The Ongoing Revolution in Software Testing

Presentation by Jon Elverkilde

February 15, 2008
1. Introduction
   - Background
   - Motivation

2. Common assertions
   - The role of testers
   - Test Planning and Documentation
   - The Practice of Testing

3. Conclusions
   - Agile and test-driven development
   - Context Driven Testing
   - Discussion
Cem Kaner

- Professor of Software Engineering, Florida Institute of Technology.
- Ph.D. in experimental psychology and a law degree.
- Author and co-author of 4 books.
Some assumptions about testing are wrong.
Some assumptions about testing are wrong.

Programs where linear and shorter (1960-70).
Some assumptions about testing are wrong.

Programs where linear and shorter (1960-70).

Software is becoming more complex and so is development.
Some assumptions about testing are wrong.
- Programs where linear and shorter (1960-70).
- Software is becoming more complex and so is development.
- These assumptions may lead to bad software.
Reasons

- Primary reason: Find bugs
Reasons

- Primary reason: Find bugs
- Conformance to specification/regulation. Verify correctness, etc.
Primary reason: Find bugs

Primary reason: Prove correctness.

Conformance to specification/regulation. Verify correctness, etc.
Reasons

- Primary reason: Find bugs
- Primary reason: Prove correctness.
- Conformance to specification/regulation.
- Verify correctness, etc.
- You may find what you expect and miss the rest.
The role of testers

Common assertions

The Practice of Testing

Reasons

- Primary reason: Find bugs
- Primary reason: Prove correctness.
- Testers are advocates of quality / quality assurance groups.
- Conformance to specification/regulation. Verify correctness, etc.
- You may find what you expect and miss the rest.
Primary reason:
Find bugs

Primary reason:
Prove correctness.

Testers are advocates of quality / quality assurance groups.

Conformance to specification/regulation.
Verify correctness, etc.

You may find what you expect and miss the rest.

Testers can’t (and shouldn’t) assure quality, but help by assessing quality.
Testers should work independently of programmers.
Conventions

- Testers should work independently of programmers.
- Based on fears of bias, etc. TDD proves this wrong and has benefits; Unit tests, fast bug finding.
Conventions

- Testers should work independently of programmers.
- Teams should use development models, like the Waterfall.
- Based on fears of bias, etc. TDD proves this wrong and has benefits; Unit tests, fast bug finding.
Testers should work independently of programmers.

Teams should use development models, like the Waterfall.

Based on fears of bias, etc. TDD proves this wrong and has benefits; Unit tests, fast bug finding.

Locks down details before implications (cost, difficulty, etc.) are known. In an evolutionary model, the trade-off is between features and time to market (not reliability).
Conventions

- Testers should work independently of programmers.
- Teams should use development models, like the Waterfall.
- Testers should base tests on documented characteristics of the program.
- Based on fears of bias, etc. TDD proves this wrong and has benefits; Unit tests, fast bug finding.
- Locks down details before implications (cost, difficulty, etc.) are known. In a evolutionary model, the trade-off is between features and time to market (not reliability).
Conventions

- Testers should work independently of programmers.
- Teams should use development models, like the Waterfall.
- Testers should base tests on documented characteristics of the program.
- Based on fears of bias, etc. TDD proves this wrong and has benefits; Unit tests, fast bug finding.
- Locks down details before implications (cost, difficulty, etc.) are known. In a evolutionary model, the trade-off is between features and time to market (not reliability).
- Specification describes how the program is supposed to work. This approach may narrow the testers’ thinking.
Test Planning and Documentation

- Testers should specify the expected result of tests in advance.

A program can fail in many ways. "High volume automated tests" (i.e. random) exposes memory leaks, etc. Testers (and developers) learn about the program during development. Resources will be wasted, if the program changes. Early key decisions make greater inertia. (Other) testers supposedly learn about the test and program design. This is not proved, rather testers are likely to be influenced. Industry worst practice!
Test Planning and Documentation

- Testers should specify the expected result of tests in advance.

