

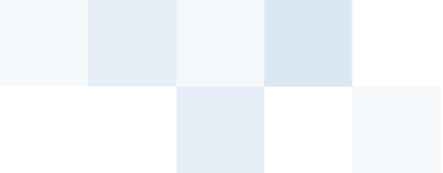


# Are Your Continuous Tests Too Fragile for Agile

Andrey Madan  
Sr. Solution Architect, Parasoft

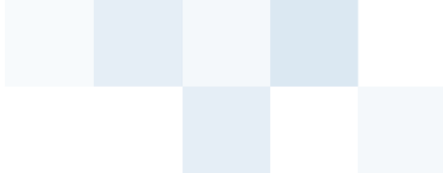
2015-03-12

- Challenges of Continuous Testing in Agile
- Testing defined
- Testing is hard (no one wants to do it)
- Best practices
- A modest proposal: Spring cleaning


- 
- QA is part of development, not an afterthought
  - Review developer testing efforts
  - Iterative during development
  - Fast discovery and reporting of defects
  - Influence development policy

- Need to be fast
- Need to be flexible
- Testing can be overlooked if not baked in
- Architectural compromises from schedule

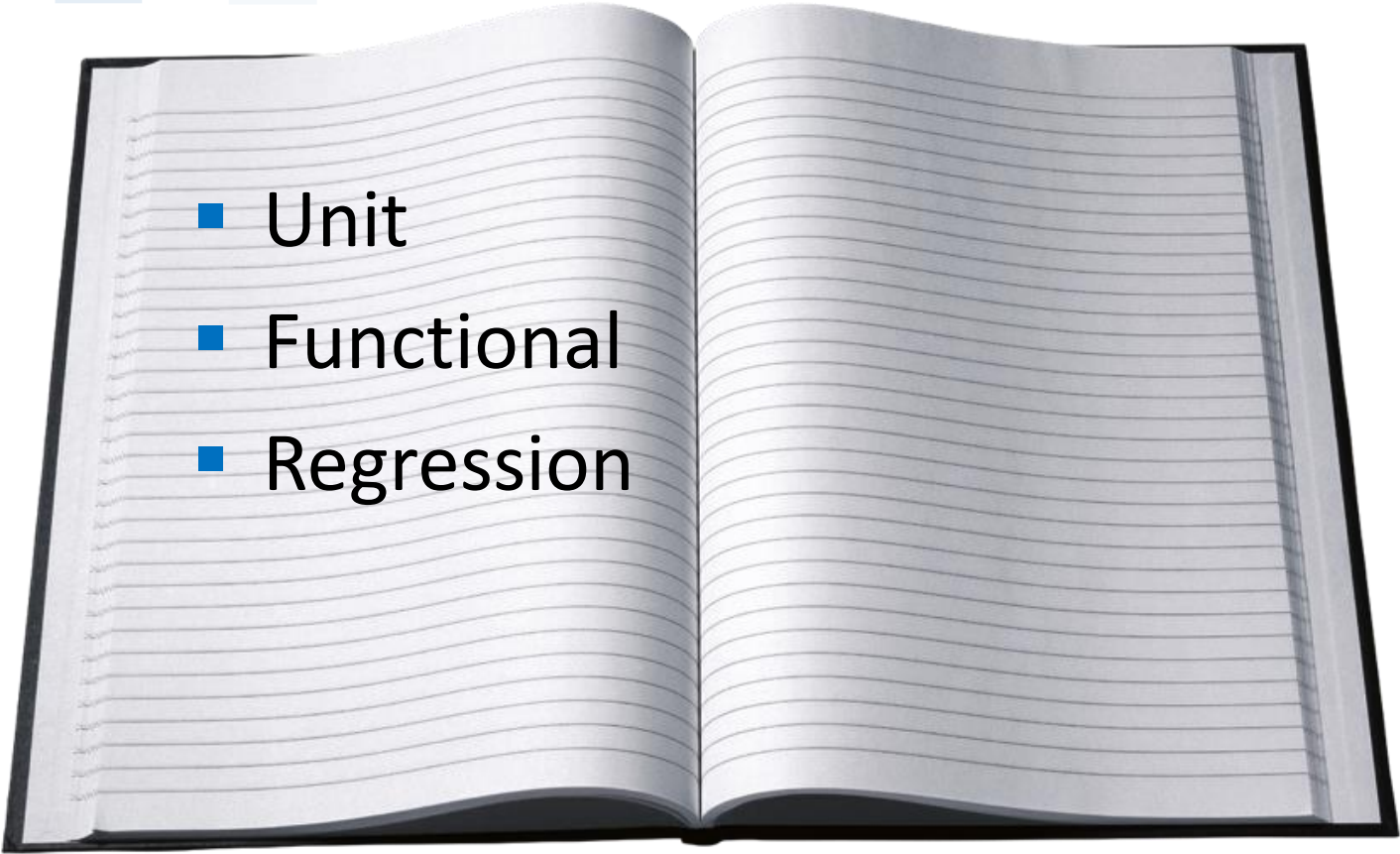


- 
- A decorative graphic consisting of a grid of light blue squares of varying sizes, arranged in a pattern that tapers off to the right.
- Tests must produce binary decision go/no-go
  - Reuse unit test and functional tests from dev to QA
  - High level of automation
  - Requires disciplined mature process
  - Testing must *automatically* answer
    - Is it stable
