Introduction to Software Engineering: Unit Testing and Inspection

John T. Bell

Department of Computer Science
University of Illinois, Chicago

Based on materials from Bruegge & DuToit 3e and Pezze & Young

Review

What is the key feature that distinguishes agile development methods from more traditional classical Software Engineering approaches?

A. Embracing and expecting change, often by iteratively delivering the product one new feature at a time.
B. Pair programming.
C. Short periods of intense programming, (e.g. sprints), managed by very brief daily meetings.
D. Small teams performing rapid development.
E. Use of user stories in place of both requirements and acceptance testing.
Recall the V Model of SW Engineering

We looked at acceptance tests earlier
Now we need to look at unit tests and integration testing.
But first a little review . . .

Testing Concepts

- **Testing** is a systematic approach to find and identify faults. It is NOT intended to show that there are no faults. (A successful test finds faults.)
- A **fault** is a coding error that may lead to an erroneous state.
- An **erroneous state** is a run-time state as a result of a fault, that could lead to failure.
- **Failure** occurs when the system fails to operate as required.
Three Approaches to Dealing with Faults

1. **Fault Avoidance** – Very careful development methods to reduce the number of faults introduced in the first place.

2. **Fault Detection** – Debugging, testing, inspections, and other tools to find and remove faults before the product ships.

3. **Fault Tolerance** – Developing SW that continues to operate normally in the face of faults.

More Testing Concepts
Different Kinds of Tests, and When They are Performed

- **Test planning occurs early in the process.**
- **Unit and integration tests are planned last and performed first.**
- **Usability tests are run in parallel.**
- **System tests test the completed system.**

A Typical Test Case Specification

<table>
<thead>
<tr>
<th>Test-case identifier</th>
<th>DriveTrain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test location</td>
<td><a href="http://www2.in.tum.de/TrainSystem/test-cases/test1">http://www2.in.tum.de/TrainSystem/test-cases/test1</a></td>
</tr>
<tr>
<td>Feature to be tested</td>
<td>Continuous operation of engine for 5 seconds</td>
</tr>
<tr>
<td>Feature Pass/Fail Criteria*</td>
<td>The test passes if the train drives for 5 seconds and covers the length of at least two tracks.</td>
</tr>
<tr>
<td>Means of control</td>
<td>1. The StartTrain() method is called via a test driver StartTrain (contained in the same directory as the DriveTrain test).</td>
</tr>
<tr>
<td>Data</td>
<td>2. Direction of trip and duration are read from an input file <a href="http://www2.in.tum.de/TrainSystem/test-cases/input">http://www2.in.tum.de/TrainSystem/test-cases/input</a>. 3. If debug is set to TRUE, then the test case will output the system messages “Enter Track n, Exit Track n” for each n, where n is the number of the current track.</td>
</tr>
<tr>
<td>Test Procedure</td>
<td>The test is started by double-clicking the test case at the specified location. The test will run without further intervention until completion. The test should take no more than 7 seconds.</td>
</tr>
<tr>
<td>Special requirements</td>
<td>The test stub Engine is needed for the test execution.</td>
</tr>
</tbody>
</table>

*Oracle: A means of determining what the correct results of a test should be.
Component Inspections

• Careful detailed review of a component by a team of qualified reviewers. (Usually 3 – 5)
• “Components” may include code, documents, diagrams, procedures, or just about anything.
• Usually conducted with the aid of checklists pre-approved by all involved.
• There are several different published approaches to conducting inspections.

Typical Approach to (Code) Inspections I

• Review team may include (experienced) programmers and/or relevant experts.
• Components to be reviewed may be distributed prior to the first meeting.
• First meeting – Author presents and explains the component(s) to be inspected.
• Reviewers separate, and inspect the component(s) individually with aid of checklist(s).
Typical Approach to (Code) Inspections II

- Second meeting – Reviewers present their findings to the author.
  - Author not allowed to speak unless asked a question.
  - No suggestions for corrections or improvement given at this meeting. (Reviewer(s) may meet with author separately for that purpose later.)
- It is important to maintain an attitude of helpfulness, not criticism.
- Optional follow-up meeting may review changes.

