Programm- & Systemverifikation

Testing

Georg Weissenbacher

184.741

TU WIEN

What happened so far

- How bugs come into being:
  - **Fault** – cause of an error (e.g., mistake in coding)
  - **Error** – *incorrect* state that may lead to failure
  - **Failure** – deviation from *desired* behaviour
- We specified *intended* behaviour using **assertions**
- We even proved our programs correct (inductive invariants).

Flashback: Inductive Assertions

- An assertion is an (loop) invariant if
  - it holds upon loop entry
  - remains true after each iteration of the loop
- An invariant is *inductive*
  - if its validity upon loop entry is sufficient to guarantee that it still holds after the iteration

Flashback: Inductive Assertions

```
int x = 2;
while (x < 100)
{
  assert (x > 0);
  x = 2 * x - 2;
}
```

- `(x > 0)` is an invariant.
- But is it inductive?

Flashback: Inductive Assertions

```
int x = 2;
while (x < 100)
{
  assert (x > 0);
  x = 2 * x - 2;
}
```

- `(x > 0)` is an invariant.
- But is it inductive?
  - Does the loop condition (`x < 100`) and the assertion (`x > 0`) guarantee that (`x > 0`) holds after iteration?

Flashback: Inductive Assertions

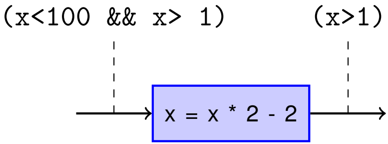
```
int x = 2;
while (x < 100)
{
  assert (x > 0);
  x = 2 * x - 2;
}
```

- `(x > 0)` is an invariant.
- But is it inductive?
  - Does the loop condition ($x < 100$) and the assertion ($x > 0$) guarantee that ($x > 0$) holds after iteration?
  - No! (try `x = 1`)

Flashback: Inductive Assertions

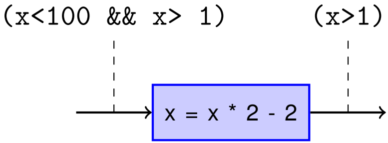
```
int x = 2;
while (x < 100)
{
  assert (x > 1);
  x = 2 * x - 2;
}
```

- `(x > 1)` is an invariant.
- But is it inductive?



Flashback: Inductive Assertions

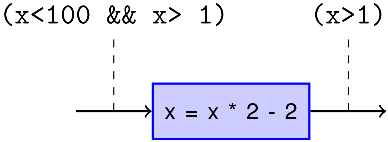
- `(x > 1)` is an invariant.
- But is it inductive?



- In which cases is `(x>1)` true after `x = x * 2 - 2`

Flashback: Inductive Assertions

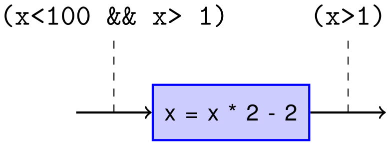
- `(x > 1)` is an invariant.
- But is it inductive?



- In which cases is `(x>1)` true after `x = x * 2 - 2`
  - if (and only if) `(x * 2 - 2 > 1)` holds before

Flashback: Inductive Assertions

- (x > 1) is an invariant.
- But is it inductive?



- In which cases is (x>1) true after x = x * 2 - 2
  - if (and only if) (x * 2 - 2 > 1) holds before
  - (guaranteed by $2 \leq x \leq 99$)

Flashback: Inductive Assertions

- Assertions *implied* by an inductive invariant are invariants
  - e.g., `(x>0)` is implied by `(x>1)`
  - Why?

Flashback: Inductive Assertions

- Assertions *implied* by an inductive invariant are invariants
  - e.g., `(x>0)` is implied by `(x>1)`
  - Why? Whenever inductive invariant holds, its implication holds

Flashback: Inductive Assertions

- Our proof technique is currently very limited!
  - We don't even know yet how to deal with `if(...)`
- Will revisit this topic in later lectures:
  - More formal proof-framework: *Hoare* logic

What happened so far

- How bugs come into being:
  - **Fault** – cause of an error (e.g., mistake in coding)
  - **Error** – *incorrect* state that may lead to failure
  - **Failure** – deviation from *desired* behaviour
- We specified *intended* behaviour using **assertions**
- We even proved our programs correct (inductive invariants).

"Beware of bugs in the above code; I have only proved it correct, not tried it"

(Donald Knuth)

What happened so far

- How bugs come into being:
  - **Fault** – cause of an error (e.g., mistake in coding)
  - **Error** – *incorrect* state that may lead to failure
  - **Failure** – deviation from *desired* behaviour
- We specified *intended* behaviour using **assertions**
- We even proved our programs correct (inductive invariants).

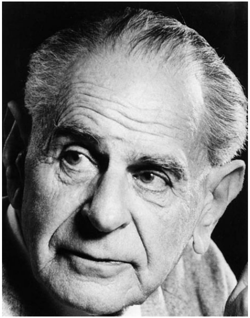
"Beware of bugs in the above code; I have only proved it correct, not tried it"

(Donald Knuth)

(Mathematical) proofs often contain implicit assumptions, may need to be revised!

(c.f. Lakatos, "Proofs and refutations")

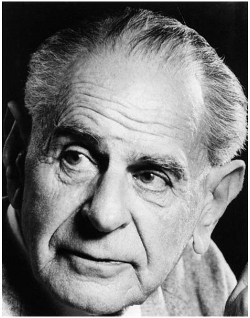
Empirical Falsification



"Good tests kill flawed theories; we remain alive to guess again."

(Sir Karl Popper)

Empirical Falsification



"Good tests kill flawed theories; we remain alive to guess again."

"In so far as a scientific statement speaks about reality, it must be *falsifiable*; and in so far as it is not falsifiable, it does not speak about reality."