    - Will it do what it's supposed to do



- 
- A decorative graphic consisting of a grid of squares in various shades of blue and white, arranged in a pattern that tapers to the right.
- How do you know when you're done?
  - How do you know if a fix for a minor bug broke a major function of the system?
  - How can the system evolve into something more than is currently envisioned?
  - Testing, both unit and functional, needs to be an integrated part of the development process.

- Perceived as “burdensome”
  - Too much effort
  - Too much time
  - We’ re on a schedule – no time to test.
    - Why is there always time to do it over, but never to do it right?
- Developer Ego
  - “I don’t need to test my code – I don’t make mistakes.”
    - Testing means I have to admit that I might have made a mistake.
  - I don’t like to test – it’s beneath me.
  - Testing isn’t fun – I just want to code!
  - That’s for the QA dept.

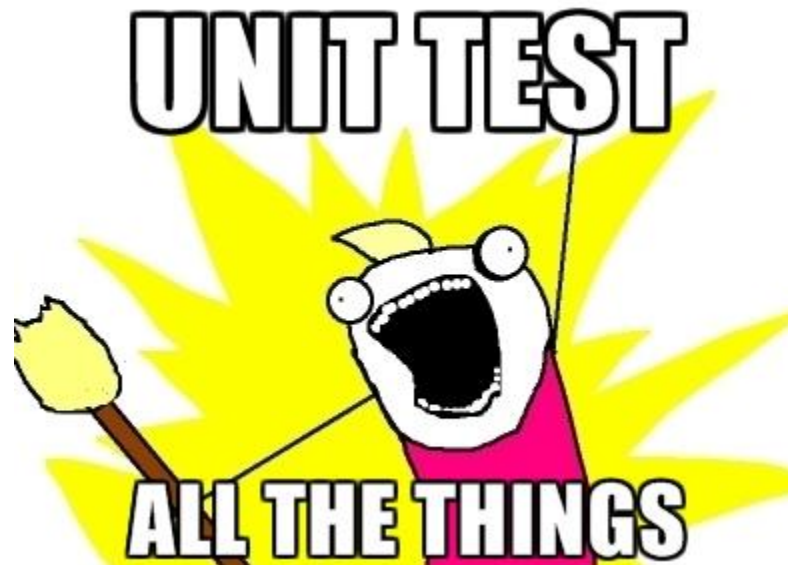
- 
- Unit
  - Functional
  - Regression



Unit tests that validates your method behaves a certain way given a specific parameters

