Quality Assurance

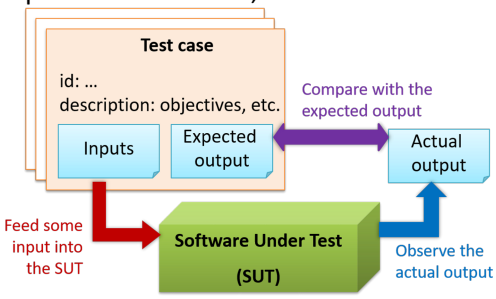
Tuesday, 4 December 2018 7:31 PM

Quality Assurance (QA)

- Testing, static analysis, code reviews, formal verification
- Validation = follow user requirements (right system built?)
- Verification = implementation of requirements (system built right?)
- Code Reviews
  - Pair programming, PRs, formal inspections
  - Can detect functionality defects and coding standard violations
  - No test drivers/ stubs needed, but is manual and error-prone
- Static Analysis: code not executed
  - Unused variables, unhandled exceptions, style errors, statistics (inbuilt in IDEs)
  - Java: CheckStyle, PMD, FindBugs
  - E.g. STAN
  - Static analysis = checkstyle, dynamic analysis = coverage
  - Linters: locate areas where code can be cleaner, check for potential errors
- Formal Verification: mathematical techniques
  - Prove absence of errors (vs testing: prove presence)
  - Proves compliance with specification, but not utility
  - Mostly in safety-critical-software (specialised notations and knowledge)

Testing

- Test-savvy coders are good coders
- Dynamic (involves execution not just examining code), finite and selected (only some scenarios tested), expected behaviour involved
- Observe SUT under specified conditions, record results and evaluate



- Test Case: input and expected behaviour
  - Unique identifier : e.g. TC0034-a
  - Descriptive name: e.g. vertical scrollbar activation for long web pages
  - Objectives: e.g. to check whether the vertical scrollbar is correctly activated when a long web page is loaded to the browser
  - Classification information: e.g. priority - medium, category - UI features
  - Cleanup, if any: e.g. empty the browser cache.
- For every test case: feed input, observe output, compare actual vs expected
  - Mismatch = failure (due to defect/ bug)
- Testability: how easy it to test your code
  - Depends on design and implementation

Testing Types

- Unit Testing: single class/ method
  - Theory: done in isolation of other things, use stubs
    - Stubs isolate SUT from dependencies
    - Other types of replacements: mocks, fakes, dummies, spies
  - Test drivers replace UI, pretend to be user probing SUT

- Integration Testing: focus on interactions, things work together
    - Run using actual dependencies (vs stubs in unit testing)
    - Test a specific interaction
    - Hybrid: U/T and I/T together, no stubs used - for bottom up
- System Testing: test against system specification, usually by QA team
    - Test cases based on specified external behaviour
    - Done in test bed/ test environment
    - Includes NFR testing: performance, load, security, compatibility/ interoperability, usability, portability
- Acceptance Testing: User Acceptance Testing (UAT)
    - Done in real/ simulation environment, usually by real user
    - Test against requirement specifications (by user), e.g. Based on use case specifications
        - Usually limited to MSS, in terms of problem domain
    - More focus on positive tests (SUT works normally)
    - System tests are typically more extensive, but system tests pass does not mean UATs pass (test bed environment, validation failure) -> though both usually test the whole system
- Alpha and Beta Testing
    - Alpha: users test under controlled conditions set by team
    - Beta: users test under natural work setting (e.g. Open beta release)
- Dogfooding: creators use their own product to test the product, can improve product design
- Developer Testing: done by developers (vs testers or end-users)
    - Important to do early to locate cause of test case failure, time for major reworks (especially if one bug hides other bugs) -> minimise delayed delivery
    - Bug is cheaper to fix the earlier it is found
- Scripted vs exploratory (use a combination of both)
    - Scripted: testing based on a set of test cases, does not mean automated
        - More systematic
    - Exploratory: follow-up test case based on previous test cases
        - Reactive, error-guessing, attack-based, bug hunting
        - Based on tester's prior experience and intuition
- Regression testing: unintended and undesirable effects
    - Retesting for regressions, best done frequently (and automated)

Test Automation

- E.g. Junit
- Increased precision
- CLI Apps: input/ output redirection, then FC (file compare, diff on Unix)

```
java AddressBook < input.txt > output.txt
```

- Test driver: drives SUT with inputs and verifies expected behaviour
- GUI tests: minimise logic, or test separately
    - Vs API testing: call operation and check expected return value
    - TestFX, VisualStudio 'record replay', Selenium for web apps
    - Can be run in headless mode (no display)

Test Coverage

- 100% coverage != bug-free (missing code)
- Function/ method coverage, statement coverage, decision/ branch coverage (if statement evaluated to both true and false), condition coverage (A && B boolean conditions), path coverage*[most intensive!] via Control Flow Graph (CFG), entry/ exit coverage
- Measured using coverage analysis tools (found in IDEs, e.g. Eclipse EclEmma)

Dependency Injection

- Inject new objects to replace dependencies (e.g. Inject test-friendly stubs)
    - Implementing using polymorphism

Test-Driven Development (TDD)

- Kent Beck - write tests before writing code
  - Write a test -> write code to pass test -> repeat (need not start will all test cases)
  - Do not write any code unless there is a failing test
  - Do not write more tests than necessary to fail
  - Do not write more code than needed to pass the tests
  - Can use JUnit
- Pros:
  - Code is definitely testable
  - Testing not neglected because of time pressures
  - Developer must think about what component should do, optimises effort
  - Developer must automate tests

Test Case Design

- Exhaustive testing not practical in non-trivial SUTs
  - Every test case increases cost
- Effective (find more bugs) vs efficient (find more bugs per test case)
  - Find bugs other tests cannot
- Positive vs Negative TCs
  - Positive = produce expected/ valid behaviour
  - Negative = invalid/ unexpected situation (e.g. assertFalse, Error message)
- Black Box vs Glass Box
  - Black = specification/ responsibility-based (design TC using specified external behaviour/ expectations of code)
  - White = structured/ implementation (design TC using known SUT implementation)
  - Gray = mix
- Equivalence Partitions: combine groups that are LIKELY to be treated same way by code (same exceptions..)
  - E.g. Square (int i) --> test for overflow
  - Avoid testing too many inputs from one partition, ensure all partitions are tested
  - EPs of target object of method call, input params, global variables and other objects accessed
  - [P1] **[P2]** [P3]...
- Boundary Value Analysis: bugs congregate near boundaries
  - Boundaries not always present, e.g. Prime numbers
  - Boundary values = corner cases
  - Below, at, above boundary

Combining Test Inputs

- Underline invalid inputs (chosen using EP or BVA)
- All combinations, at least once strategy, all pairs (all combinations between any given pair of inputs)
  - All pairs: bug is rarely the result of > 2 interacting factors
  - Variation: only test all pairs for inputs that could influence each other
  - Random: pick subset of one of the strategies
- Each valid input at least once in a positive test case
- No more than one invalid input in a test case
- Heuristics cannot determine exact number of TCs required to test SUT effectively
- VV/ IV: any valid/ invalid value

Testing based on UCs

- System and Acceptance testing
- More attention given to high-priority use cases (e.g. Scripted)

Junit

- @Test annotation to mark test methods
    - Method name: whatIsBeingTested_descriptionOfTestInputs_expectedOutcome
- Rules: flexible addition or redefinition of behaviour of test methods
    - `public final TemporaryFolder, public final ExpectedException`
- Able to write automatic methods for pre/ post cleanups
- Able to test private methods