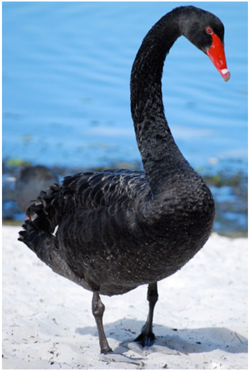
(Sir Karl Popper)

Empirical Falsification

- A statement or theory (about the empirical world)
  - can never be proven ultimately correct
  - is only meaningful if it can be put to the test

Empirical Falsification

- A statement or theory (about the empirical world)
  - can never be proven ultimately correct
  - is only meaningful if it can be put to the test

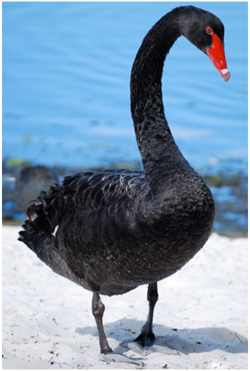


"All swans are white"

- Northern Hemisphere species have white plumage

Empirical Falsification

- A statement or theory (about the empirical world)
  - can never be proven ultimately correct
  - is only meaningful if it can be put to the test



"All swans are white"

- Northern Hemisphere species have white plumage
- Southern hemisphere species are mixed *black* and white!

Empirical Falsification

- Statements can never be proven ultimately correct
  - can only increase confidence in validity
- A statement is only meaningful if it is *falsifiable*
  - if it is false, this can be shown by observation or experiment

Empirical Falsification

"Statements can never be proven ultimately correct"

- What about formal proofs?
  - Realistic programs are too large and complex; can't be proven correct entirely
  - Even proofs rely on *abstractions* and *assumptions*

Empirical Falsification

“A statement is only meaningful if it is *falsifiable*”

- Think of “statement” as a specification/requirement!
- A requirement is falsifiable only if there exists a way of checking whether it is satisfied
  - Can you think of specifications that are not falsifiable?

Empirical Falsification

"A statement is only meaningful if it is *falsifiable*"

- Think of "statement" as a specification/requirement!
- A requirement is falsifiable only if there exists a way of checking whether it is satisfied
  - Can you think of specifications that are not falsifiable?
    - The software shall be fast.

Empirical Falsification

"A statement is only meaningful if it is *falsifiable*"

- Think of "statement" as a specification/requirement!
- A requirement is falsifiable only if there exists a way of checking whether it is satisfied
  - Can you think of specifications that are not falsifiable?
    - The software shall be fast.
    - The user interface shall look good.

Empirical Falsification

"A statement is only meaningful if it is *falsifiable*"

- Think of "statement" as a specification/requirement!
- A requirement is falsifiable only if there exists a way of checking whether it is satisfied
  - Can you think of specifications that are not falsifiable?
    - The software shall be fast.
    - The user interface shall look good.
  - Are *assertions* falsifiable?
    - Yes. If they fail, there is a *counterexample*.

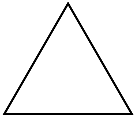
How to “verify” if we can’t verify

- Increase *confidence* in correctness
- This is a time consuming process:
  - 50%-70% of development time spent on testing and validation

Topic of this Lecture

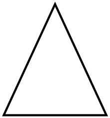
- Testing
  - Analyse *subset* of all behaviours
  - Goal: *falsify*, rather than prove absence of bugs

Example [G. Myers, “Art of Software Testing”]



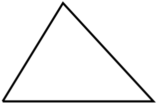
Equilateral Triangle

- 3 equal sides
- 3 equal angles



Isosceles Triangle

- 2 equal sides
- 2 equal angles



Scalene Triangle

- 0 equal sides
- 0 equal angles

Example [G. Myers, “Art of Software Testing”]

```
typedef enum { SCALENE = 0,
               ISOSCELES = 2,
               EQUILATERAL = 3,
               INVALID } Triangle;

Triangle classify (float a, float, b, float c);
```

- How would you test the implementation of `classify`?

Test-Cases for Triangle Classification

- *Valid* scalene triangle
  - (1,2,3) and (2,5,9) does not count!
- *Valid* equilateral triangle
- *Valid* isosceles triangle
  - (2,2,4) does not count!

Test-Cases for Triangle Classification

- *Valid* scalene triangle
  - (1,2,3) and (2,5,9) does not count!
- *Valid* equilateral triangle
- *Valid* isosceles triangle
  - (2,2,4) does not count!
- *Three* test-cases representing isosceles triangle
  - all three permutations, e.g., (3,3,4), (3,4,3), and (4,3,3)?

Test-Cases for Triangle Classification

- *Valid* scalene triangle
  - (1,2,3) and (2,5,9) does not count!
- *Valid* equilateral triangle
- *Valid* isosceles triangle
  - (2,2,4) does not count!
- *Three* test-cases representing isosceles triangle
  - all three permutations, e.g., (3,3,4), (3,4,3), and (4,3,3)?
- Test case with one side of length *zero*?
  - Ideally: check all 3 sides separately

Test-Cases for Triangle Classification

- *Valid* scalene triangle
  - (1,2,3) and (2,5,9) does not count!
- *Valid* equilateral triangle
- *Valid* isosceles triangle
  - (2,2,4) does not count!
- *Three* test-cases representing isosceles triangle
  - all three permutations, e.g., (3,3,4), (3,4,3), and (4,3,3)?
- Test case with one side of length *zero*?
  - Ideally: check all 3 sides separately
- Test case with one side of *negative* length?
  - Ideally: check all 3 sides separately

Test-Cases for Triangle Classification (continued)

- Inputs a, b, c such that $a + b = c$
  - it's a bug if `classify` returns `SCALENE`!