- Regressions testing
  - Validate behavior of existing code hasn't changed
  - Scheduled and run automatically
- Test Driven Development
  - Validate behavior of new code to be written
  - Created before developing, turned into regression tests after

# Does unit testing provide value?



- Incarnations of functional tests
  - Unit tests (JUnit, CppUnit, NUnit, etc)
  - Integration tests
  - Scripted tests
  - Manual tests
- Ways to create functional tests
  - Manually written
  - Machine generated from application tracing
  - Machine generated from capturing traffic
  - Machine generated from recording clicks

- Contributors to the regression test suite
  - Functional unit tests that pass
  - Machine generated unit tests for current behavior
  - Unit test written by developers and testers
  - Tests written to validate bug fixes
- Regression test failures
  - Unwanted change in behavior – fix the code
  - Intentional change in specification – update test controls

## Unit Tests

- The code is doing things right
- Written from programmers perspective
- Confirm code gives proper output based on specific input

## Functional Tests

- The code is doing the right thing
- Written from users perspective
- Confirm the system does what the user expects it to do

## Regression Tests

- It still works

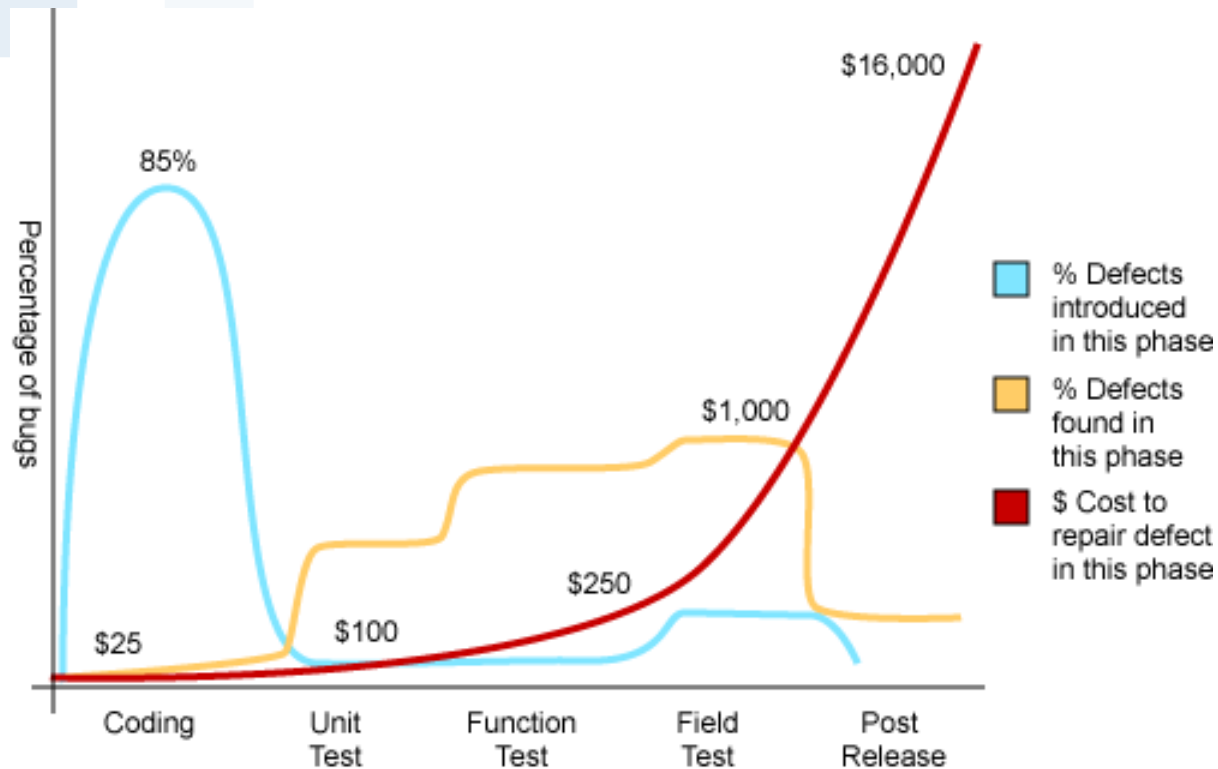


Software bugs are inserted during coding phase, but often aren't found until functional testing, or worse they are found by customers. The longer a bug stays in the system, the more expensive it is to fix it.

