Agenda

• What is testing debt?
• Spotting and measuring testing debt
• Fixing testing debt
  • Avoidance
  • Controlling scope
  • Reducing test times
  • Legacy tests and removing tests
• Avoiding technical debt
• Learning from best practice
• Some key takeaways
Understanding Debt

Definition – Technical Debt
• The implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer

Why is it important?
• Debt decreases productivity
• Unable to run in a sustainable fashion
• Unpredictable schedules
• Reduced software quality

Testing Debt
• Technical debt related specifically to testing
Evaluating testing debt

- Poorly written tests contribute to testing debt
- Ways to “smell” testing debt
  - Is it possible to select tests for a particular feature?
  - Is it possible to measure what a test does?
  - Is it possible to assess whether a test is still effective or is now outdated?
  - What happens when a test fail is not understood?
  - What is the run history of the test?
  - Is it possible to identify which parts of the code are no long effectively tested?
Measuring testing debt #1: Metrics

Metrics Rules

• Ensuring our metrics are effective

Code metrics

Test metrics
Measuring testing debt #2: Metrics

- Process Metrics
- Progress Metrics
- Effectiveness Metrics
- Efficiency Metrics
Fixing testing debt #1: Avoidance

• First – stop adding to technical debt

• Throw away tests that are no longer effective

• Reduce test suites that take too long

• Reduce testing debt with new tests
Fixing testing debt #2: Controlling test scope

The single biggest reduction in testing comes from reducing combinations of:

• releases
• configurations
• target platforms that need to be tested

There are a number of ways of approaching this problem
Fixing testing debt #3: Reducing total test times

1. Any testing, except release testing, should be time boxed
2. Reducing the number of software versions, configurations and targets
3. Improved test selection
4. Split up big bucket test sets
5. Replacing legacy tests that are ineffective or inefficient.
6. Removing testing bottlenecks
7. Minimise manual testing
# Testing Stages

<table>
<thead>
<tr>
<th>PHASE</th>
<th>FREQUENCY</th>
<th>SCOPE</th>
<th>TEST SCOPE</th>
<th>COMPLETE</th>
<th>AUTOMATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit</td>
<td>Continuous</td>
<td>Unit</td>
<td>Optional testing of unit development code</td>
<td>Optional</td>
<td>Optional</td>
</tr>
<tr>
<td>Unit pre-release</td>
<td>Pre-release</td>
<td>Unit</td>
<td>Functional testing of unit release candidate, static analysis and code review</td>
<td>Time-boxed (&lt; 10 mins.)</td>
<td>YES</td>
</tr>
<tr>
<td>Unit and Integration</td>
<td>Nightly</td>
<td>Code base</td>
<td>Code base functional testing of release candidate(s)</td>
<td>Time-boxed (&lt; 12 hours)</td>
<td>YES</td>
</tr>
<tr>
<td>Stress</td>
<td>Weekends</td>
<td>Code base</td>
<td>Code base fuzz testing on trunk</td>
<td>Time-boxed (&lt; 60 hours)</td>
<td>YES</td>
</tr>
<tr>
<td>Exploratory</td>
<td>Not repeated</td>
<td>Code base</td>
<td>Product non-functional testing on trunk eg: perf., usability, ...</td>
<td>N/A</td>
<td>Optional</td>
</tr>
<tr>
<td>Product Release</td>
<td>Pre-release</td>
<td>Product</td>
<td>Product acceptance testing of release candidate</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
Fixing testing debt #4: Removing tests

01
Effective test reduction requires
• clear policies
• good information that allow teams to make these hard decisions

02
Test suite reduction requires the collection of metrics on individual tests

03
This data can normally captured in the test management tool
Fixing testing debt #5: Legacy Test Issues

01. Over time tests are added but rarely removed.
02. Understanding of the old tests is normally poor.
03. The software is typically required to support an increasing number of configurations and additional target platforms.
04. More releases of the software are created.
05. Tests may require access to scarce resources that create a bottleneck.
Avoiding testing debt: A strategic approach
Testing Debt: Best practise research

**Based on paper-based research on**
- Google
- Spotify
- Facebook
- Microsoft
- Ericsson
- Unity
- Panda Strike

**And consultancy at a number of companies**
- 3rd party independent consultancy to review and reduce technical debt.
The “Best Practise”: organisation

Agile
- process, organisation, culture
- trust the developers
- hold them accountable

Scalable
- process, organisation

Strong
- including tools and continuous improvement

investment in infrastructure
The “Best Practise”: process

- Clarity
- Avoid formal handoff and synchronised releases
- Automate all repetitive testing
- Include additional types of product testing
- Prevent an accumulation of 'testing debt'
- Metrics
- Release management
- Single code base with all changes subject to code review
What (I hope) we have covered

• What is testing debt?
• Spotting and measuring testing debt
• Fixing Testing debt
  • Avoidance
  • Controlling scope
  • Reducing test times
  • Legacy tests and removing tests
• Taking a strategic approach “Avoiding technical debt”
• Learning from Best practice
Some things to try tomorrow

• Time box your testing
• Reduce the scope of your testing
• Are your developers doing enough testing?
• Identify your worst test bottleneck
• Identify a legacy test to remove

mike@testandverification.com  +447796307958
www.testandverification.com