Baseline Testing

Jim Snyder
Jimmy Wan

Why Do We Test?

- Measure Software Quality
  - How good is our code right now?
  - Tests do not ensure quality
  - Design ensures quality; it is built into the product.
- Measure Progress
  - Are we converging on a stable solution?
  - Did we loose functionality, quality, or correctness?

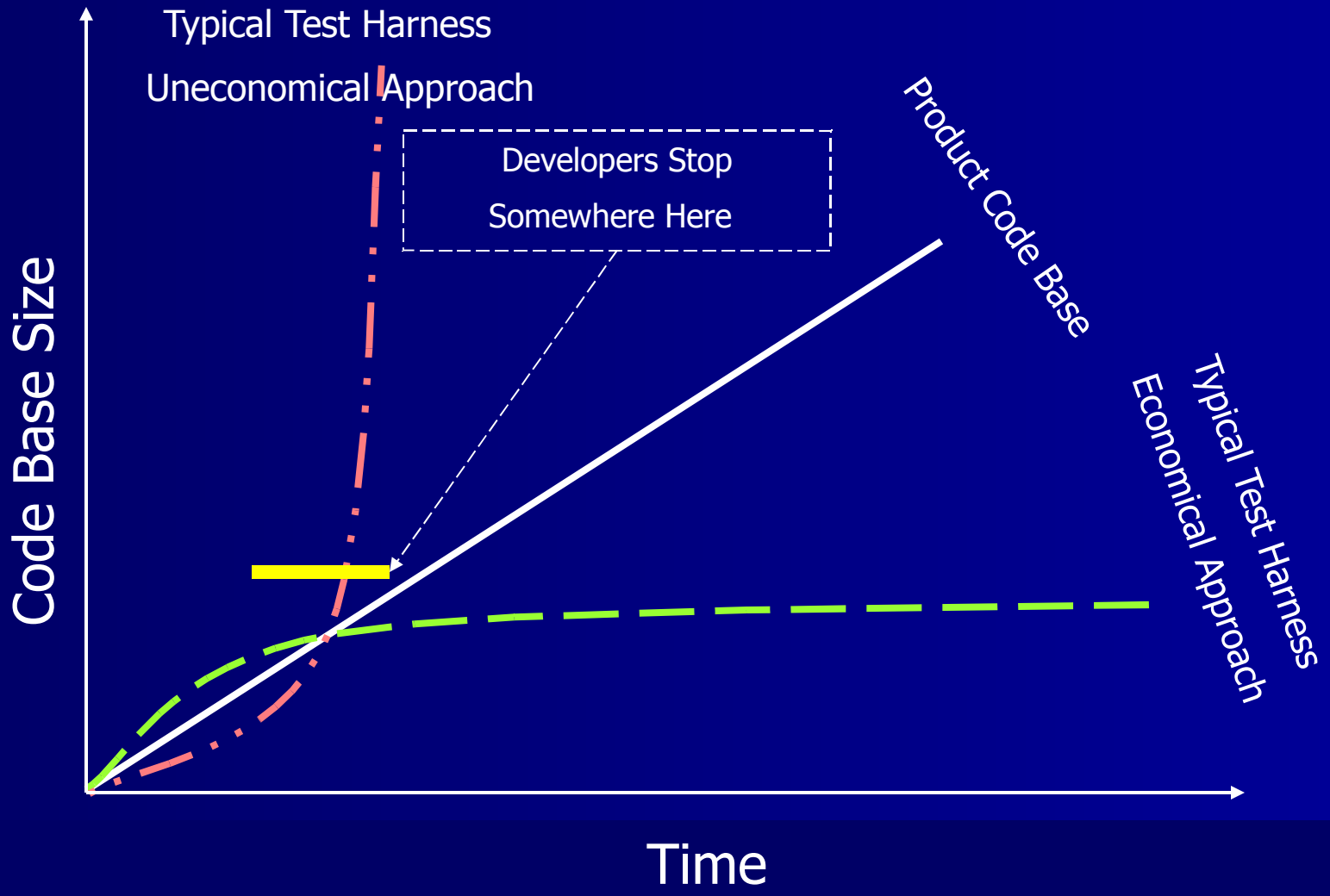
What Kind Of Tests Do We Need?

- Unit Tests (i.e. Abstract Data Types)
  - Pre-conditions, Post-conditions, Invariants
- Component Tests
  - Cluster Classes integrated into small functional units
- Regression Testing
  - Artifact testing tied to issue tracking
- Integration Tests
  - Assembled components in a contextualized environment
- Layer Tests
  - Leverage the principal of Systematic Isolation
  - If testing is difficult, revisit software architecture
- Performance Tests
  - Measure elements individually as appropriate
  - Measure integrated elements against functional scenarios

Test Plans

- A design problem by itself
  - How can I test an artifact well for a good price?
- An organizational structure to ensure systematic validation
  - Requires an understanding of Software Architecture
  - Requires synchronization with a Project Plan

The Testing Paradox: How to test the test?



Key Principals To Reduce Complexity

- Separation of Execution and Validation
  - Did my test execute (i.e. invoke functionality) as expected?
  - Did I get the answer I expect? Can I write down the answer before I code?
  - People **and** machines can understand visual difference; we should exploit this overlap.
- Build Lego Style Testing Elements
  - Allow Test Elements share state
  - Tests can measure deltas from a known state
- Leverage Systematic Isolation
  - Lego blocks + ordered execution + baseline = suite

Baseline Test Framework

- Test plan implementation toolkit without dictating test plan structure
- Not a test plan substitute
  - Frameworks don't eliminate thinking or planning
  - Designed to realize the previously stated testing principals
- Build System Agnostic
  - Maven, Ant, IDEs, command line
- Should Augment other Software Metric Collection
  - Code Coverage (e.g. the 80%/80% rule)
  - Profiling (e.g. performance tests)

Baseline Test Framework: Software Elements

- Framework Elements
  - Test Context
  - Test Suite
  - Test Case
  - Test Case Step
- Runtime Environment
  - Sharable Test State
  - Explicit Points of Variation
    - input inheritance hierarchy from least to most specific.