Unit testing is a strategy that allows you to find out if your system is behaving the way you expect early in the SDLC.

Unit testing allows for easy regression testing, so you can find out quickly whether you have introduced some unexpected behavior into the system.

# The Cost of Defects



Source: *Applied Software Measurement*, Capers Jones, 1996




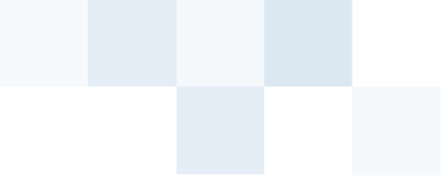
- Unit tests require a LOT of care
- Unit testing is a continuous operation
- Test suite maintenance is a commitment
- Control data (assertions) in the code

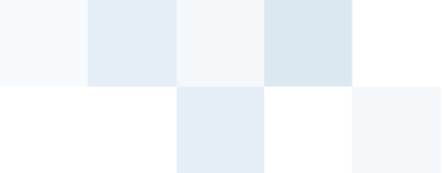


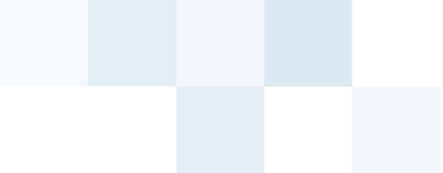
- Take existing unit tests and schedule them into build process
- Run and fix tests as part of build process
  - Ideal goal: 0 failures and 100% code coverage
  - Realistic goal: 10% test failures and 80% code coverage
- If the unit test suite is noisy, don't qualify the build based on assertion failures



- 
- A decorative graphic consisting of a grid of squares in various shades of blue and white, arranged in a pattern that tapers to the right.
- Maintainability
  - Proper assertions
  - Nightly runs
  - Tests should be:
    - Automatic (no human input or review)
    - Repeatable (no strange dependencies)

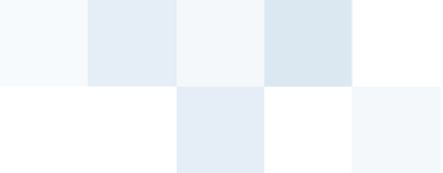
- 
- Leverage your existing suite
  - Clean up incrementally
  - Set policy based on current results
  - Tighten policy when in compliance

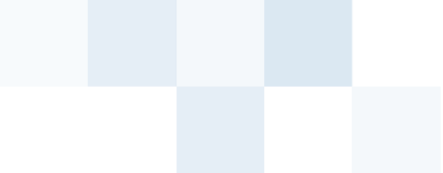
- 
- Maintaining is more of a problem than creating.
  - Don't create a test if you can't or don't plan to maintain it.
  - Creating as you go creates a better test suite
  - Tests not run regularly become irrelevant

- 
- Create the test when you're writing the code
  - Have a test for every task you complete
  - Increases understanding
  - Improve assertions

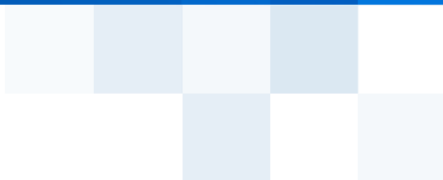
- Associate tests with Code and Requirements
- Naming Conventions should have meaning
- Use code comment tags @task @pr @fr
- Use check-in comments @task @pr @fr
- Improve change-based testing