Test-Cases for Triangle Classification (continued)

- Inputs a, b, c such that $a + b = c$
  - it's a bug if `classify` returns `SCALENE`!
  - Try all 3 permutations

Test-Cases for Triangle Classification (continued)

- Inputs a, b, c such that $a + b = c$
  - it's a bug if `classify` returns `SCALENE`!
  - Try all 3 permutations
- Inputs a, b, c such that $a + b < c$
  - `classify` should return `INVALID`

Test-Cases for Triangle Classification (continued)

- Inputs a, b, c such that $a + b = c$
  - it's a bug if `classify` returns `SCALENE`!
  - Try all 3 permutations
- Inputs a, b, c such that $a + b < c$
  - `classify` should return `INVALID`
  - Try all 3 permutations

Test-Cases for Triangle Classification (continued)

- Inputs a, b, c such that $a + b = c$
  - it's a bug if `classify` returns `SCALENE`!
  - Try all 3 permutations
- Inputs a, b, c such that $a + b < c$
  - `classify` should return `INVALID`
  - Try all 3 permutations
- All sides set to zero

Test-Cases for Triangle Classification (continued)

- Inputs a, b, c such that $a + b = c$
  - it's a bug if `classify` returns `SCALENE`!
  - Try all 3 permutations
- Inputs a, b, c such that $a + b < c$
  - `classify` should return `INVALID`
  - Try all 3 permutations
- All sides set to zero
- At least one test-case with non-integer values

Test-Cases for Triangle Classification (continued)

- Specify output for each test-case!
  - Otherwise, it is not *falsifiable*

Questions about Testing

Before we learn *how* to test. . .

- *What* is testing
- *Who* should test
- *What* to test for
- *Where* to look for bugs
- *When* to stop

What is Testing?

- Execute program with the *intent* to find errors
  - Specify **test cases** (or **test scenarios**)
  - A collection of test-cases is a **test suite**
  - The execution of a test case is a **test run**

What is Testing?

- Execute program with the *intent* to find errors
  - Specify **test cases** (or **test scenarios**)
  - A collection of test-cases is a **test suite**
  - The execution of a test case is a **test run**
- Destructive, even sadistic process. [Myers]

What is Testing?

- Execute program with the *intent* to find errors
  - Specify **test cases** (or **test scenarios**)
  - A collection of test-cases is a **test suite**
  - The execution of a test case is a **test run**
- Destructive, even sadistic process. [Myers]
- Testing is *not* a proof of correctness.
  Even trivial programs have
  - infinitely many inputs
  - infinitely many executions/behaviours

Who should do Testing?

- Whenever you write a program, you already *implicitly* test
  - Unavoidable for debugging
  - However, this is not *systematic* testing

Who should do Testing?

- Whenever you write a program, you already *implicitly* test
  - Unavoidable for debugging
  - However, this is not *systematic* testing
- Thou shalt not test thy own software!
  - You are *biased* (coding is more fun than bug-fixing!)
  - You might have misunderstood the specification

Evaluate and Document Testing Results

- Expected result is necessary part of test-case (falsifiability!)

Evaluate and Document Testing Results

- Expected result is necessary part of test-case (falsifiability!)
- Thoroughly inspect the results of each test

Evaluate and Document Testing Results

- Expected result is necessary part of test-case (falsifiability!)
- Thoroughly inspect the results of each test
- Document the test results

Evaluate and Document Testing Results

- Expected result is necessary part of test-case (falsifiability!)
- Thoroughly inspect the results of each test
- Document the test results
  - Often required by quality assurance standards

Evaluate and Document Testing Results

- Expected result is necessary part of test-case (falsifiability!)
- Thoroughly inspect the results of each test
- Document the test results
  - Often required by quality assurance standards
- Add regression test

What to Test for

- Test whether the software does what it's supposed to do
  - in case of valid and expected, but also
  - invalid and unexpected inputs/conditions

What to Test for

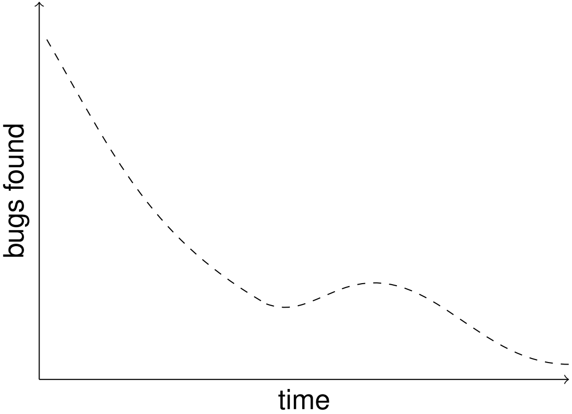
- Test whether the software does what it’s supposed to do
  - in case of valid and expected, but also
  - invalid and unexpected inputs/conditions
- Test whether it does what it’s not supposed to do
  - Unwanted side effects

Where to look for Bugs

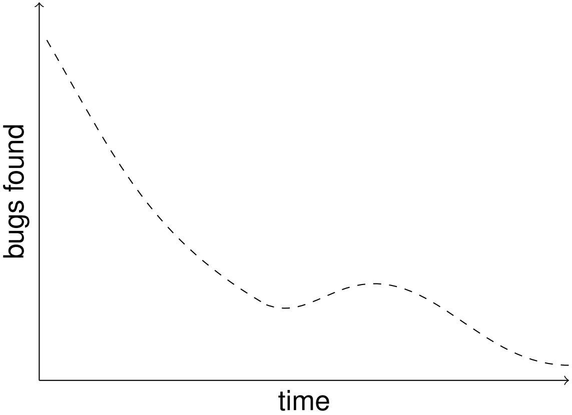
- Code sections in which you've already found bugs!
  - High probability there will be more
- Sections that *change* often
  - Can be determined using *versioning systems*
- Code with high complexity

> "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as *cleverly* as possible, you are, by definition, not smart enough to debug it."
>
> (Brian Kernighan)

When to Stop Testing



When to Stop Testing



There's no general answer, except: you're never 100% done

When to Stop Testing

Exit criteria should be defined by test-plan

- Bug detection ration drops under certain level
- No more high priority bugs
- Requirements sufficiently exercised through test-cases
- Coverage criteria reached (we'll hear about that later)
- Approaching deadline, budget depleted
- ...

Allow for enough time for testing!

Validation versus Verification

- *Validation:* Are we building the right system?
  - Do the requirements/the system satisfy the customer's needs?
- *Verification:* Are we building the system right?
  - Does the product satisfy the requirements/specification?

