Testing

Ravishankar Rajagopalan

What’s the big deal on testing?

- Some companies have a culture of taking testing very lightly
- A systematic attempt to break a program
- Famous quote by Dijkstra
  - Testing can demonstrate the presence of bug not absence
- Thinking about problems/test situations as you code
- Writing code which can generate code
- Program is expected to work correctly under a broad set of conditions

When

- Design the tests beforehand
  - Automate as much as possible
- Test as you write the code
  - Incremental (TDD)
  - can ensure bugs can be detected before they get introduced
  - easier change management when combined with automation
- After you are done writing the code

How

- Testing boundary condition
    - Reads characters and fills in a buffer till newline or buffer is filled

```
char str[MAX];
int j;

for (j = 0; (str[j] = getchar()) != '\n' && j < MAX-1; ++j);
str[--j] = '\0';
```

How

- Testing boundary condition

```
for (j = 0; j < MAX-1; j++)
    if ((str[j] = getchar()) == '\n')
     break;
str[j] = '\0';
```

How

- Testing boundary condition

```
for(j = 0; j < MAX - 1; j++)
    if((str[j] = getchar()) == '\n' || str[j] == EOF)
     break;
str[j] = '\0';

//Will look into it in the another section
```

How

- Input and output validation

```
double average(int num, double arr[])
{
    int i;
    double total = 0.0;

    for (i = 0; i < num; i++) {
     total += arr[i];
    }
    return total/num;
}
```

How

- Assertions and recovery, defensive programming
    - `assert(n <= MAX_ITEMS);`
    - `assert(n > 0);`
    - `if (finalMarks > 100 || finalMarks < 0)...`

How

- Expecting right return values including possible errors
  - Identify set of valid return values
  - Identify possible errors the function can return
    - Map correctly what error leads to what return values
    - Generic errors always don’t work (e.g. always returning a -1 or a standard exception), caller needs to get a correct idea of what may have happened

How

- Test output
  - Should show up on error pinpointing the problem area/code
  - Desirable to have parameters to display outputs
  - Mapping expected inputs & outputs
    - make an exhaustive set for different set of conditions
    - start building a test scaffold as you code, if needed
- Measure coverage
  - Use tools, profilers
  - Tests need to be designed such that all code lines are covered.
    - Lines not covered are not important? Why is the code there? - Reviewers role!
- Ignoring ‘unexpected’ errors?
  - Many programs can get overlooked even while reviewing. The logic for the kind of errors which it may have to handle may be far from sufficient to handle field conditions
  - Some errors safely assumed to never happen do occur in real life!

How

- Efforts of test driven development or testing as we code is minimal compared to doing all testing - all at once `some day` which for **many** *never comes*
- If we wait until someone breaks the program, that'll take time and one would have possibly forgotten the code/not remember it well enough
    - Working under pressure it could be harder to remember and get through the code/logic
        - Result : Possibly fragile fixes
    - On the other hand reviewing the program after a short break can bring fresh perspectives which may not have been obvious when one is continuously involved with it

How summary till now!

- Testing boundary condition
- Input and output validation
- Assertions and recovery
- Handling so called “can never happen situations”
  - Never trust what a caller may pass you
- Measure code coverage
- Expecting right output, compare with known values
  - Pair functions testing
    - Encode and decode
    - Comparing with known results
  - Conservation properties
- Returning results, error and more..

How

- Automate tests
  - Regression testing - making sure regression is currently valid, else this can be a disaster as one can be tricked into thinking all is well.
- Self contained test
  - Program contains (or can be fed) output also and then check with produced output
- Tests evolve as the code does
- Stress testing
  - High volume inputs, concurrency testing, garbage / malicious inputs, random inputs
    - Test overflow and underflow cases, error handling
- Remove/disable tests from production code
  - Enabling/disabling test/debug on production. Have a sanity test for checking

How

- Test on all combinations
  - Test on all combinations of machines, platforms and browsers etc. as applicable
    - Dependencies on byte order, handling of null pointers, handling carriage return and newline in libraries etc. may be different and **do** introduce bugs
- Implementing new features or testing existing ones while there are still known bugs?
- Get new users to test your program
- For interactive programs
  - see if you can capture user behaviour
  - use scripts which can replay user behaviour
- Why automate as much as possible
  - “Machines don’t make mistakes or get tired or fool themselves into thinking that something is working when it is not”

Summary

- Well designed code, fewer bugs
- Wary of unexpected conditions
- Test as you develop
  - Keep in mind boundary conditions, garbage/unexpected inputs
- Automation & Regression
  - Design code in such a way that it's easy to automate testing
- The most important thing in testing is not to skip it!
  - The client will find the bug if you don't ! And the cost ~~could be~~ is pretty high !