path explosion, the STP solver may never find the correct test cases

If we have libraries functions with source code not directly available then can't be solved

Oracle problem in cases other than for eg. Robustness testing

6.5. Generation Technology - Test Case Generation

- Search problem: find a state or an output or parameters (CPS)
- Techniques
  - Dedicated algorithms for dedicated criteria
  - (Bounded) model checking
  - Symbolic execution
  - Heuristic Search pioneered by *[Bühler&Wegener 2004]*; *newer examples: [Ben Abdessalem et al.2018, Hauer et al. 2019] discussed in chapter "Test Selection II"*



Heuristic search: simulated annealing, genetic algos, ant colony optimization

Coverage criteria always boils down to partition based testing which is not better than random testing unless the partitions are based upon a defect hypothesis.

Our test cases are as good as our specification.

Model based testing : building, maintaining, validating the model

In practice we are always able to find the right level of abstraction so that we don't miss on testing important details.

Model based testing: good for cyber physical systems, OO software

Fault localization:

For single fault: based on coverage of test cases

Code metrics

Code churn: how often parts of a program have changed, the more people work on a piece of a code

Spectrum based fault localization: divide program into syntactic blocks and try to find the block which causes failure. Good for fault clustering.

Metrics can help to determine if fault localization will work

There is correlation between different complexity metrics and defect density.

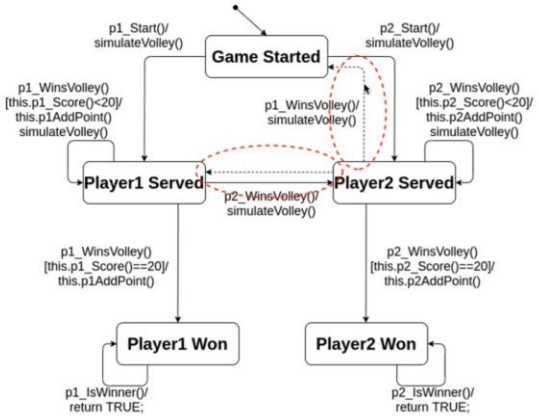
Exercise sheet 9 – Exercise 1: Testing OO Software

1. Name and describe the main characteristics of OO software that impact testing.

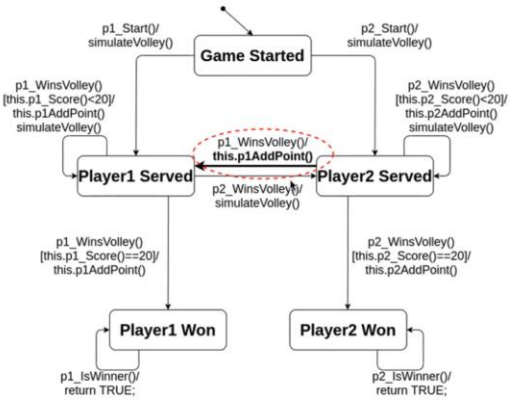
OO software characteristics that impact testing:

- State dependent behavior
- Encapsulation
- Inheritance
- Polymorphism and dynamic binding
- Abstract and generic classes
- Exception handling

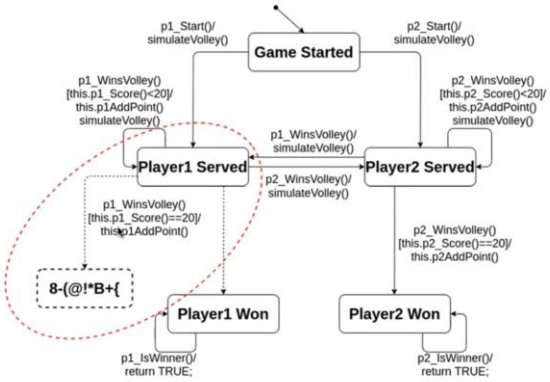
- Missing or incorrect transition or event
  - Resulting state is incorrect yet not corrupt