Baseline Testing

Putting it all together ...

File System Layout

- The contexts
  - Context descriptor
  - Input data
- The suites
  - Suite descriptor
  - Expected baseline
- The results
  - Testrun log
  - Actual to compare
  - Transient files

```
src/
   test/
      contexts/
         simple/
            context.xml
            inputdata.txt
      suites/
         suite_001_smoketest/
            suite.xml
            baseline.simple.out
      java/
         net/
            sf/
               sample/
                  SmokeTest.java
target/
   test/
      results/
         suite_001_smoketest/
            baseline.simple.out
            testrun.simple.out
```

Define a Test Context

- XML with two parts
  - Java System Properties
  - Context Arguments
    - Inherited by all suites
- At Runtime
  - Files relative to the context are readable.
  - Each suite instance is bound to a context.

```
<?xml version='1.0' encoding='ISO-8859-1'?>

<context>
    <environment>
        <arg name="environmentArg1">
        Environment Variable 1
        </arg>
        <arg name="environmentArg2">-22222222</arg>
    </environment>
    <!-- Supported arg types
    String
    Boolean
    Short
    Integer
    Long
    -->
    <arguments>
        <!-- context-level arguments. -->
        <arg name="arg1">contextArg1</arg>
        <arg name="arg2" type="String">contextArg2</arg>
        <arg name="arg3">contextArg3</arg>
        <arg name="arg5">contextArg5</arg>
    </arguments>
</context>
```

Define a Test Suite

- XML with args and test cases in order
  - Must define the comparator.
  - Must define applicable contexts.
  - Inherited args can be overridden.
- Suite Life-cycle
  - init, doWork, cleanup
  - Each test case is integrated into the life-cycle.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<suite comparator="text" contexts="SampleBaselineTest">
    <!-- suite-level argument. -->
    <arguments>
        <arg name="arg1">suiteArg1</arg>
        <arg name="arg2" type="String">suiteArg2</arg>
        <arg name="arg3" type="String">suiteArg3</arg>
        <arg name="arg4">suiteArg4</arg>
    </arguments>

    <testcase class="org.baselinetest.OutputArgumentsTest"
      arguments="args.xml">
        <arguments>
            <!-- testcase-level argument with
             optional type specification. -->
            <arg name="arg2" type="String">
            suiteTestCaseArg2
            </arg>
        </arguments>
    </testcase>

    <testcase class="org.baselinetest.OutputArgumentsTest"
      arguments="args.xml">
    </testcase>
</suite>
```

Define A TestCase

- A TestCase subclass
- Optional args
  - Can have test case steps
  - Iterated over during doWork.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<testcaseArguments>
    <arguments>
        <arg name="arg3">testcaseArg3</arg>
        <arg name="arg4">testcaseArg4</arg>
    </arguments>

    <!-- First step -->
    <testcaseStep>
        <arguments>
            <arg name="arg4">step1Arg4</arg>
        </arguments>
    </testcaseStep>

    <!-- Second step -->
    <testcaseStep>
    </testcaseStep>

    <!-- Third step -->
    <testcaseStep>
        <arguments>
            <arg name="arg4">step3Arg4</arg>
        </arguments>
    </testcaseStep>
</testcaseArguments>
```

Define a Test Case (cont.)

- Four life-cycle methods
  - Constructor
  - Init
  - Cleanup
  - DoWork
- Args and State available via API

```
public class OutputArgumentsTest extends TestCase {

    public OutputArgumentsTest(TestSuite testSuite) {
        super(testSuite);
    }

    public void init() {
    }

    public void cleanup() {
    }

    public void doWork() {
        ArgumentCollection argumentCollection =
            getArgumentCollection();
        printArgumentCollection(argumentCollection);
    }
```

Example Baseline Output

```
<TEST_SUITE>
<TEST_SUITE_INIT>
<TEST_CASE_INIT>
</TEST_CASE_INIT>
<TEST_CASE_INIT>
</TEST_CASE_INIT>
</TEST_SUITE_INIT>
<TEST_SUITE_DO_WORK>
<TEST_CASE>
<TEST_CASE_STEP>
# of inherited arguments: 5
key: arg1 value: suiteArg1
key: arg2 value: suiteTestCaseArg2
key: arg3 value: testcaseArg3
key: arg4 value: step1Arg4
key: arg5 value: contextArg5
</TEST_CASE_STEP>
<TEST_CASE_STEP>
# of inherited arguments: 5
key: arg1 value: suiteArg1
key: arg2 value: suiteTestCaseArg2
key: arg3 value: testcaseArg3
key: arg4 value: testcaseArg4
key: arg5 value: contextArg5
</TEST_CASE_STEP>
<TEST_CASE_STEP>
# of inherited arguments: 5
key: arg1 value: suiteArg1
key: arg2 value: suiteArg2
key: arg3 value: testcaseArg3
key: arg4 value: step3Arg4
key: arg5 value: contextArg5
</TEST_CASE_STEP>
</TEST_CASE>
</TEST_SUITE_DO_WORK>
<TEST_SUITE_CLEANUP>
<TEST_CASE_CLEANUP>
</TEST_CASE_CLEANUP>
<TEST_CASE_CLEANUP>
</TEST_CASE_CLEANUP>
</TEST_SUITE_CLEANUP>
</TEST_SUITE>
```