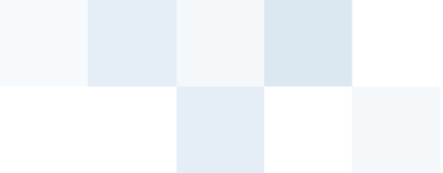


- 
- Don't be lured by 100% coverage
  - Auto assertions are not meaningful unless validated
  - Review the tests as you work on the code

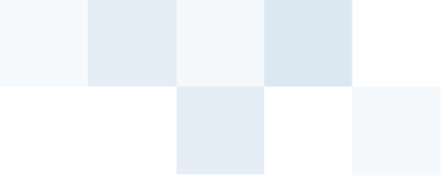
- 
- Collect and combine Code Coverage:
    - Unit Testing
    - Component Testing
    - Functional Testing

- Code for unit tests should be reviewed
- Consider doing test-design review
- Ensures you're testing what you think you are
- Helps reduce noise
- Helps improve understanding
- Review the code and the test code together

- 
- A decorative graphic consisting of a grid of light blue squares of varying sizes, arranged in a pattern that tapers to the right.
- Frequently Skipped
  - Prevents crashes
  - Prevents death spiral
  - Rule of thirds
    - 1/3 basic functionality
    - 1/3 advanced features / infrequent settings
    - 1/3 error handling

- 
- Verifies that the code behaves as intended
    - True/false
    - Null or not null
    - Specific values
    - Complex conditions



- 
- Excuses:
    - My code already has console output
    - I can check it visually/manually
  - Binary output (pass/fail)
  - Part of overall test suite
  - No human intervention

## Introduced coding error

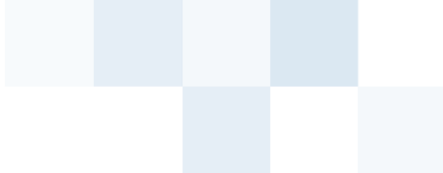
- Fix the new code

## Functionality changed

- Update the assertion

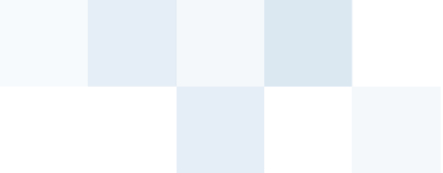
## Improper assertion

- Comment it out

- 
- A decorative graphic consisting of a grid of squares in various shades of blue and white, arranged in a pattern that tapers to the right.
- Ease of creating assertions leads to poor choices
  - Anything CAN be checked – but should it be?
  - Assertions should be connected to what you're trying to test
  - Assertions should be logically invariant
  - Check values by range, not equivalence
  - “What do I expect here logically?”

- How often do you WANT to manually verify an assertion?
- Daily or near-daily checking indicates poor assertion.
- Troublesome possibilities:
  - Date
  - User or machine name
  - OS
  - Filesystem names and paths