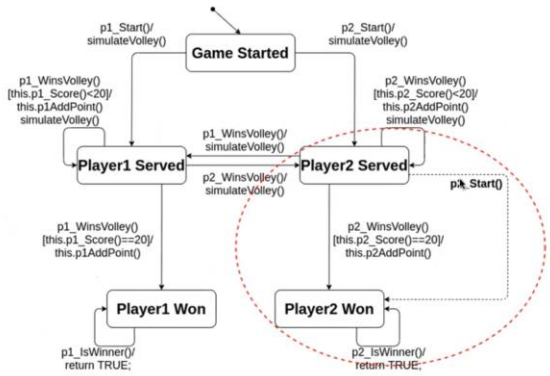
- Missing or incorrect transition or event
  - Resulting state is incorrect yet not corrupt
- **Missing or incorrect action**
  - **Wrong thing happens when executing a transition**



- Missing or incorrect transition or event
  - Resulting state is incorrect yet not corrupt
- Missing or incorrect action
  - Wrong thing happens when executing a transition
- **Extra, missing, or corrupt state**
  - **Behavior becomes unpredictable**



- Missing or incorrect transition or event
  - Resulting state is incorrect yet not corrupt
- Missing or incorrect action
  - Wrong thing happens when executing a transition
- Extra, missing, or corrupt state
  - Behavior becomes unpredictable
- **Sneak path or illegal message failure**
  - **Message accepted when it shouldn't or unexpected message causes a failure**



3. Name and describe criteria that can be used to select & derive test cases from an EFSM.

- State coverage
  - Cover all states in the (finite) state machine (FSM)
- Transition coverage
  - Cover all transitions in the FSM
- Transition path coverage
  - Cover all possible paths of transitions in the FSM

4. Describe 5 fault types that should be checked when testing OO software with deep inheritance hierarchies.

- Incorrect initialization
- Inadvertent bindings
- Naked access
- Naughty children
- Worm holes

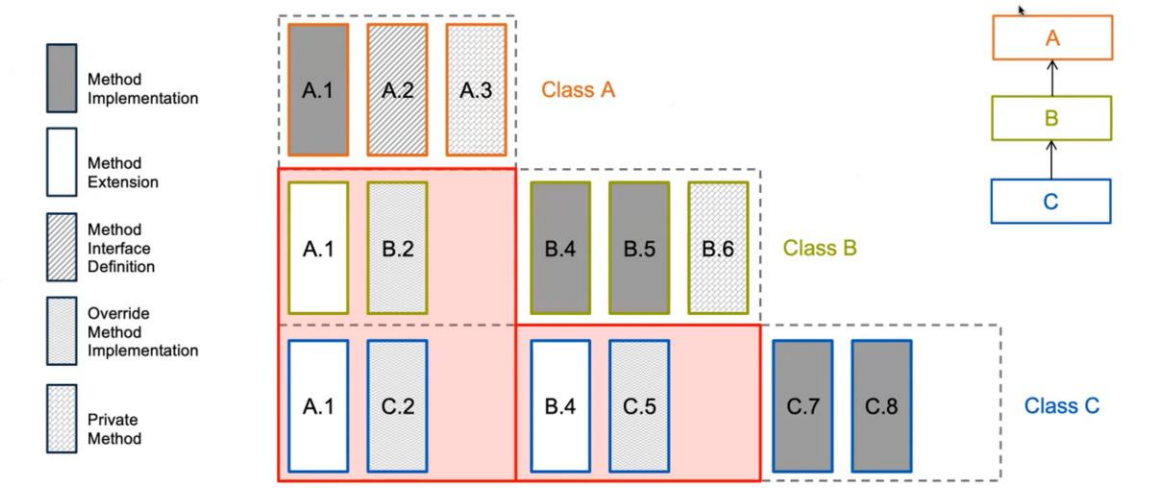
If we don't override an inherited method

If we directly access members of a class instead of using getters and setters

Naughty children: the subclass makes the object illegal for the superclass and violates the base class invariance property