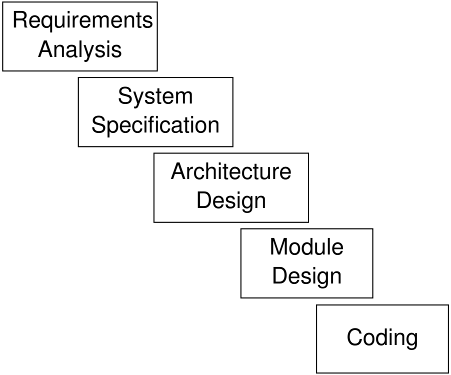
Validation versus Verification

- *Validation:* Are we building the right system?
  - Do the requirements/the system satisfy the customer's needs?
- *Verification:* Are we building the system right?
  - Does the product satisfy the requirements/specification?

Focus of this course: Verification

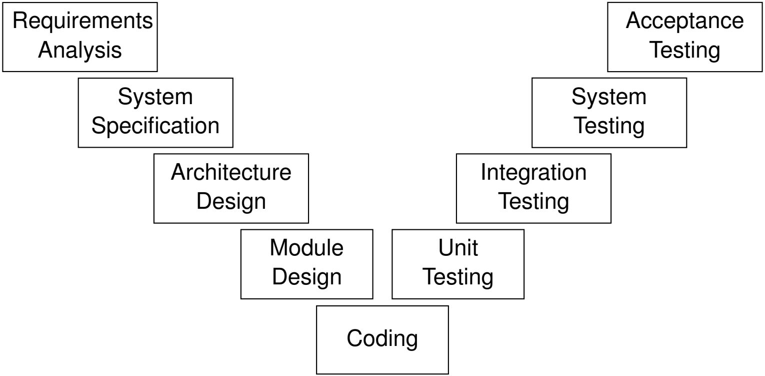
Testing in the Development Cycle

From the *waterfall model* . . .



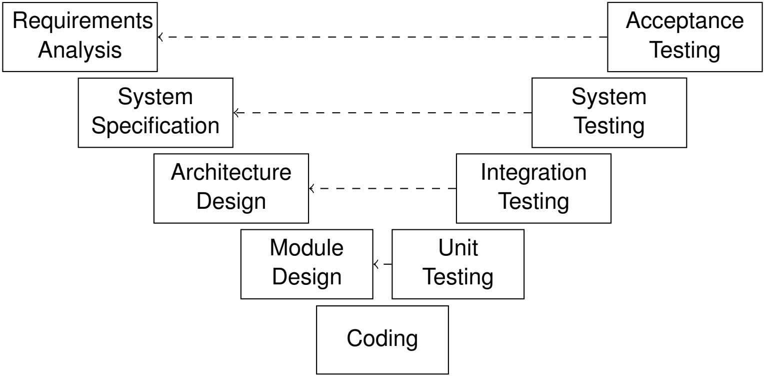
Testing in the Development Cycle

From the *waterfall model* . . . to the V-model



Testing in the Development Cycle

From the *waterfall model* . . . to the V-model



Testing in the Development Cycle

The V-model is simplistic; but: it identifies important phases:

- Unit (module) testing
  Testing of (small) components that are part of the system
- Integration testing
  Testing whether components work together
- System testing
  Testing of the entire system
- Acceptance testing
  Testing performed by customer/client
- Regression testing
  Testing performed after updates/fixes

  (also element of modern techniques such as extreme programming)

Unit Testing

- So, how do we find bugs in software modules?

Unit Testing

- So, how do we find bugs in software modules?

Code Inspections

- Bugs can be found by looking at the code
- Can be done
  - in solitude
  - in groups
- Can be

Code Inspections

- Bugs can be found by looking at the code
- Can be done
  - in solitude
  - in groups
- Can be
  - formal
    - meeting of software developers, designers, testers
    - review of code line by line (printed copies)
    - error check-lists
    - about 150 lines of code per hour
    - multiple phases

Code Inspections

- Bugs can be found by looking at the code
- Can be done
  - in solitude
  - in groups
- Can be
  - formal
    - meeting of software developers, designers, testers
    - review of code line by line (printed copies)
    - error check-lists
    - about 150 lines of code per hour
    - multiple phases
  - “lightweight”
    - Source code management notifies team about code commits
    - Pair programming (common in XP)
    - . . .

Code Inspections

Error checklists ([Myers79], includes bugs from lecture on “Bugs”)

- Arithmetic bugs
  - Underflow or overflow
  - Division by zero
  - Incorrect (automatic) conversions
  - Variables outside meaningful range
- Data declaration bugs
  - Uninitialised variables
  - Arrays and strings properly initialised?
  - Correct typing of variables
  - Variable names (are there similarities?)

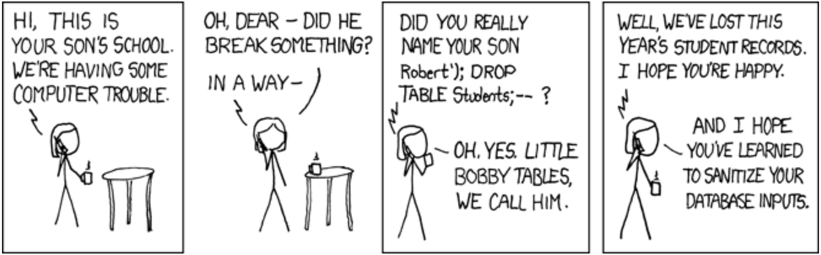
Code Inspections

- Comparisons
  - Comparisons and relations correct? (order of parameters)
  - Boolean expressions correct?
  - Operator precedence
    `(a && b || c)` or `(a && (b || c))`
  - Compiler evaluation of Boolean expressions understood?
- Control flow bugs
  - Loop termination
  - Program termination
  - Loops bypassed because of entry condition?
  - Off-by-one errors in iterations
  - Non-exhaustive decisions

Code Inspections

- Interface errors
  - Number and (evaluation-)order of parameters
  - Parameter values valid (pre-condition)
  - Error codes/exceptions handled
- I/O errors
  - Reading from file/stream in correct format
  - Buffer size matches record size
  - File/stream opened before used
  - End-of-file handled?
  - I/O errors handled?