Some benefits of inspections

- Can be applied to components that can’t be tested – documents, rarely encountered code.
- Code can be tested in the absence of necessary supporting HW and SW.
- More than one error can be found at once.
- Style and good practices can be inspected.
- Inspections typically find more faults faster.
- However inspections do not replace testing.
The following 3 slides (and some later material) are from Pezze & Young:

### Typical Checklists for Java Code

#### Java Checklist: Level 1 (inspection: single-pass, read through, context-independent)

<table>
<thead>
<tr>
<th>Checklist</th>
<th>Pass</th>
<th>Fail</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java type</td>
<td>yes</td>
<td>fail</td>
<td>comments</td>
</tr>
<tr>
<td>Java class</td>
<td>yes</td>
<td>fail</td>
<td>comments</td>
</tr>
<tr>
<td>Java method</td>
<td>yes</td>
<td>fail</td>
<td>comments</td>
</tr>
<tr>
<td>Java expression</td>
<td>yes</td>
<td>fail</td>
<td>comments</td>
</tr>
<tr>
<td>Java statement</td>
<td>yes</td>
<td>fail</td>
<td>comments</td>
</tr>
<tr>
<td>Java documentation</td>
<td>yes</td>
<td>fail</td>
<td>comments</td>
</tr>
</tbody>
</table>

#### Java Checklist: Level 2 (inspection: comprehensive review in context)

<table>
<thead>
<tr>
<th>Checklist</th>
<th>Pass</th>
<th>Fail</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java type</td>
<td>yes</td>
<td>fail</td>
<td>comments</td>
</tr>
<tr>
<td>Java class</td>
<td>yes</td>
<td>fail</td>
<td>comments</td>
</tr>
<tr>
<td>Java method</td>
<td>yes</td>
<td>fail</td>
<td>comments</td>
</tr>
<tr>
<td>Java expression</td>
<td>yes</td>
<td>fail</td>
<td>comments</td>
</tr>
<tr>
<td>Java statement</td>
<td>yes</td>
<td>fail</td>
<td>comments</td>
</tr>
<tr>
<td>Java documentation</td>
<td>yes</td>
<td>fail</td>
<td>comments</td>
</tr>
</tbody>
</table>

---

*Images and diagrams are not included in this text. Please refer to the original document for visual aids.*
Individual Items Have Detailed Explanations


FEATURE: CLASS DECLARATION: Are the following requirements satisfied?

ITEM: The visibility of the Class is consistent with the design document

Detailed checklist item reference:

Description: The fields and methods exposed by a class must correspond to those in the specification, which may be in the form of a UML diagram. If the class specializes another class, method header comments must specify where superclass methods are overridden or overloaded. Overriding or overriding methods must be intentionally consistent with ancestor methods. Additional public utility or convenience methods may be provided if well documented in the implementation.

The class name should be identical to the name of the class in the specifying document.

For ease of reference, Names of methods and fields may differ from those in the specifying document, provided header comments (class header comments for public fields, method header comments for public methods) provide an explicit mapping of implementation names to specification names. Order and grouping of fields and methods need not follow.

Motivations: Clear correspondence of elements of the implementation to elements of the specification facilitates maintenance and reduces integration faults. If significant deviations are needed (e.g., renaming a class or replacing or changing a public method signature), these are design decisions that should be discussed and reflected in the specifying document.

Example: The code implementing the following UML specification of class ComponentUnit should export fields and methods corresponding to the fields of the implementation of class ComponentUnit and its ancestor class LineUnit. Implementations that use different names for some fields or methods or that do not replicate the method pattern in class ComponentUnit are acceptable if properly documented. Similarly, implementations that export an additional method compared that specializes the default method export to aid test oracle generation are acceptable.

Even Test Plans Can be Inspected

TEST PLAN CHECKLIST: Comprehensive review in context

FEATURES (where to look and how to check):

Items (where to look and how to check):

ITEMS TO BE TESTED OR ANALYZED: For each item, does the plan include:

- A reference to the specification for the item
- A reference to installation procedures for the item

TEST AND ANALYZE APPROACH: Are the following requirements satisfied?