- A program can fail in many ways. “High volume automated tests” (i.e. random) exposes memory leaks, etc.
Test Planning and Documentation

- Testers should specify the expected result of tests in advance.
- Testers should design most tests early in development.
- A program can fail in many ways. “High volume automated tests” (i.e. random) exposes memory leaks, etc.
Test Planning and Documentation

- Testers should specify the expected result of tests in advance.
- Testers should design most tests early in development.
- A program can fail in many ways. “High volume automated tests” (i.e. random) exposes memory leaks, etc.
- Testers (and developers) learns the program during the development. Resources will be wasted, if program changes. Early key decisions make greater inertia.
Test Planning and Documentation

- Testers should specify the expected result of tests in advance.
- Testers should design most tests early in development.
- Testers should document manual tests in great detail so [...]

- A program can fail in many ways. “High volume automated tests” (i.e. random) exposes memory leaks, etc.
- Testers (and developers) learns the program during the development. Resources will be wasted, if program changes. Early key decisions make greater inertia.
Test Planning and Documentation

- Testers should specify the expected result of tests in advance.
- Testers should design most tests early in development.
- Testers should document manual tests in great detail so [...]

- A program can fail in many ways. “High volume automated tests” (i.e. random) exposes memory leaks, etc.
- Testers (and developers) learns the program during the development. Resources will be wasted, if program changes. Early key decisions make greater inertia.
- (Other) testers supposedly learns about the test and program design. This is not proved, rather testers are likely to be influenced. *Industry worst practice(!)*
The Practice of Testing

- Tests should cover every line and branch in the program.
The Practice of Testing

- Tests should cover every line and branch in the program.
- Again this can narrow attention on specific test attributes. Programs might have interdependencies, these tests will miss.
The Practice of Testing

- Tests should cover every line and branch in the program.
- We can tell how close we are to release by examining the "bug curve".

Again this can narrow attention on specific test attributes. Programs might have inter dependencies, these tests will miss.
The Practice of Testing

- Tests should cover every line and branch in the program.
- We can tell how close we are to release by examining the “bug curve”.
- Again this can narrow attention on specific test attributes. Programs might have inter dependencies, these tests will miss.
- A project will (perhaps) naturally fit a curve. By using this, e.g. to predict a release, the model no longer fits.
Agile and test-driven development

- *Agile development* is characterized by iterations (of weeks);
Agile development is characterized by iterations (of weeks);
Each iteration consists of planning, analysis, design, coding, etc.
Agile development is characterized by iterations (of weeks);
Each iteration consists of planning, analysis, design, coding, etc.
Every iteration starts with a complete review and ends with a (working) release.
**Agile development** is characterized by iterations (of weeks);
- Each iteration consists of planning, analysis, design, coding, etc.
- Every iteration starts with a complete review and ends with a (working) release.
- In *TDD* the project cycle starts with adding a test, then writing the code, then refactoring.
The Seven Basic Principles of the Context-Driven School

1. The value of any practice depends on its context.
2. There are good practices in context, but there are no best practices.
3. People, working together, are the most important part of any project’s context.
4. Projects unfold over time in ways that are often not predictable.
5. The product is a solution. If the problem isn’t solved, the product doesn’t work.
6. Good software testing is a challenging intellectual process.
7. Only through judgment and skill, exercised cooperatively throughout the entire project, are we able to do the right things at the right times to effectively test our products.
The Seven Basic Principles of the Context-Driven School

- The value of any practice depends on its context.
- There are good practices in context, but there are no best practices.
- People, working together, are the most important part of any project’s context.
- Projects unfold over time in ways that are often not predictable.
- The product is a solution. If the problem isn’t solved, the product doesn’t work.
- Good software testing is a challenging intellectual process.
- Only through judgment and skill, exercised cooperatively throughout the entire project, are we able to do the right things at the right times to effectively test our products.
What “school”/style is DIKU preaching?
Discussion

- What “school”/style is DIKU preaching?
- Can some of this be applied at DIKU? (academia in general?)
Discussion

- What “school”/style is DIKU preaching?
- Can some of this be applied at DIKU? (academia in general?)
- If not, could it pose a problem?