- Other problems
  - Check compiler warnings
  - Input checked for validity/sanitized?



(http://xkcd.com/327/)

Running Test-Cases

Different levels of automation:

- Test suite generated manually (most common)
- Test suite generated with tool assistance
- Automated Test-Case Generation

Running Test-Cases

- Black-box testing
  no access to code, test-cases derived from specification
- White-box testing
  access to source code, test-cases from specification and code

Black-box Testing

- Equivalence Partitioning
  - Partition the *input domain* into equivalence classes
  - Program expected to behave similar on all inputs in a class
- Boundary Testing
  - Pick values from boundaries of equivalence classes
  - “on”, “above”, “beneath”

Black-box Testing

- Equivalence Partitioning
  - Partition the *input domain* into equivalence classes
  - Program expected to behave similar on all inputs in a class
- Boundary Testing
  - Pick values from boundaries of equivalence classes
  - “on”, “above”, “beneath”
- Usually applied in combination

Equivalence Partitioning

Two phases:

- Identify equivalence classes
  - From specification, function signature, pre-conditions
  - Split into groups of valid and invalid inputs/equivalence classes
- Define the test cases
  1. Assign unique identifier to each equivalence class
  2. Until all equivalence classes covered by test cases:
     - Write new test case covering covering as many valid equivalence classes as possible
     - Write new test case covering one and only one invalid equivalence class

Equivalence Partitioning

Two phases:

- Identify equivalence classes
  - From specification, function signature, pre-conditions
  - Split into groups of valid and invalid inputs/equivalence classes
- Define the test cases
  1. Assign unique identifier to each equivalence class
  2. Until all equivalence classes covered by test cases:
     - Write new test case covering covering as many valid equivalence classes as possible
     - Write new test case covering one and only one invalid equivalence class (Why?)

Example: Password Rules

- The password must be at least 8 characters long
- The password **must** contain at least:
  - one alphabetic character `[a-zA-Z]`
  - one numeric character `[0-9]`
  - one of the following special characters:
    `‘ ! @ $ % ^ & * - _ = + [ ] ; : ’ " , < . > / ?`
- The password **must not**:
  - contain spaces
  - begin with an exclamation or question mark (`!`, `?`)
  - contain your login ID
  - contain your registered email address
  - contain 3 or more repeating identical characters (e.g., `aaa`)
- Passwords are treated as case sensitive

Example: Equivalence classes for passwords

Condition	Valid	Invalid
$8 \leq \|\text{password}\|$	$8 \leq \|\text{password}\|$ (1)	$\|\text{password}\| < 8$ (2)
$\geq$ 1 of `[a-zA-Z]`	yes (3)	no (4)
$\geq$ 1 of `[0-9]`	yes (5)	no (6)
$\geq$ 1 special ch.	yes (7)	no (8)
no spaces	yes (9)	no (10)
not start with `!`,`?`	yes (11)	starts with `!` (12), starts with `?` (13)
not contain login	yes (14)	no (15)
not contain email	yes (16)	no (17)
no 3 rep. char.	yes (18)	no (19)

Example: Test cases for passwords

Test case	Result	Covers
`mrKl9?dn`	✓	1, 3, 5, 7, 9, 11, 14, 16, 18
`mrKl9?d`	✗	2
`124532!9`	✗	4
`duRkL!n’`	✗	6
`duRkL9n7`	✗	8
`Du k2!n’`	✗	10
`!uMk2Dn’`	✗	12
`?uVk2Dn’`	✗	13
`D3U`*user*`?`	✗	15
`D1U`*email*	✗	17
`RlZaaa?9`	✗	19

Example: Test cases for passwords

Test case	Result	Covers
mrKl9?dn	✓	1, 3, 5, 7, 9, 11, 14, 16, 18
mrKl9?d	✗	2
124532!9	✗	4
duRkL!n’	✗	6
duRkL9n7	✗	8
Du k2!n’	✗	10
!uMk2Dn’	✗	12
?uVk2Dn’	✗	13
D3U$user$?	✗	15
D1U$email$	✗	17
RlZaaa?9	✗	19

don’t use any of these passwords. . .

Example: Test cases for passwords

Test case	Result	Covers
`mrKl9?dn`	✓	1, 3, 5, 7, 9, 11, 14, 16, 18
`mrKl9?d`	✗	2
`124532!9`	✗	4
`duRkL!n'`	✗	6
`duRkL9n7`	✗	8
`Du k2!n'`	✗	10
`!uMk2Dn'`	✗	12
`?uVk2Dn'`	✗	13
`D3U`*user*`?`	✗	15
`D1U`*email*	✗	17
`RlZaaa?9`	✗	19

don't use any of these passwords... they are *mine*!

What is an “Equivalence Class”?

- In mathematics, equivalence classes are *disjoint*!
- So how can *one* test case cover *several* equivalence class?

What is an “Equivalence Class”?

- In mathematics, equivalence classes are *disjoint*!
- So how can *one* test case cover *several* equivalence class?

- “Equivalence Classes” partition program behavior
- Determined by *expected* behavior
- Different aspects of behavior result in different partions

Equivalence Classes and Program Behaviour

```
void foo (unsigned x)
{
  if (x%2)
    even (x)
  else
    odd (x);

  if (x>=50)
    larger (x);
  else
    smaller (x);
}
```

- Equivalence classes:
  1. x is odd
  2. x is even
  3. x is smaller than 50
  4. x is larger or equal 50

Equivalence Classes and Disjointness

	even					odd				
$50 \leq x$	90	92	94	96	98	91	93	95	97	99
	80	82	84	86	88	81	83	85	87	89
	70	72	74	76	78	71	73	75	77	79
	60	62	64	66	68	61	63	65	67	69
	50	52	54	56	58	51	53	55	57	59
$x < 50$	40	42	44	46	48	41	43	45	47	49
	30	32	34	36	38	31	33	35	37	39
	20	22	24	26	28	21	23	25	27	29
	10	12	14	16	18	11	13	15	17	19
	0	2	4	6	8	1	3	5	7	9

- even and odd don’t overlap
- $x < 50$ and $50 \leq x$ don’t overlap

Combining Equivalence Classes?