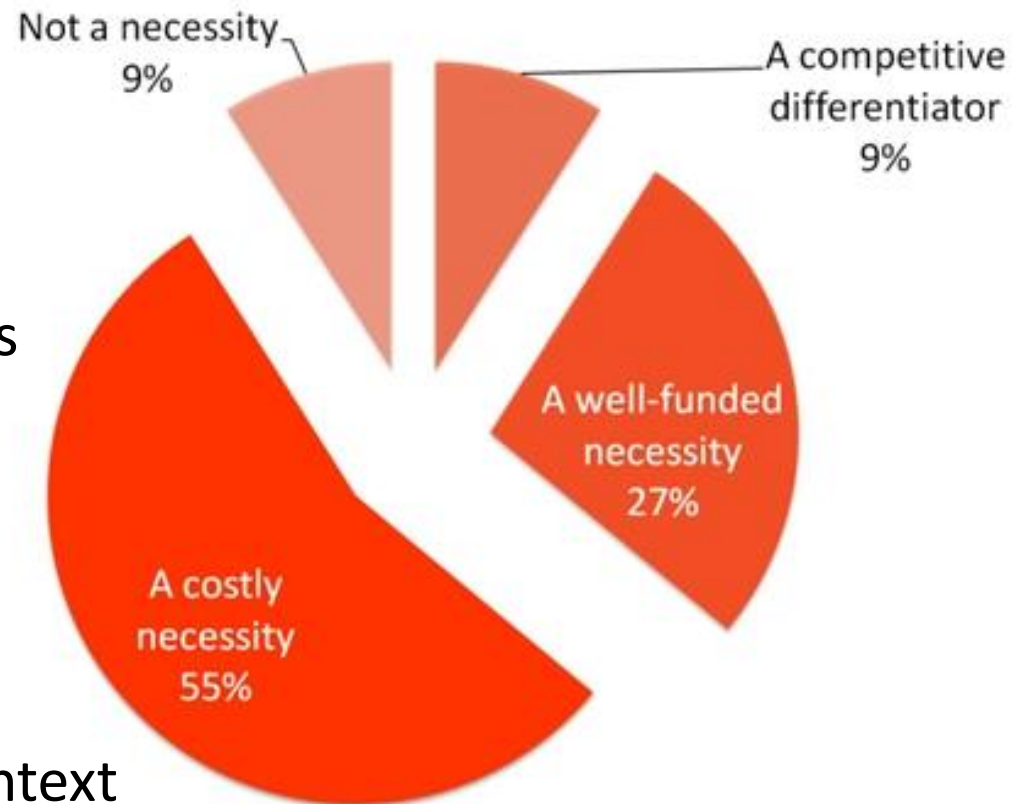


- 
- Error was found after code changed
  - Test suite never caught the error
  - Update unit-test with new assertion
  - Write new unit tests as necessary





How do you think senior management perceive the activities of the software testing and quality assurance function?



- Need tests to reduce risks
- Testing takes time from features/fixes
- Not enough time
- Not enough people
- Tests and reports lack context

## UNIT TESTS



**Y U NO WRITE MORE UNIT TESTS?**

Troll.me

- Hundreds of failing or broken tests
- Test requirements unclear
- Existing tests are dubious
- Deadline pressure

- Throw out the useless, old junk
  - Identify and eliminate “noisy” tests
- Organize the remaining chores
  - Prioritize unit test tasks
- Keep it clean
  - Establish and follow a routine test maintenance plan



- Tests that are fragile
  - Pass/Fail/Incomplete results frequently change
  - Frequent changes are required to keep up-to-date
- Tests that are outdated
  - More than “n” releases have occurred despite this test failure
  - The test applies to legacy code that is frozen
  - The test has not been executed for “x” period of time
- Tests that are unclear
  - It takes more than 5 minutes to understand why the test exists
  - The failure message is incorrect/unhelpful/irrelevant
  - There is no associated requirement or defect
- Be merciless!



- Management and Development need to co-define a Policy
  - Clear testing goals that tie tests to business needs
  - Clear delineation of responsibility
    - Test creation and deletion
    - Test failure resolution
    - Naming conventions
    - Peer review
  - Clear and frequent communication
    - Understand the technical and business impact
    - Enhance traceability to project requirements
    - Ask questions
  - Focus on safety/quality, not coverage
- Keep the number of tests minimal
- Run all tests on a regularly scheduled, frequent basis
- Clean as you go



- Find and delete “noisy” tests
- Use policies to prioritize, assign, and plan
- Co-design and follow a process to keep it clean





- Email: [andrey.madan@parasoft.com](mailto:andrey.madan@parasoft.com)
- Web
  - <http://www.parasoft.com/jsp/resources>
- Blog
  - <http://alm.parasoft.com>

## ■ Social

- Facebook: <https://www.facebook.com/parasoftcorporation>
- Twitter: @Parasoft
- LinkedIn: <http://www.linkedin.com/company/parasoft>
- Google+: +Parasoft
- Google+ Community: Development Testing Because its Better