- The test and analysis techniques to be applied are cost-effective for issues of this type.
- The test and analysis techniques to be applied cover the relevant properties cost-effectively.
- The description is sufficiently detailed to identify major tasks and estimate time and resources.

PASS/FAIL CRITERIA: Are the following requirements satisfied?

- The criteria clearly indicate the pass/fail conditions.
- The criteria are consistent with quality standards specified in the test and analysis strategy.

UNSATISFY CRITERIA: Are the following requirements satisfied?

- The criteria clearly indicate conditions for suspending test and analysis due to excessive defects.
- The criteria clearly indicate conditions for resuming test and analysis after suspension and rework.

RISKS AND CONTINGENCIES: Are the following risks addressed?

- External risks, bias, or non-availability of suitable tools.
- Technology risks.
- Schedule risks.
Review

With respect to testing software, what is an “oracle”?

A. A means of knowing what the correct answer to the test should be.
B. A means of predicting at the beginning of the project what tests will be needed at the end of the project.
C. A test driver.
D. A test stub.
E. An oar that has been frozen solid.

Black Box vs. White Box Testing

- Black box tests are developed without knowledge of the internal workings of the component being tested.
  - Acceptance tests are typically black box.
- White box tests are developed based on knowledge of the internal workings of the component being tested. (a.k.a. clear box.)
  - Unit tests may be black box or white box.
- (Gray box tests use knowledge of the algorithms/data structures in use but not the specific code.)
Equivalence Classes / Testing

• It is rarely possible in practice to test all possible values of all input variables.

• An **Equivalence Class** is a range of input values for which the results are expected to be equivalent. (The same errors are expected.)

• **Equivalence Testing** strives to include at least one test case involving every (combination of) equivalence class(es).

---

Example: What equivalence classes can we identify for this method?

class MyGregorianCalendar {
    ...
    public static int getNumDaysInMonth(int month, int year) {...}
    ...
}

<table>
<thead>
<tr>
<th>Equivalence class</th>
<th>Value for month input</th>
<th>Value for year input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Months with 31 days, non-leap years</td>
<td>7 (July)</td>
<td>1901</td>
</tr>
<tr>
<td>Months with 31 days, leap years</td>
<td>7 (July)</td>
<td>1904</td>
</tr>
<tr>
<td>Months with 30 days, non-leap years</td>
<td>6 (June)</td>
<td>1901</td>
</tr>
<tr>
<td>Month with 30 days, leap year</td>
<td>6 (June)</td>
<td>1904</td>
</tr>
<tr>
<td>Month with 28 or 29 days, non-leap year</td>
<td>2 (February)</td>
<td>1901</td>
</tr>
<tr>
<td>Month with 28 or 29 days, leap year</td>
<td>2 (February)</td>
<td>1904</td>
</tr>
</tbody>
</table>
Boundary Testing is Related to Equivalence Classes

- **Boundary Testing** recognizes that errors most commonly occur at the boundary of equivalence classes.
- This is particularly true at the boundary between valid and invalid inputs. (Off by 1 errors.)
- Consider array[0], array[1], array[N-1], array[N].
- Also consider extremely large, small, or negative values, & uninitialized or missing values.

Example: What boundary cases can we identify for this method?

```java
class MyGregorianCalendar {
    ...
    public static int getNumDaysInMonth(int month, int year) {
        ...
    }
}
```

<table>
<thead>
<tr>
<th>Equivalence class</th>
<th>Value for month input</th>
<th>Value for year input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leap years divisible by 400</td>
<td>2 (February)</td>
<td>2000</td>
</tr>
<tr>
<td>Non-leap years divisible by 100</td>
<td>2 (February)</td>
<td>1900</td>
</tr>
<tr>
<td>Nonpositive invalid months</td>
<td>0</td>
<td>1291</td>
</tr>
<tr>
<td>Positive invalid months</td>
<td>13</td>
<td>1315</td>
</tr>
</tbody>
</table>

Table 11-3 Additional boundary cases selected for the getNumDaysInMonth() method.
Concept: Percent Coverage

• Based on some criteria, the % coverage of a test (suite) can be defined as the amount of that criteria covered / the total amount of that criteria present * 100%. For example:

\[
\text{% statement coverage} = \frac{\# \text{ lines tested}}{\text{total} \# \text{ lines}} \times 100
\]

Some criteria are more rigorous than others. For example:

• **Statement testing** – Strives to test every executable line of code.