1. ($\mathtt{x}$ is even) $\wedge$ ($\mathtt{x} < 50$)
2. ($\mathtt{x}$ is odd) $\wedge$ ($\mathtt{x} < 50$)
3. ($\mathtt{x}$ is even) $\wedge$ ($\mathtt{x} \geq 50$)
4. ($\mathtt{x}$ is odd) $\wedge$ ($\mathtt{x} \geq 50$)

- Now all equivalence classes are disjoint
- But potential combinatorial explosion
- Approach: Decision table testing (not covered here)
  - cf. e.g. "Essentials of Software Testing",
    Bierig et al., Cambridge University Press 2022

Program Behavior Formalized

- $\rightarrow$ defines single steps of a program
- Terminating execution of program P:

$$\langle P, \sigma_0 \rangle \quad \longrightarrow^* \quad \langle \texttt{skip}, \sigma_n \rangle$$

Program Behavior Formalized

- $\rightarrow$ defines single steps of a program
- Terminating execution of program P:

$$\langle P, \sigma_0 \rangle \rightarrow \langle P', \sigma_1 \rangle \rightarrow \cdots \rightarrow \langle P^{n-1}, \sigma_{n-1} \rangle \rightarrow \langle \texttt{skip}, \sigma_n \rangle$$

Program Behavior Formalized

- $\rightarrow$ defines single steps of a program
- Terminating execution of program P:

$$\langle P, \sigma_0 \rangle \rightarrow \langle P', \sigma_1 \rangle \rightarrow \cdots \rightarrow \langle P^{n-1}, \sigma_{n-1} \rangle \rightarrow \langle \mathtt{skip}, \sigma_n \rangle$$

- Induces sequence of program states (a "trace" or "path"):

$$\pi \stackrel{\text{def}}{=} \sigma_0, \sigma_1, \ldots, \sigma_{n-1}, \sigma_n$$

(σ_0 is an initial state, σ_n is a final state)

Execution Traces and Properties

- "Properties" correspond to sets of traces:
  - Traces start in state in which $\texttt{x} < 50$:

$$\{\pi | \pi = \sigma_0, \ldots, \sigma_n \ \wedge \ \sigma_0(\texttt{x}) < 50\}$$

Execution Traces and Properties

- "Properties" correspond to sets of traces:
  - Traces start in state in which $\texttt{x} < 50$:

$$\{\pi | \pi = \sigma_0, \ldots, \sigma_n \ \wedge \ \sigma_0(\texttt{x}) < 50\}$$

  - Traces terminate in state in which $\texttt{x} \neq 0$:

$$\{\pi | \pi = \sigma_0, \ldots, \sigma_n \ \wedge \ \sigma_n(\texttt{x}) \neq 0\}$$

Execution Traces and Properties

- "Properties" correspond to sets of traces:
  - Traces start in state in which $\texttt{x} < 50$:
    $$\{\pi | \pi = \sigma_0, \ldots, \sigma_n \ \wedge \ \sigma_0(\texttt{x}) < 50\}$$
  - Traces terminate in state in which $\texttt{x} \neq 0$:
    $$\{\pi | \pi = \sigma_0, \ldots, \sigma_n \ \wedge \ \sigma_n(\texttt{x}) \neq 0\}$$
  - Traces that have more than k steps:
    $$\{\pi | \pi = \sigma_0, \ldots, \sigma_n \ \wedge \ n > k\}$$

Execution Traces and Properties

- "Properties" correspond to sets of traces:
  - Traces start in state in which $\texttt{x} < 50$:
    $$\{\pi | \pi = \sigma_0, \ldots, \sigma_n \ \wedge \ \sigma_0(\texttt{x}) < 50\}$$
  - Traces terminate in state in which $\texttt{x} \neq 0$:
    $$\{\pi | \pi = \sigma_0, \ldots, \sigma_n \ \wedge \ \sigma_n(\texttt{x}) \neq 0\}$$
  - Traces that have more than k steps:
    $$\{\pi | \pi = \sigma_0, \ldots, \sigma_n \ \wedge \ n > k\}$$
  - Traces that visit program location ℓ:
    $$\{\pi | \pi = \sigma_0, \ldots, \sigma_n \ \wedge \ \exists i \,.\, 0 \leq i \leq n \wedge \sigma_i(\texttt{pc}) = \ell\}$$

Execution Traces and Properties

- "Properties" correspond to sets of traces:
  - Traces start in state in which $\mathtt{x} < 50$:
    $$\{\pi | \pi = \sigma_0, \ldots, \sigma_n \ \wedge \ \sigma_0(\mathtt{x}) < 50\}$$
  - Traces terminate in state in which $\mathtt{x} \neq 0$:
    $$\{\pi | \pi = \sigma_0, \ldots, \sigma_n \ \wedge \ \sigma_n(\mathtt{x}) \neq 0\}$$
  - Traces that have more than k steps:
    $$\{\pi | \pi = \sigma_0, \ldots, \sigma_n \ \wedge \ n > k\}$$
  - Traces that visit program location ℓ:
    $$\{\pi | \pi = \sigma_0, \ldots, \sigma_n \ \wedge \ \exists i \,.\, 0 \leq i \leq n \wedge \sigma_i(\mathtt{pc}) = \ell\}$$
  - Traces in which predicate Q becomes true at least once:
    $$\{\pi | \pi = \sigma_0, \ldots, \sigma_n \ \wedge \ \exists i \,.\, 0 \leq i \leq n \wedge \sigma_i \models Q\}$$

Properties and Equivalence Partitioning

- We can define Equivalence Classes using Properties
- Note: More powerful than just partitioning input values
  - But in black-box testing observability is limited to inputs/outputs
  - We will encouter properties again in white-box testing

Boundary Testing

Differences to equivalence partitioning:

