Michael Shintaku, shintaku
Matthew Staehely, mstaehel

Motivation

- Key problem:
  - Regression testing can be difficult to prioritize
  - Full regression testing takes too much time in practice
    - Most of the time you are just interested in a subset of the tests
  - Complex regression testing techniques can be unnecessary or too much “red tape” for their scope (for the day-to-day programmer and their modules)
    - Mandated large test suites can be made more efficient and relevant
- How everyday programmers deal with this today, and limitations of their approaches:
  - Imposed/mandatory test suites
    - Focuses the attention on where it matters
- Related thoughts:
  - Is probability based prioritization more effective than other methods

Approach

- High-level approach:
  - Build the tool/write and run the tests
  - Categorize based on probability: simple pass/fail over the initial tests
  - Select category/subset of tests, continually recalculate probabilities
  - Allow for running x number of tests < y time
- Why this addresses the key problem (why we will succeed):
  - It’s easy to use
  - No-nonsense simplistic approach to help productivity on large or small codebases
    - Other techniques may be more effective for truly massive codebases
- Key difference between this and other approaches:
  - Fast success on small portions of codebases prioritized over that on large-full codebases
- Limitations:
  - Scaling:
    - Large codebases take extremely long to test fully (how to get significant runs to calc. probability)
- Architecture:
  - jUnit or TestNG?

Distinctions

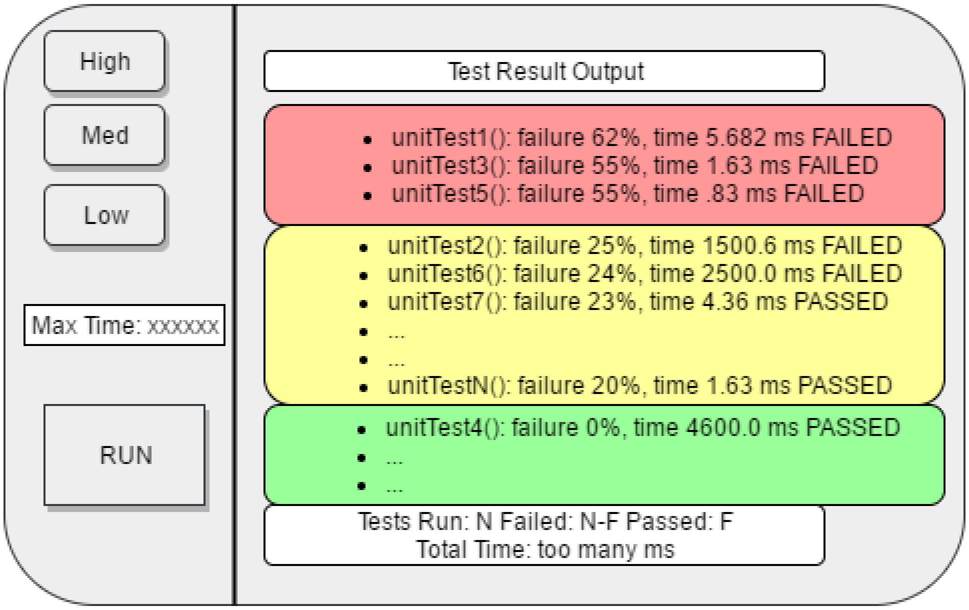
- Distinctions between this and other tools:
  - Focus on known, recurring faults
    - Not estimating probability of tests revealing unknown faults
  - Strictly probabilistic: not prioritizing based on code coverage
  - Cost/optimized resource: only time

- Clarifications:
  - Productivity:
    - Organization of test ~= better time management
      - If highly occurring failed tests are simple to fix
    - MIGHT help focus productivity on core functionality than integrative parts
      - This can also be done through better design/organization on the user
  - Scope:
    - May not scale well, as it requires multiple runs of entire test suite to gather probabilities
    - Best for small-medium codebases ~lean teams or organizations
      - Helps to quickly solve recurring failed tests -- timecrunch

<u>Challenges/Risks</u>

- Implementing it correctly in our selected architecture may be more challenging than anticipated. Good research and code discipline will mitigate this.
- As with any project, proper management and teamwork is required

<u>Basic UI Mock-up</u>:



- unitTest1(): failure 62%, time 5.682 ms FAILED
- unitTest3(): failure 55%, time 1.63 ms FAILED
- unitTest5(): failure 55%, time .83 ms FAILED

- unitTest2(): failure 25%, time 1500.6 ms FAILED
- unitTest6(): failure 24%, time 2500.0 ms FAILED
- unitTest7(): failure 23%, time 4.36 ms PASSED
- ...
- ...
- unitTestN(): failure 20%, time 1.63 ms PASSED

- unitTest4(): failure 0%, time 4600.0 ms PASSED
- ...
- ...

Tests Run: N Failed: N-F Passed: F
Total Time: too many ms