Code Quality - Testing
Week 6 Tuesday

COMP1100/1130

Testing and Verification

How do you know whether your code is correct?

- ❑ Looking at your code.
- ❑ Testing your code.
- ❑ Formal verification (mathematical proofs).

Testing can only show the presence of bugs, not the absence (Dijskstra).

Testing and Verification

Is it really that important to test your code? Can we instead just wait until a user finds something wrong and fix it then?

Example: The Ariane 5 rocket - it exploded due to a software bug.

Ben-Ari, M. The Bug That Destroyed a Rocket, http://docenti.ing.unipi.it/~a009435/issw/extra/ariane5-benari.pdf

- Also think about everything that software controls: financial systems, defence systems, vehicles, medical equipment, etc.!

Black Box Testing vs. White Box Testing

Black box testing

- Based on the specification.
- Doesn’t use the code (can even be designed before you write the code).

White box testing

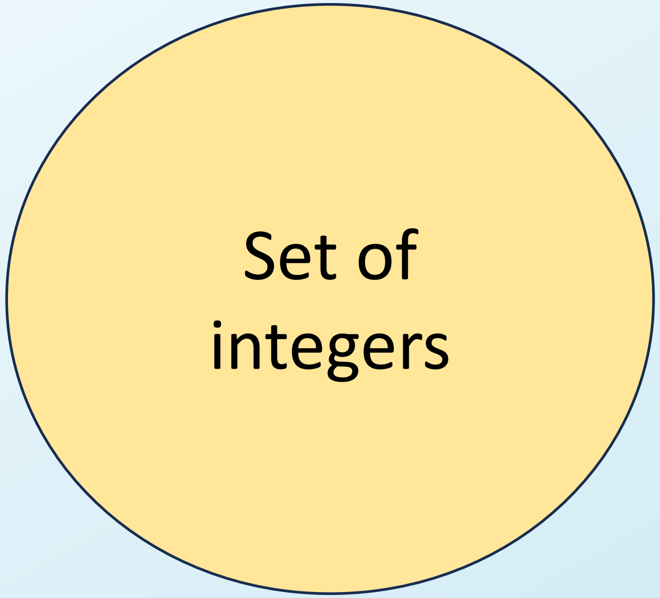
- Based on the code.
- Find ways to try and break the code, e.g. border cases.

Black Box Testing

Black box testing

Black Box Testing

What should we test?



Zero

Positive value

Negative value

Max/min values of Int

Black Box Testing

Identify groups (equivalence classes)

- No need to check several elements in the same group.
  - Check one element from each group.
- Inputs in the same group should behave in a similar way.
- Inputs in different groups should behave differently.
- Groups should collectively cover all possibilities.
- Pay attention to special cases, e.g. boundaries, zero.

Black Box Testing

Equivalence classes:

- If an input can be from a given range of values, e.g. {1… 5}, identify one valid equivalence class (correct input), e.g. 2, and two invalid classes, e.g. 0 and 6.

- For inputs from a set of values, where each value is handled differently, e.g. a set of choices on a screen (Delete, Create, Copy, etc.) then make one class for each of them, and an invalid input case.

Black Box Testing

- Strings: test a string with one char, many chars, empty string.
- Lists: test a list with one element, many elements, empty list.
- {1..10}: test 1, test 10, test a number in the middle. (Also might be useful to test invalid inputs if they are possible).

Black Box Testing

```
maxThree :: Int -> Int -> Int -> Int
```

- The group where the first number is the greatest.
- The group where the second number is the greatest.
- The group where the third number is the greatest.
- Boundary cases: some inputs are equal.
- Also include 0 and negative inputs.

Black Box Testing

```
length :: [a] -> Int
```

What should we test?

- ❑ An empty list.
- ❑ A list with one element.
- ❑ A list with two or more elements.
- ❑ Also: A list with duplicate elements. - This might catch errors where the code is ignoring duplicates.

White Box Testing

White box testing

- Based on the code.
- Identify points where the code makes a choice, e.g. cases, guards, base case vs. step case in recursions.
- Watch out for otherwise and _
- Focus on inputs at boundaries and overlapping situations.

White Box Testing

```
maxThree :: Int -> Int -> Int -> Int
maxThree x y z
  | x > y && x > z = x
  | y > x && y > z = y
  | otherwise      = z
```

- ❑ Make test cases to cover each of the choices.

- ➢ To make a test case to reach the 3rd choice: based on the boundary of the 1st two cases, test e.g. 2 2 1

White Box Testing

Branch Coverage:

- The test cases should cover each of the branches at least once.
  - Both the *true* and *false* cases of each branch should be covered.

```
maxThree :: Int -> Int -> Int -> Int
maxThree x y z
  | x > y && x > z = x
  | y > x && y > z = y
  | otherwise      = z
```

White Box Testing

Branch Coverage:

- The test cases should cover each of the branches at least once.
  - Both the *true* and *false* cases of each branch should be covered.

```
maxThree :: Int -> Int -> Int -> Int
maxThree x y z
  | x > y && x > z = x
  | y > x && y > z = y
  | z > x && z > y = z
```

Here, it is possible for all three cases to be false.

White Box Testing

Branch Coverage:

- The test cases should cover each of the branches at least once.

The 1st choice is covered by using an empty list; the 2nd choice with any other list. Also test a list with one element vs. a list with a few elements (where the recursion would loop several times).

```
mysteryFunc :: [Int] -> Int
mysteryFunc list = case list of
  []    -> 0
  _:xs -> 5 + mysteryFunc xs
```

Testing

Remember that tests can't cover all the possibilities.

The goal is to find a set of test cases that will be the most likely to find bugs – that's why we look at things like boundary cases.

The aim is also to find test cases that cover the largest range of possibilities, e.g. by using equivalence classes.

Documenting Tests

How do you run the tests? Just typing inputs into ghci?

- This can be a way to start testing, but think about large or complex programs.

    - It would be difficult to remember what has already been tested.

    - Also, what if your code changes? The tests have to be created again.

➢ Documenting your tests is important.

Doctests

Recall Doctests from Lab 3:

```
-- | Compute Fibonacci numbers
-- >>> fib 10
-- 55
-- >>> fib 5
-- 5

fib :: Int -> Int
```

Doctests

The correct format is required:

Format: The | is essential:

```
-- | Compute Fibonacci numbers ✓
-- Compute Fibonacci numbers    ✗
```

Format: Indenting and other spacing have to be perfect:

```
-- 5    ✓
--  5   ✗
```

Call `doctest MyFileName.hs` from **outside** ghci.

Other Types of Testing

Randomised testing

- allows you to run many tests with minimum effort
- could miss special cases

Property-based testing

- Haskell's QuickCheck library see: online or the textbook.
- We won't cover this in this course.

References

Some references if you're interested in further reading (but these go far beyond what we're learning here):

- The Art of Software Testing, by Myers, Sandler, Badgett & Thomas, Wiley, 2004.
- Software Testing: A Craftsman's Approach, by Jorgensen, Auerbach Publishers, 2013.