- Choose one *or more* elements close to boundaries of equivalence class
  - In equivalence partitioning, we pick a test case in “the middle”
- Also take *result* into account (output equivalence classes)

Guidelines:

- Choose end of range for valid inputs
- Just beyond the ends for invalid inputs
- Think about test cases causing output outside range
- For ordered sets (e.g., strings): focus on first and last elements

Boundary Testing

```
float sqrt (float x);
```

pre: $\mathtt{x} \geq 0$

post: $|\mathtt{result}^2 - \mathtt{x}| < \varepsilon$

- Domain: floating point (defined by IEEE 754 format)
  - comprises *sign s*, *coefficient c*, *exponent q*, *base b* $\in \{2, 10\}$

$$(-1)^s \cdot c \cdot b^q, \quad \text{e.g.,} \quad (-1)^1 \cdot 12345 \cdot 10^{-3} = -12.345$$

- Finite elements determined by *precision p* (# bits of exponent) and *emax*:

$$0 \leq c \leq b^p - 1 \qquad 1 - \text{emax} \leq q + p - 1 \leq \text{emax}$$

- Additional elements: ± 0, $\pm\infty$, `NaN` (quiet/signaling)

Boundary Testing

Valid equivalence classes:

- $[0, \infty)$

Invalid equivalence classes:

- $[-\infty, 0)$
- $+\infty$
- `NaN` (quiet/signaling)

Output equivalence classes:

- $[0, \infty)$ (or $(-\infty, \infty)$, depending on specification)
- `NaN`

Boundary Testing

```
float sqrt (float x);
```

pre: $\texttt{x} \geq 0$

post: $\texttt{result}^2 - \texttt{x} < \varepsilon$

Test cases from valid equivalence classes:

- $+0$, -0, `FLT_MAX`, `FLT_EPSILON` (see `float.h`), some $v \in [0, \infty)$

Test cases from invalid equivalence classes:

- -`FLT_MAX`, -`FLT_EPSILON`, some $v \in (-\infty, 0)$
- $-\infty$, $+\infty$
- `NaN` (quiet and signaling)

Test cases for output equivalence classes:

- Already covered

Boundary Testing

Writing test cases:

```
 /* positive test-case */
float x = FLT_MAX;
float result = sqrt (x);
assert (result * result - x < EPSILON);

/* negative test case */
float x = -42;
float result = sqrt (x);
assert (isnan(result));
```

- Also available: unit testing libraries (JUnit, CUnit, cppUnit…)
- Provide special functions (e.g., `CU_ASSERT`, `CU_FAIL`, `CU_PASS`) for reporting outcome

Boundary Testing: Password Example

Consider length:

- Test cases where $|\text{password}| \in \{0, 1, 8, 9\}$

Consider content:

- Password that contains *no* blanks
- Password with first, last, or all characters blanks
- Password with only first/last characters is numeric
- Password with only first/last characters is special
- Password with only first/last characters is alphabetic
- Password with no numeric/special/alphabetic characters
- ...

Testing a Balanced Binary Search Tree

- Derive test cases for the insertion function of a **balanced (AVL) binary search tree**.
- using the following techniques:
  a) Equivalence class partitioning
  b) Boundary value testing

Signature

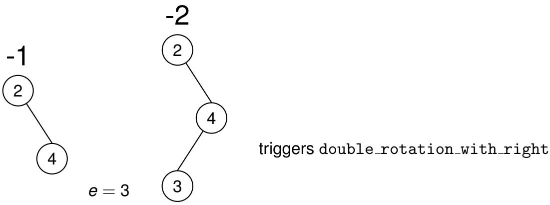
```
/* recursive tree structure          */
typedef struct _tree
{
  struct _tree * left;
  struct _tree * right;
  int element;
  int height;
} Tree;
```

`insert(int e, Tree *t)`: Insert element `e` into the tree `t`

Note:

- You don't know the concrete implementation
- But you know how an AVL is supposed to work:
  - $|\text{left height} - \text{right height}| \leq 1$

Inner Workings of AVL Trees

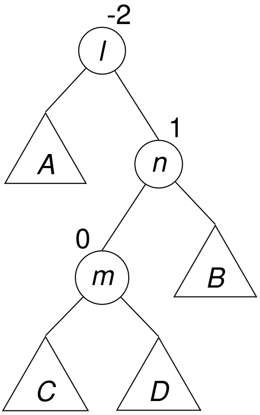


- after `single_rotation_with_left` 3 becomes child of 2
- after `single_rotation_with_right` 3 becomes root

Equivalence Classes for Inputs

Remember: Tree `t` is an input, too!

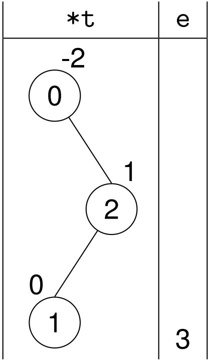
- Balanced: $|\text{left height} - \text{right height}| \leq 1$
- Elements in left sub-tree are smaller than elements in right sub-tree



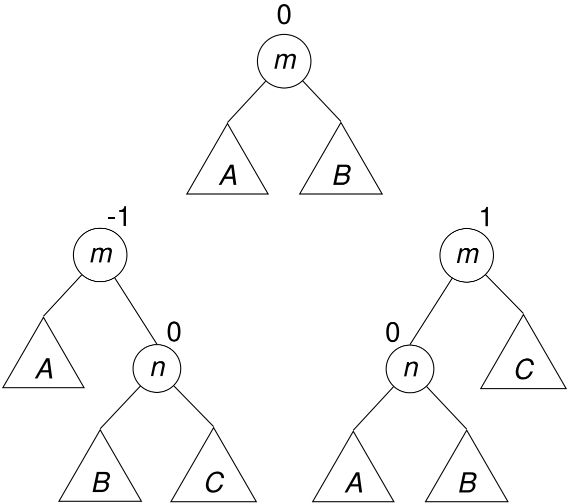
1. Derive equivalence classes:
   - based on balance
   - number of elements
   - content
   - …
2. Illustration of equivalence classes (see right).
3. Use table to list your equivalence classes