5. Describe the core idea of flattening resp. how it can be used to check for OO-specific fault types.

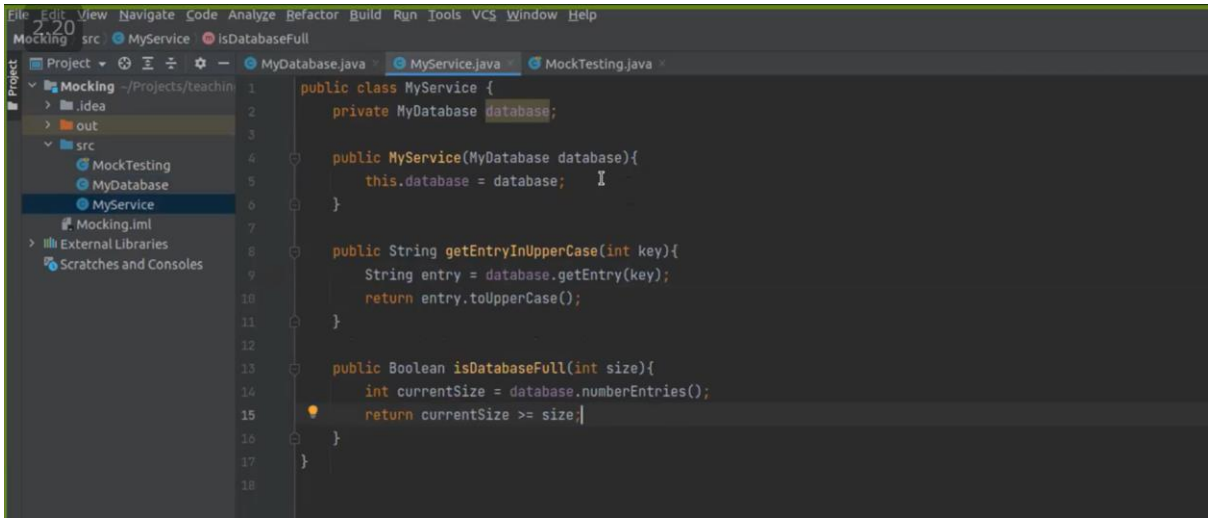
- Inheritance implemented by three mechanisms
  - **Extension:** Automatic inclusion of superclass features
  - **Overriding:** Same method in super- and subclass
  - **Specialization:** Addition of class features
- Flattened class makes all inherited features explicit
  - Precise inheritance semantics depends on language (and/or compiler)

