Software Testing for Continuous Delivery

Seminar 11: Quality Assurance

Dr. Byron J. Williams
October 07, 2019

UF Herbert Wertheim College of Engineering
Department of Computer & Information Science & Engineering
UNIVERSITY *of* FLORIDA

Content sourced from: “Software Quality Engineering” by Jeff Tian
Images courtesy Google Image Search

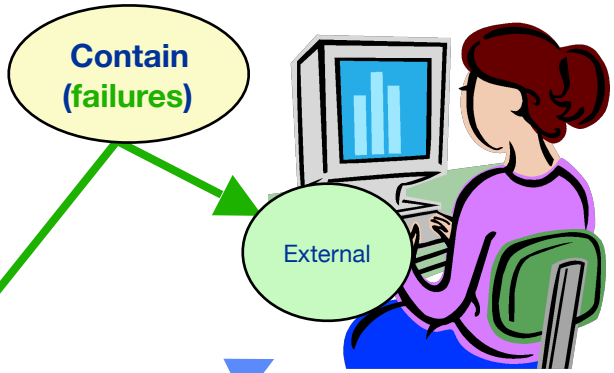
FCS Research

Legacy Code - Testing Challenges

- "Our system can't be tested"
- "Unit testing/test-driven development only works in green field projects."
- complex or botched architecture, inconsistent design, or simply code written with everything but testability in mind
- Problem: dependencies
  - Different parts of a system depend on each other in different ways, and the exact nature of these dependencies affects testability.

Relationship b/w Program Elements

- In object-oriented systems, the tested object will make use of other objects, from now on called ***collaborators***
  - Some are heavyweight and deeply entrenched in the system;
  - others are simple and provide very narrow functionality
- We use **test doubles** to deal with collaborators
- systems are usually composed of thousands of classes, and their instances form intricate webs of relations between collaborating objects

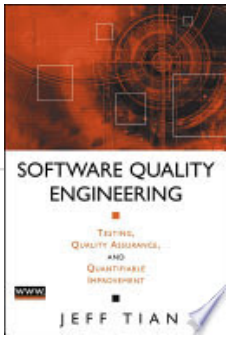


Correctness: few problems with limited damage to customers

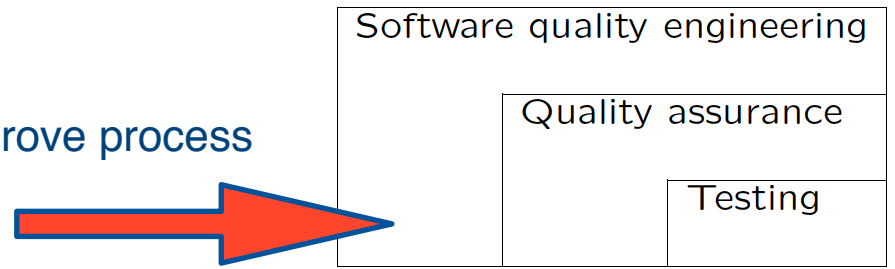
Quality Assurance

To ensure that few, if any, defects remain in the software when it is delivered to its customers or released to the market

Software Quality Engineering



- Goal - Ensure software quality
- Premise - Quality cannot be achieved by assessing an already completed product. The aim, therefore, is to **prevent quality defects** or deficiencies in the first place, and to make the products assessable by quality assurance **measures**
  - Defined: ***Quality assurance*** is the systematic activities providing evidence of the fitness for use of the total software product
- QE Outcome
  - Learn from and package experiences to improve process