Boundary Value Testing

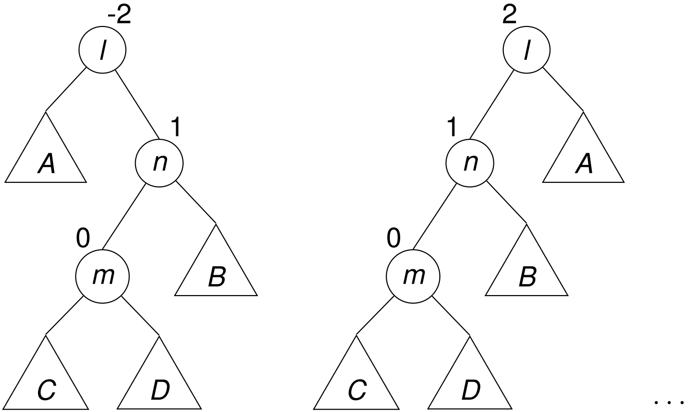
1. Derive test cases using boundary value testing:
   - cover all equivalence classes (valid, invalid)
   - take outputs into account
2. Illustration of test cases (see right)
3. Use table to list test cases



Balanced Trees



Unbalanced Trees



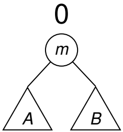
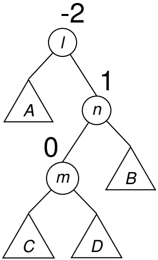
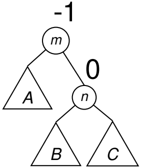
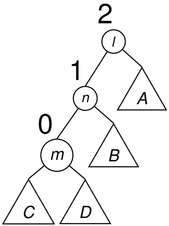
Derive valid and invalid equivalence classes for the function `insert`. Assign a unique number/id to each equivalence class.

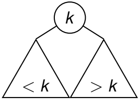
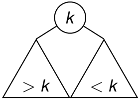
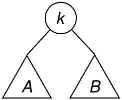
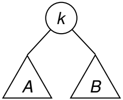
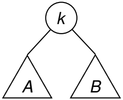
Condition	Valid	ID	Invalid	ID

Equivalence Partitioning

- **Invalid** denotes *invalid inputs*
  - e.g., condition: “Tree is balanced”, invalid: unbalanced tree
  - Not always simply answered with Yes/No!
- One condition can result in multiple equivalence classes
  - e.g., “Tree is balanced”
  - valid: possible height differences: -1, 0, 1
  - invalid: possible height differences: -2, 2
- Also consider *output* equivalence classes
  - Especially for trees, there many (different balance!)

Equivalence Partitioning

Condition	Valid	ID	Invalid	ID
balanced	 insert $e > m$	1		2
–"–	$e < m$ 	3		4
	. . .			

Condition	Valid	ID	Invalid	ID
ordered	0  $e > k$	5	0  $e > k$	6
no duplicates	0  $k \notin A \cup B$ $e > k$	7	0  $k \in A$ $e > k$	8
_”_			0  $k \in B$ $e < k$	9
	...			

Equivalence Partitioning

Numerous other cases you could consider:

- Try to trigger rotations
  - e smaller than elements in left subtree A
  - e larger than elements in right subtree A
  - ...
- Try to insert elements already contained
  - $e \in A$, $e \in B$
  - Warning! These insertions are *not* invalid!
- Could also consider `null` as separate equivalence class
  - Warning! Insertion into empty tree *not* invalid!
- ...

Use *Boundary Value Testing* to derive a test-suite for the method `insert`. Indicate which equivalence classes each test-case covers by referring to the numbers from before.

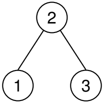
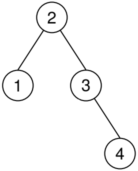
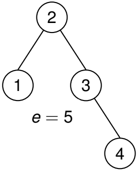
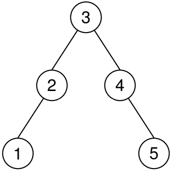
Input	Output	Classes Covered

Hint: in exam no points for *redundant* and *non-boundary* test cases

Boundary Value Testing

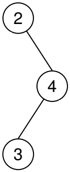
- “Boundaries” a bit unclear here, requires creativity
  - empty tree (`null`), tree with one element
  - “full” tree (all leaves filled)
  - two elements, leaning left/right
  - …

Boundary Value Testing

Input	Output	Classes Covered
0  $e = 4$	-1 	1,5,7
-1  $e = 5$	0 	. . .

Boundary Value Testing

Cover invalid classes **individually**!

Input	**Output**	**Classes Covered**
-2  $e = 5$	exception	**2**

Equivalence Testing/Boundary Testing

Important:

- Specify **expected result** for test cases
- Test cases need to specify *concrete values*, also for output
- Which equivalence classes are covered? (enumerate them!)
  - Cover as many valid classes as possible with few test cases
  - Cover each invalid class with a *separate* test case
- Also cover output equivalence classes
  - Especially for trees, there many (different balance!)

Black-box Testing: Random Testing

Randomly choose inputs

- Generally considered as inferior
- May be hard to generate *valid* inputs
  - probability of “guessing” 3 equal sides of isosceles triangle!
- May miss many relevant behaviours
  - E.g., if code contains if `(x==y)`
- Known to find “simple” bugs quickly, though

Black-box Testing: Random Testing

- Can be combined with equivalence partitioning
  - Pick element from each equivalence class at random

Limitations of Black-box Testing

- Can easily miss relevant inputs
- Are all program behaviours explored?

Limitations of Black-box Testing

- Can easily miss relevant inputs
- Are all program behaviours explored?
- Program behaviour induces more *equivalence classes*
  - e.g., “inputs resulting in same control flow”
  - requires access to source code!

Summary

- Verification is difficult, never ultimate
- Instead: *falsification*/testing
- Black-box testing
  - Equivalence partitioning
  - Boundary testing

Next lecture:

White box testing/Coverage metrics

Automated test case generation