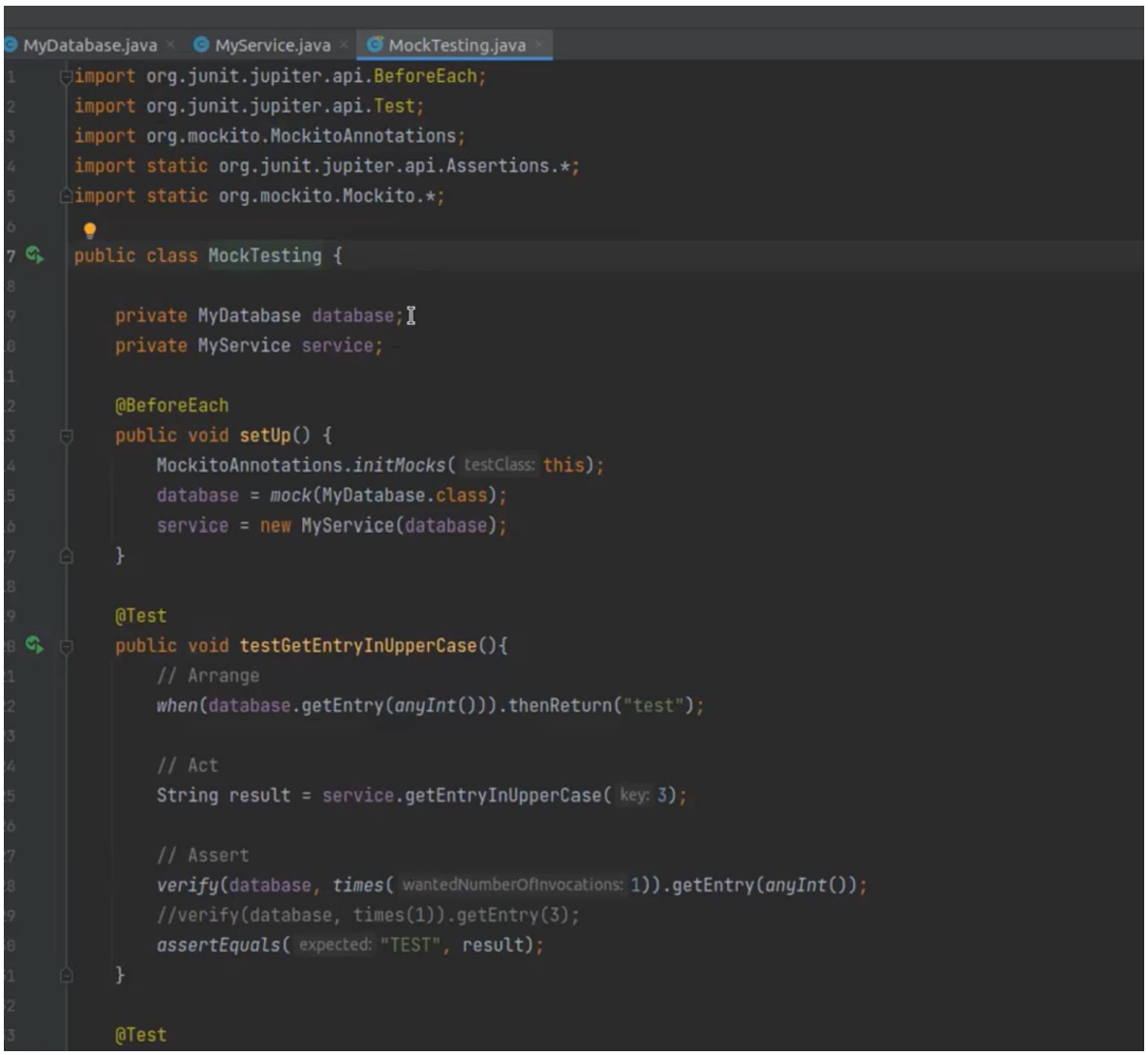


Combinatorial explosion

```java
import java.util.Map;

public class MyDatabase {

    private Map<Integer, String> content;

    public MyDatabase(){
        content = new HashMap<>();
    }

    public void addEntry(int key, String value){
        content.put(key, value);
    }

    public String getEntry(int key){
        return content.get(key);
    }

    public int numberEntries(){
        return content.size();
    }
}
```

```
Tests passed: 4 of 4 tests – 309 ms
309 ms  /usr/lib/jvm/java-8-openjdk-amd64/bin/java ...
309 ms
eStarting 303 ms
se()  2 ms
()  2 ms
)  2 ms

Process finished with exit code 0
```


2.20
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Mocking src MyService isDatabaseFull
Project
Mocking ~/Projects/teachin
.idea
out
src
MockTesting
MyDatabase
MyService
Mocking.iml
External Libraries
Scratches and Consoles
MyDatabase.java
MyService.java
MockTesting.java
public class MyService {
private MyDatabase database;
public MyService(MyDatabase database){
this.database = database;
}
public String getEntryInUpperCase(int key){
String entry = database.getEntry(key);
return entry.toUpperCase();
}
public Boolean isDatabaseFull(int size){
int currentSize = database.numberEntries();
return currentSize >= size;
}
}

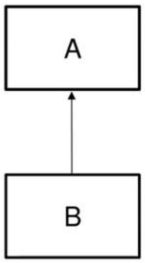

MyDatabase.java
MyService.java
MockTesting.java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.MockitoAnnotations;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;
public class MockTesting {
private MyDatabase database;
private MyService service;
@BeforeEach
public void setUp() {
MockitoAnnotations.initMocks(testClass: this);
database = mock(MyDatabase.class);
service = new MyService(database);
}
@Test
public void testGetEntryInUpperCase(){
// Arrange
when(database.getEntry(anyInt())).thenReturn("test");
// Act
String result = service.getEntryInUpperCase(key: 3);
// Assert
verify(database, times(wantedNumberOfInvocations: 1)).getEntry(anyInt());
//verify(database, times(1)).getEntry(3);
assertEquals(expected: "TEST", result);
}
@Test

Exercise sheet 10 – Exercise 1: Testing with Inheritance

1. Explain the Liskov substitution principle (LSP) briefly.

Properties that hold for an object of **base class A** must also hold for objects of **subclass B**.

