The Path Not Taken: Maximizing the ROI of Increased Decision Coverage

Laura Bright
McAfee
Introduction

• Decision coverage is popular metric for many teams
  – High coverage as an indicator of product quality
  – Many QA teams set a min-ship target
• But what does high coverage really mean in terms of quality?
  – Clearly it is good to cover as many decisions as reasonably possible
  – Some new tests have higher ROI than others
• If your numbers are above target, does that mean you have high quality?
• If your numbers are below target, is that a cause for concern?
Goals

• Reframe the question:
  – Instead of asking “What should we do to improve our code coverage numbers?”, ask “What areas of the code need increased coverage to improve the quality of our software?”

• Goal is not just to increase coverage, but to cover overlooked test cases and other reasonable test scenarios

• Some uncovered decisions have a low ROI

• Focus on overall quality rather than reaching a target number
Benefits

• Identifying automated functional test cases that were overlooked by QA
• Removing dead or obsolete code
• Finding product defects that cause some code to not get executed
• Identifying manual test cases to cover parts of code that are difficult to test through automation
• Determining areas of the code that would benefit from increased unit testing
• Identifying refactoring opportunities
Presentation Overview

- Background and Terminology
- Project Overview
- Examples
- Results
Background and Terminology

- **Line coverage** – percentage of lines of code executed
- **Function coverage** – percentage of functions executed.
- **Decision coverage (a.k.a. Branch coverage)** – percentage of all branches executed in conditional statements (for example, for a single “if” statement, have tests been executed where the entire statement evaluates to both true and false)
- **Condition coverage** – percentage of conditions that have evaluated to both true and false. This is different from decision coverage because a single “if” statement may consist of multiple conditions e.g. if (a > b || c > d)
- **Condition/decision coverage** – This is the union of the condition coverage and decision coverage. It has the advantage of simplicity without the individual limitations of each of these two metrics [Bullseye].
- Our team’s focus was **condition/decision coverage**
Previous Work

• Marick (PNSQC 1991)
  • Case study of achieving different coverage goals through unit testing
  • Better to achieve high coverage through black-box tests and fill in gaps with unit tests
• Marick 1999
  • Coverage goal should be a guide for improvement, not a min-ship requirement
• Manu et al. (PNSQC 2010)
  • Case study on increasing block coverage from 91% to 100%
Project Overview

• McAfee Endpoint Security
  – Multiple components to protect systems for small and medium businesses
  – Mix of legacy and new code
• Multiple scrum teams distributed across sites in US and India
• C++ code uses Bullseye code coverage tool to measure coverage
• All teams report condition/decision coverage at end of each sprint
Efforts to improve coverage

• Team meetings to review coverage data and identify areas that require additional testing
  – Entire team (all Dev and QA) attends
  – Aim for one meeting per sprint
    • One component per meeting
    • Limit meeting time to one hour
• For uncovered conditions/decisions - moderator identifies action required and assigns follow up tasks, for example:
  – Determine if function X is still called anywhere in the code
  – Write test cases and automation scripts to cover condition Y
  – Remove a block of obsolete code
Examples
Uncovered functions

• Low hanging fruit
• Most fell into the following categories:
  – **Additional test cases needed** – Typically these were overlooked test cases that covered less commonly used product functionality
  – **Wrapper functions** – multiple APIs for the same functionality. Most are low priority.
  – **Dead code** – obsolete API’s or functionality removed from product
  – **Functions that were expected to be covered** – required additional investigation, may indicate a product defect
Dead code

- Code blocks that were unreachable in practice
- Some could technically be covered by unit tests, but never would be covered by end user
- Features not scoped for current release
  - Functionality permanently dropped from product
  - “Placeholders” for functionality scoped for a future release
- Removing or commenting out this code can improve metrics and allow us to focus on testing supported functionality
Boundary tests

• Values that are outside of the acceptable range
• Usually straightforward to cover through black box or unit tests.

```java
if (x >= MIN_VALUE && x <= MAX_VALUE) {
    // do something
} else {
    // handle the invalid value
}
```
Error Handling

- Error and null pointer checks are common in many projects
- Good practice to check return values
- Static analysis tools report unchecked values as defects
- Some of these are legitimate error conditions that need to be covered (Example 2)
- Others should never happen unless there is a bug in the code (Example 1)
- If the error should never happen, adding a test to explicitly cover it probably has a low ROI

Example 1

```c
void importantFunction() {
    int errorCode = DoSomethingImportant();

    if (errorCode == SUCCESS) {
        // <continue execution here>
    } else {
        // this should not happen
        Log_Debug("DoSomethingImportant() failed");
        return;
    }
}
```

Example 2

```c
void anotherFunction() {
    int errorCode = DoSomethingElse();

    if (errorCode == SUCCESS) {
        // <continue execution here>
    } else {
        Log_Event("a serious error occurred");
        ErrorRecoveryFunction();
    }
}
```
Null Pointer Checks

• Good practice to check that a pointer is not null before dereferencing it
• Memory allocation will normally succeed (testing with low memory is a valid test scenario, but low ROI to explicitly testing all of these conditions)
• Cleanup functions should free memory only if it exists, but explicitly testing both cases is low priority
• Explicitly testing the “false” condition for all of these conditions everywhere in the code is a low priority

```c
int * integer_array;
integer_array = new int[MAX_ARRAY_SIZE];
if (integer_array) {
    // <continue execution here>
}

void cleanUp (Object * obj) {
    if (obj) {
        delete obj;
    }
}
```
Results

- Our reviews indicated that most uncovered conditions/decisions were error and null pointer checks with low ROI
- We identified dead functions/code and new automated/manual test cases
- Much of this code was legacy code – difficult to unit test
- Increased overall coverage for the components to >50%

<table>
<thead>
<tr>
<th></th>
<th>Percent condition/decision coverage before review</th>
<th>Percent condition/decision coverage after review</th>
<th>Test cases added</th>
<th>Unreachable conditions/decisions removed</th>
<th>Defects filed</th>
<th>Additional conditions/decisions covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component 1</td>
<td>46</td>
<td>53</td>
<td>25</td>
<td>121</td>
<td>2</td>
<td>114</td>
</tr>
<tr>
<td>Component 2</td>
<td>48</td>
<td>50</td>
<td>20</td>
<td>~20</td>
<td>2</td>
<td>~25</td>
</tr>
</tbody>
</table>
Conclusions

• Need to consider how high coverage translates to high quality
• Your mileage may vary
  – These examples may not apply to your code base
  – But review process can help you maximize ROI of increased testing
• Consider the following:
  – Is this uncovered condition a valid test case?
  – Is it straightforward to cover this condition by a manual test, or by an automated functional or unit test?
  – Is there a good chance this condition will be covered in practice by an end user?
  – Is it a high risk if we don’t test this condition?
• If you answer “Yes” to these questions, there is a high ROI to adding test cases for the condition