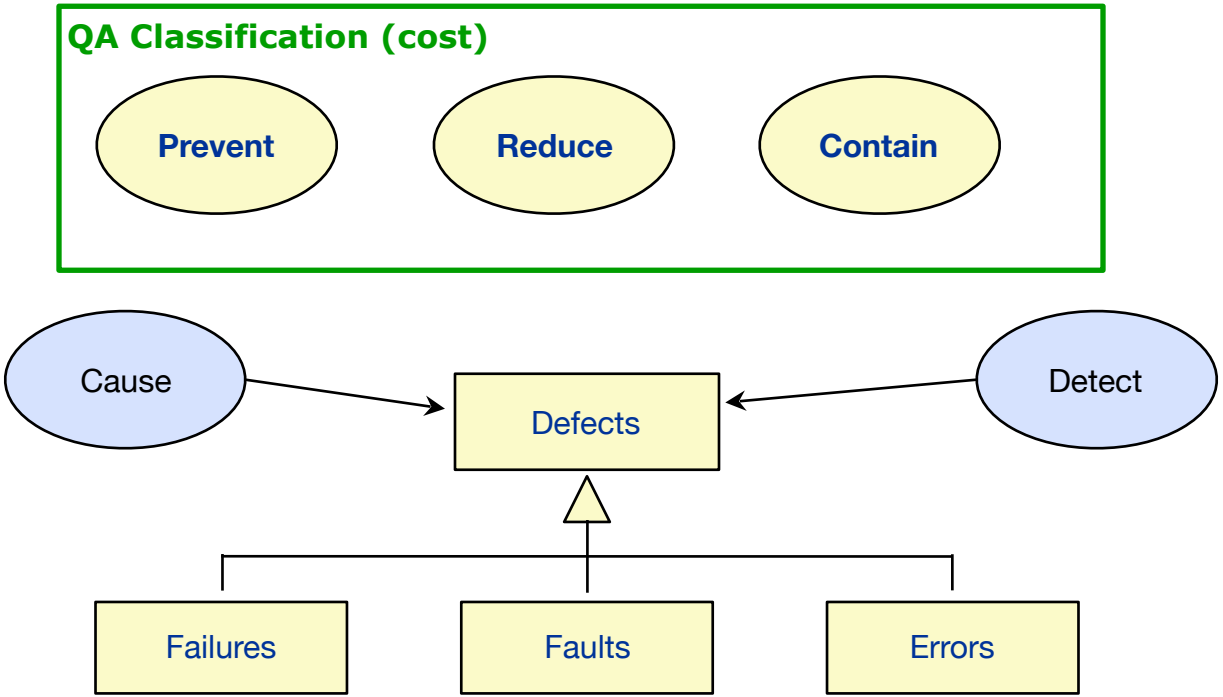
Reliability

- **Reliability**: Probability of failure-free operation for a specific period or a given set of input under a specific environment

- **Accomplished through**
  - availability, latency, performance, efficiency, change management, monitoring, emergency response, and capacity planning

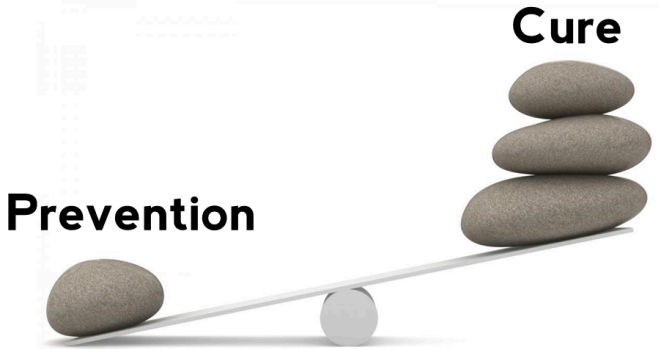
- Key Measures - System Quality
  - Mean-time-to-**Failure** (**MTTF**) - how often does the thing stop working
  - Mean-time-to-**Repair** (**MTTR**) - once it stops working, how long does it take until you fix it

QA: Dealing with Defects



Category 1: Defect Prevention

- Reduce the **chance** of defect injection
- Approach depends on source
  - Human misconceptions (most important)
    - Education and training
  - Imprecise design and implementation
    - Formal methods
  - Non-conformance to processes or standards
    - Process conformance or standard enforcement
- Important to establish the correct root-cause



"an ounce of prevention is worth a pound of cure"
— *Benjamin Franklin*

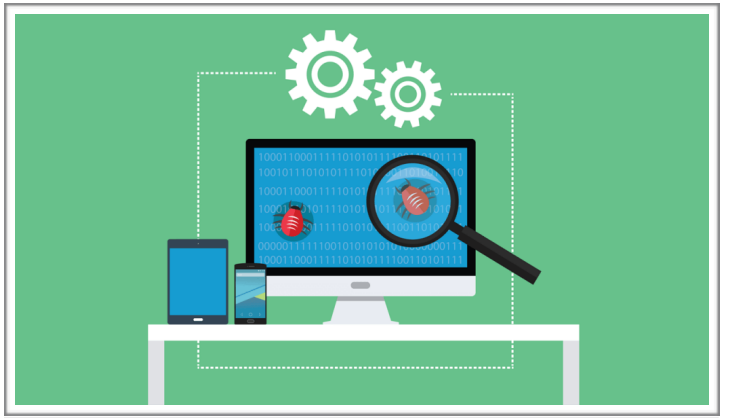
Defect Prevention: **Education and Training**

- Tenets
  - People are the **most important factor** in quality and success
  - Elimination of misconception will reduce the probability of defect injection
- Product and Domain Specific Knowledge
  - Unfamiliarity could lead to misunderstandings
- Software Development Expertise
  - Poorly written requirements/design can lead to problems
- Knowledge about Tools, Methods, Techniques
  - Lack of knowledge could lead to misuse
- Development Process Knowledge
  - Hard to properly implement the process if developers do not understand it

Greatest payback

Category 2: Defect Reduction

- **Unrealistic** to expect Defect Prevention step to stop all defects
- Different approaches
  - Inspection
  - Testing
  - Other techniques (e.g., static analysis)
- Commonly applied to code (can be used earlier)
  - Find problems earlier
  - Less expensive to fix
- Allow for causal analysis



Defect Reduction: **Inspection**

- Examination of software artifacts by humans to find faults
- Characteristics
  - ↳Critical analysis of requirements/code/design
  - ↳Multiple people
  - ↳Direct detection of faults
  - ↳Faults removed and verified
  - ↳Different processes can be followed
  - ↳Different amounts of formality can be used
  - ↳Approach from varying perspectives (tester, UI designer, user, sys admin)

Defect Reduction: Testing

- **Execution of software and checking results**
  - Locates failures
  - Isolate and fix the fault(s) that led to the failure
- When to test
  - **Need some executable**
  - Unit tests --> components —> integration, system --> acceptance test of entire system
  - Alpha & Beta testing
  - Can also use prototypes
- Questions:
  - **What to test? When to stop? Lots more…**
  - checklists, coverage information, usage scenarios, reliability…others *(next topics)*

Defect Reduction:
Observations

- Many other techniques available

- Important to determine risky components
  - Typically 80% of faults occur in 20% of components
  - Often these components can be identified with appropriate metrics (i.e. size, complexity)