*=> Any object of **base class A** must be replaceable by objects of **subclass B** without violating the properties that hold for **A**.*



Exercise sheet 10 – Exercise 1: Testing with Inheritance

2. Specify the pre- and post-condition of the method *addAccount* in *AccountManager*.

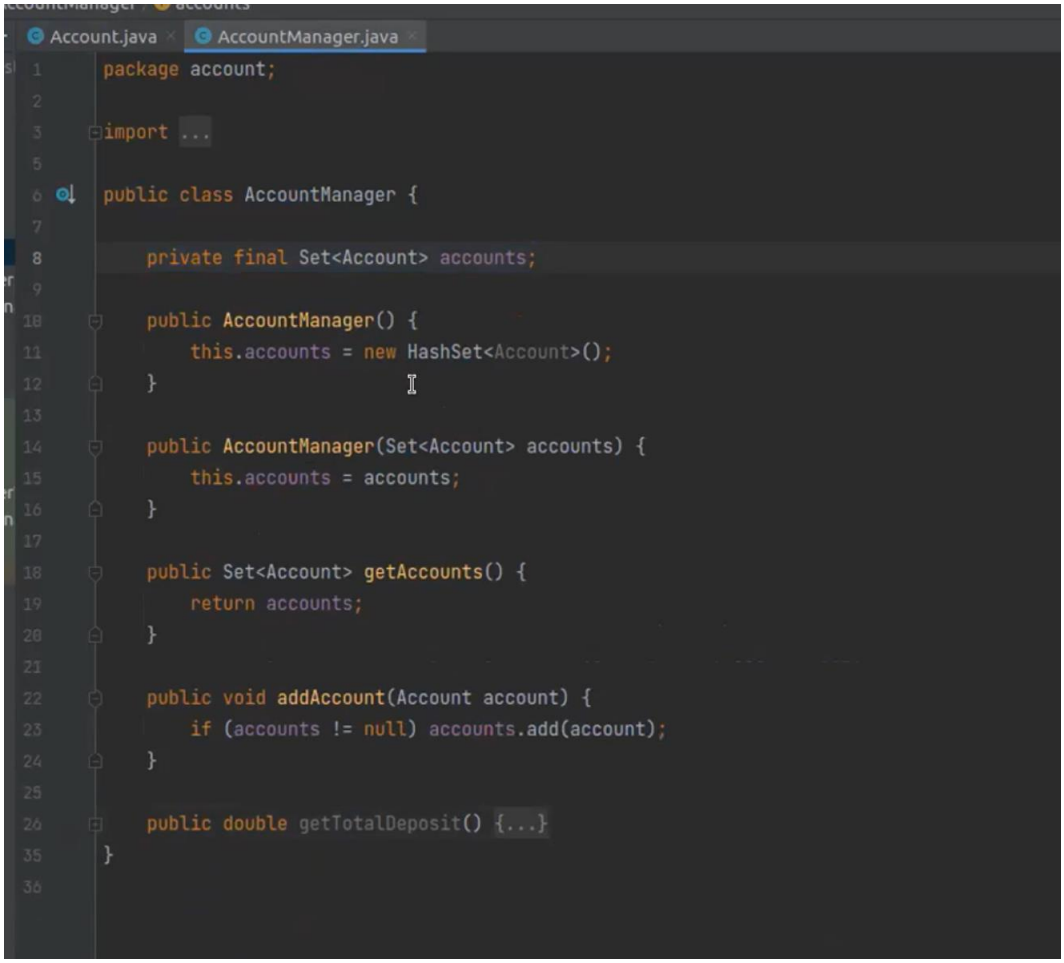
Each *account* object is defined by "**id**", "**name**" and "**deposit**".

```
public void addAccount(Account account)
  // pre: !accounts.contains(account) && accounts.size() == N
  // post: accounts.contains(account) && accounts.size() == N + 1
{
  ...
}
```