• **Branch testing** – Strives to test every branch at every decision point, even if some involve no executable statements. (e.g. empty elses.)

• **Condition testing** – Strives to test every condition in both true and false cases. (More rigorous than branch testing for compound logic.)

• **(Path testing** tests all paths through the flowchart.)
Flowcharts are useful for branch testing.
State-based testing strives to test every state transition in a FSM

Polymorphism complicates the testing of OO Software

(Strategy design pattern)
Code Equivalent for Strategy Example

Original Code

```
public class NetworkConnection {
    //...
    private NetworkInterface nif;
    void sendByte msg() {
        queue.concat(msg);
    }
    if (nif.isSuccess()) {
        nif.send(queue);
        queue.setLength(0);
    }
}
```

Equivalent Code, with polymorphism expanded

```
public class NetworkConnection {
    //...
    private NetworkInterface nif;
    void sendByte msg[]() {
        queue.concat(msg);
    }
    if (nif.isSuccess()) {
        nif.send(queue);
        queue.setLength(0);
    }
    else if (nif instanceof Ethernet) {
        EthernetShift = (Ethernet) nif;
        ready = nif.isSuccess();
    } else if (nif instanceof WaveLAN) {
        WaveLANShift = (WaveLAN) nif;
        ready = nif.isSuccess();
    } else if (nif instanceof UMTS) {
        UMTSShift = (UMTS) nif;
        ready = UMTSShift.isSuccess();
    }
    if (ready) {
        if (nif instanceof Ethernet) {
            EthernetShift = (Ethernet) nif;
            nif.send(queue);
        } else if (nif instanceof WaveLAN) {
            WaveLANShift = (WaveLAN) nif;
            nif.send(queue);
        } else if (nif instanceof UMTS) {
            UMTSShift = (UMTS) nif;
            nif.send(queue);
        }
    }
    queue.setLength(0);
}
```

Flowchart Equivalent for Strategy Example
Integration testing requires a methodical approach

- Errors are most likely to occur at interfaces.
- Horizontal integration approaches:
  - Top-down – Requires stubs for lower layers.
  - Bottom-up – Requires drivers for upper layers.
  - Sandwich – Top and bottom, then the middle.
- Vertical integration – Test units together needed for a particular use-case, one by one.
- Big Bang – Test everything at once. (Not good.)

System testing tests the complete system.

- **Functional testing** – Tests functional requirements.
  - May be scenario / use-case based.
- **Usability testing** – Tests the user interface.
- **Performance testing** – Push (past) the limits.
- **Pilot testing** – Done in production environment.
  - Alpha testing is conducted in-house.
  - Beta testing is conducted by external evaluators.
- **Security testing** – May involve “tiger teams”.
And some more testing issues

- **Regression Testing** involves re-testing things that used to work, after other changes are made.
  - Usually a subset of the original tests of the item.
- **Automating testing** supports frequent regression testing.
  - Requires scaffolding to run automated tests, input files, oracles, reports on results, and possibly roll-backs.
  - Often run overnight, over the weekend, or when (major) changes are made to the project repository.

Test planning may require the consideration of dependencies
Which of the following is NOT an advantage of conducting inspections?

A. Inspections can be used to examine artifacts that cannot be tested, such as requirements documents.
B. Inspections can detect flaws in code that are undetectable via testing, such as incorrect comments or flaws that only lead to faults under rare circumstances.
C. Inspections can examine code even when supporting hardware, code, and/or data files are not available.
D. Inspections can find many faults at the same time, without getting stopped by the first fatal flaw that is encountered.
E. Inspections can totally replace testing, if done properly and completely.