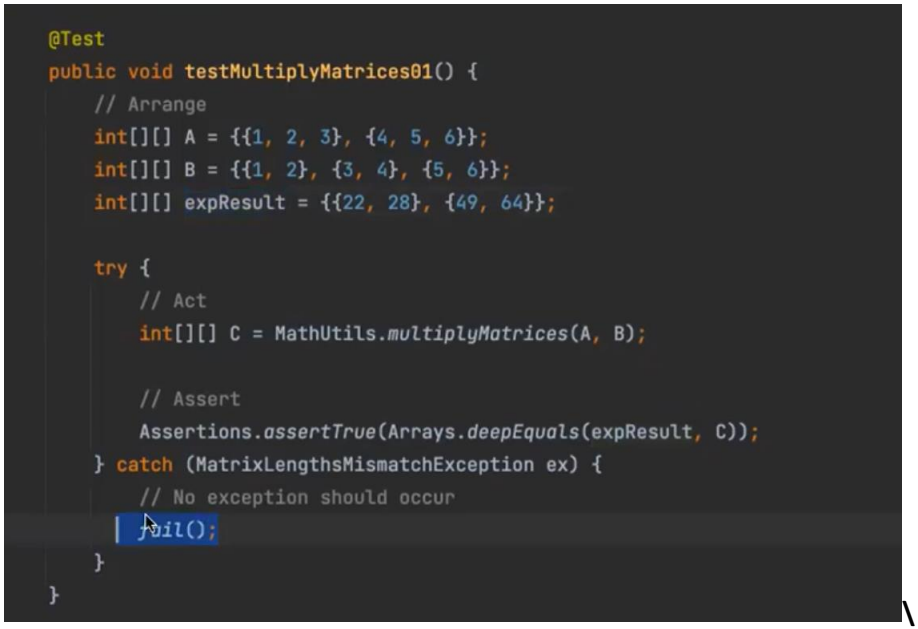
Account.java | AccountManager.java

```java
package account;

public class Account {

    private final int id;

    private final String name;

    private final double deposit;

    public Account(int id, String name, double deposit) {
        this.id = id;
        this.name = name;
        this.deposit = deposit;
    }

    public int getId() { return id; }

    public String getName() { return name; }

    public double getDeposit() { return deposit; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Account account = (Account) o;

        if (id != account.id) return false;
        if (Double.compare(account.deposit, deposit) != 0) return false;
        return name != null ? name.equals(account.name) : account.name == null;
    }
```

```java
@Override
public int hashCode() {
    int result;
    long temp;
    result = id;
    result = 31 * result + (name != null ? name.hashCode() : 0);
    temp = Double.doubleToLongBits(deposit);
    result = 31 * result + (int) (temp ^ (temp >>> 32));
    return result;
}
```



Liskov Substitutiom : objects of superclasses should be able to be replaced by objects of subclass without breaking the application



\

```java
@Test
public void testMultiplyMatrices02() {
    // Arrange
    int[][] A = {{1, 2}, {3, 4}};
    int[][] B = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

    // Act + Assert
    try {
        int[][] C = MathUtils.multiplyMatrices(A, B);
        fail("Exception expected!");
    } catch (Exception ex) {
        Assertions.assertSame(ex.getClass(), MatrixLengthsMismatchException.class);
    }
}
```

1. Describe the purpose of using Delta Debugging. Does Delta Debugging always result in a globally optimal solution?
   - To minimize the test case input to a minimal failure-inducing input.
   - No, the presented version finds a local optimum, but not necessarily a globally minimal input.
     - We will see this in exercise 2 as well.

Exercise sheet 11 – Exercise 1: From Failures to Faults

2. Explain in your own words the different fault localization methods that were introduced in the lecture.
   - Fault localization with program / hit spectra
     - Check how often each block of the program is visited by passing and failing test cases.
     - Use similarity metrics to create a ranked list of similar blocks.
     - But, which metric to use?
   - Fault localization based on code metrics
     - Do certain code properties affect software failures? E.g., code complexity, problem domain, past history, or process quality.
     - Case study reveals significant correlation between complexity metrics and post-release defects.
     - But, different metrics correlate for each project.
   - Fault localization based on code churn
     - Is the amount of change in a module an indicator for its error-proneness?
     - Promising results for single projects, but further research needs to be done.

3. How does a confounding bias arise, and how can we tackle it?
   - Confounding bias: an independent part of the program is associated with a program failure
   - Example:

```
public void F1(int i){
    if (i < 0){
        …   // Faulty
        F2(x);
    } else {
        …
        F3(x);
    }
}
```

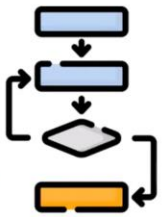
   - Tackle confounding bias: take dynamic calls and explicit data-dependency graphs of failing tests into account

4. What is the idea of failure clustering, and what do we aim to achieve with it?
   - When performing regression testing, often a large number of test is frequently executed.
     - A small number of faults often causes a large number of tests to fail.
   - Idea: Cluster failures to reduce the failure analysis time.
     - Cluster the failures such that one cluster encompasses failures that are caused by the same fault.
     - Then, investigate only one representative of each of these clusters to find the underlying fault.
   - We can use this approach also for test selection and prioritization.

Apply the Delta Debugging algorithm to the 16-character input "a-debugging-exam" to find a minimal input that still causes function *func* to return *true*.

Let in be the input part of a test case that triggers a failure. We want to minimize in so that the failure persists. Let $test$ and in be given such that $test(\emptyset) = PASS$ and $test(in) = FAIL$ hold. Let N be the number of subsets.

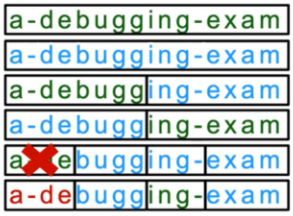
First we split in into $N = 1$ subsets s_i, then we take a look at each subset s_i:



- 1. We test the complement $c_i = in - s_i$ of the currently regarded subset s_i.
- 2. If the complement c_i is not failure-causing ($test(c_i) = PASS$):
  - We try the next complement c_{i+1}.
- 3. If the complement c_i is failure-causing ($test(c_i) = FAIL$):
  - We can ignore everything outside this complement.
  - If $N > 2$, we split the complement c_i in $N - 1$ subsets, otherwise we split in $N = 2$ subsets. Then repeat.
    - If $N \leq 2$ and the subsets include only one character each before splitting, stop.
    - Why $N - 1$? Testing results of earlier invocation can be re-used!
- 4. If we run out of complements, we have to increase the granularity (try again by splitting into $2 * N$ subsets).
- 5. Repeat until granularity cannot be increased. Then, we have tried to remove any single element of the input.

Icon made by Freepik from www.flaticon.com

Algorithm: 3. If the complement is failure-causing, we ignore everything outside, split in $N-1$ subsets & recurse.



P	$N = 1$
F	$N = 1$
P	$N = 2$
P	$N = 2$
F	$N = 4$
P	$N = 3$

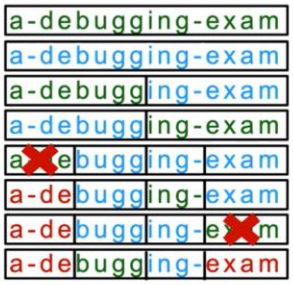
"ing-exam" was already tested
→ regard next complement

we currently test these characters
we currently do not test these chars
these chars are not tested anymore

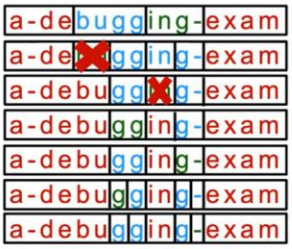
P: the test passes (tested chars not failure-causing)
F: the test fails (tested chars are failure-causing →
here: # 'g' >= 3 or # 'e' >= 2)

Exercise sheet 11 – Exercise 2: Delta Debugging

Algorithm: 2. If the complement is not failure-causing, we try the next complement.



P $N = 1$
F $N = 1$
P $N = 2$
P $N = 2$
F $N = 4$
P $N = 3$
F $N = 3$
P $N = 2$



P $N = 2$
F $N = 4$
F $N = 3$
P $N = 2$
P $N = 2$
P $N = 4$
F $N = 4$

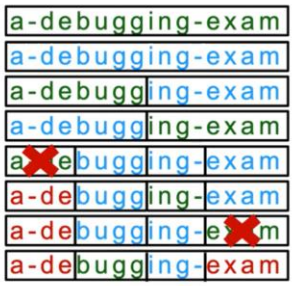
"gg-" was already tested → regard next complement

we currently test these characters
we currently do not test these chars
these chars are not tested anymore

P: the test passes (tested chars not failure-causing)
F: the test fails (tested chars are failure-causing → here: # 'g' >= 3 *or* # 'e' >= 2)

Exercise sheet 11 – Exercise 2: Delta Debugging

Algorithm: 3. If the complement is failure-causing, we ignore everything outside, split in $N - 1$ subsets & recurse.



P $N = 1$
F $N = 1$
P $N = 2$
P $N = 2$
F $N = 4$
P $N = 3$
F $N = 3$
P $N = 2$

P $N = 2$
F $N = 4$
F $N = 3$

"ing-" was already tested
→ regard next complement

we currently test these characters
we currently do not test these chars
these chars are not tested anymore

P: the test passes (tested chars not failure-causing)
F: the test fails (tested chars are failure-causing → here: # 'g' >= 3 *or* # 'e' >= 2)

Exercise sheet 11 – Exercise 3: Fault Localization Using Hit Spectra

1. Create a hit spectrum for the method "RationalSort" and the test cases T1 to T6.

Block	Program Elements	T1	T2	T3	T4	T5	T6
1	int i, j, temp; for(i = n - 1; i >= 0; i--){	X	X	X	X	X	X
2	for(j = 0; j < i; j++){		X	X	X	X	X
3	if(RationalGT(num[j], den[j], num[j+1], den[j+1]){			X	X	X	X

4	temp = num[j];
	num[j] = num[j+1];
	num[j+1] = temp;
	}
	}
	}
				X	X	X	

	Pass (P) /Fail (F) of Test Case Execution	P	P	P	P	F	P

$T1 = \langle\ \rangle$

$T2 = \left\langle \frac{1}{5} \right\rangle$

$T3 = \left\langle \frac{6}{2}, \frac{3}{2} \right\rangle$

$T4 = \left\langle \frac{4}{1}, \frac{2}{2}, \frac{0}{1} \right\rangle$

$T5 = \left\langle \frac{3}{1}, \frac{2}{2}, \frac{4}{3}, \frac{1}{4} \right\rangle$

$T6 = \left\langle \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{1}{1} \right\rangle$

Exercise sheet 11 – Exercise 3: Fault Localization Using Hit Spectra

2. Compute the Jaccard score for all blocks.

a_{11}: # of times block is executed in failing test cases;
a_{10}: # of times block is executed in passing test cases;
a_{01}: # of times block is not executed in failing test cases;
a_{00}: # of times block is not executed in passing test cases;

- Jaccard metric: $s_j = \frac{a_{11}}{a_{11} + a_{01} + a_{10}}$
- $s_1 = \frac{a_{11}}{a_{11} + a_{01} + a_{10}} = \frac{1}{1+0+5} = \frac{1}{6} = 0.17$
- $s_2 = \frac{a_{11}}{a_{11} + a_{01} + a_{10}} = \frac{1}{1+0+4} = \frac{1}{5} = 0.20$
- $s_3 = \frac{a_{11}}{a_{11} + a_{01} + a_{10}} = \frac{1}{1+0+3} = \frac{1}{4} = 0.25$
- $s_4 = \frac{a_{11}}{a_{11} + a_{01} + a_{10}} = \frac{1}{1+0+2} = \frac{1}{3} = 0.33$

Block	T1	T2	T3	T4	T5	T6
1	X	X	X	X	X	X
2		X	X	X	X	X
3			X	X	X	X
4			X	X	X	
	P	P	P	P	F	P

Exercise sheet 11 – Exercise 3: Fault Localization Using Hit Spectra

4. Which block is most likely to be failure-causing according to each of these metrics?
   - The blocks with the highest suspiciousness scores are the most likely to be failure-causing.
   - Block 4 has the highest score in both metrics.
   - But, we don't know a-priori which metric will perform good for our program and which not!

Model-Based Testing is (1) the derivation of test cases (input and expected output) from models of a SUT and its environment on the grounds of test case specifications, and (2) the execution of these tests on the SUT after bridging the levels of abstraction between SUT and model.