Testability, Issue Tracking, Monitoring, Bug counts, Traceability

1. Testability
   1. Testing: Not just “Finding bugs”
   2. Identifying other quality problems
      1. Design departures from requirements
      2. Usability problems (particular power users)
   3. Should focus on important bugs
   4. Give immediate feedback on rough quality
      1. Broad look at entire system
   5. Identify usability issues early thru test design
   6. Using different bug identifications skills than developers
   7. Helping identify process problems that cause or defer identification of bugs
   8. Effective reporting critical
2. One component of Risk management
   1. Designing for testability
3. Testability
   1. Specifications
   2. Modularity (small, focused goals of test)
   3. Separation of concerns
   4. Separating interfaces from implementations
   5. Writing test first
   6. Good documentation
   7. Mocking frameworks
   8. Assertions
   9. Logging/Profiling
   10. Hooks (spying on intermediate results, ability to look inject errors and observe result)
4. Testability: Ease of Testing
   1. Programs often have large amount of internal state
   2. Often can only access poor functionality/state, or must access it through a complex process
      1. GUI often expose very small fraction of total information
   3. It can be difficult to determine cause for errors
      1. For debuggers
      2. For developers during debugging
   4. Should consider how operation of the system will be revealed for to testers and developers
      1. Logging
      2. Hook
5. Testability
   1. Modularity
   2. Specifications
   3. Separation of interface from implementation
   4. Harnesses/Drivers
      1. Drive application programmatically (not via UI)
      2. May repeatedly test over series of input
      3. Record output
   5. Hooks
      1. Expose functionality to
         1. Set
         2. Get
   6. Assertions
   7. Allowing multiple instances on a client
   8. Allowing recording off/driving by config. File
   9. Stubs/Mocks/Fakes
      1. Return hard-wired results (e.g. fixed constants or simple instantiated data structures)
      2. Common uses
         1. Unit testing
         2. Prior to availability of code
      3. E.g. mock object driven by UI and logging request
6. Test Hooks
   1. Often require very small additional amounts of coding
      1. Frequently created by developers for assertions
   2. Tradeoffs for keeping in shipped code
   3. Often improve
      1. Debuggability
      2. Third-party integration
   4. Hooks for inspecting internal state
      1. Generally through privileged interfaces
      2. Often very helpful for debugging as well
   5. Processing event notification (e.g. with callbacks)
   6. Diagnostic routines (to check integrity, consistency, etc.)
   7. Methods for programmatically
      1. Driving application
         1. Independent of UI
         2. Often reused by UI
      2. Reading state of application
         1. For GUI application, much GUI feedback
         2. Without hooks, can’t tell if application did what supposed to
7. Example Test Hooks
   1. Enabling/disable logging of
      1. Connections
      2. Memory usage/structure
      3. Keystrokes/Commands
   2. Browsing of internal data structures via
      1. Custom UI
      2. Custom Languages
      3. Privileged interfaces
   3. Checking on intermediate state of algorithm
   4. Ask about step taken in running a particular case
   5. Custom command language
   6. Find out connection state
   7. Request amount of data sent
   8. Examine state of cache
   9. Report back database tables
   10. Report locks taken out
   11. Testing locks taken out
   12. Testing implementation state
   13. Checking invariants browsing
   14. Test points get/set specific state
   15. Deliberately cause rare error conditions
       1. Make network look congested
       2. Force network timeout
       3. Pretend disk is full
       4. Pretend file is corrupted
   16. Simulate early end to connection
   17. Simulate a very long delay connection
   18. Pretend tape drive experienced error
8. Recording logging and multilevel logging
9. Tips when collecting automated crash reports
   1. Leave in user-shipping code
   2. Ask user minimal info (e.g. what doing when crashed)
   3. Crash description info
      1. Version of software/JVM or OS version/3rd party software/line number file/method/error message/error id/description of user/email
   4. Format crash description so can easily research
      1. E.g. surround each part by recognizable delimiters
   5. Suggestion: collect reports via HTTP
   6. Consider automatically adding to STI db
      1. New case if new crash description
      2. Add to existing STI if repeat crash description
   7. See Spolsky, Get crash reports from users – automatically
10. Fakes, Mocks, and Stubs
    1. Often during software development, we must test code (A) not against a full implementation of other code (B)
       1. B may not yet be written
       2. B’s actual implementation may be non-deterministic, slow, difficult to get into the states we want, complex
       3. Seek to confine our debugging to A
    2. For these cases, we create a substitute version of the other object
    3. Fakes
       1. Just return trivial (e.g. identical or random) values
    4. Mocks
       1. Check context via assertions
       2. Help in spotting errors
       3. Log, etc.
    5. Stubs
       1. Fake or mock code for particular method
11. Key points of Use
    1. Integration tests of A and B, requiring C (not yet implemented)
    2. Unit testing of A (mocking up B and C)
    3. Testing UI
    4. Seeking to verify sequence of interactions
    5. Debugging (narrowing down areas of concern)
12. Useful tips
    1. For some tests, connect GUI to Fake/Mock rather than to system
       1. Log behavior
       2. Return dummy value (or semi-intelligent value)
       3. Return records of state of (fake) system
    2. Use tests harnesses to drive other components of system
    3. Test your hooks, mocks and fakes
13. JMock
    1. Mocked class matches same interface as class being mocked
    2. System will automatically create a “mock object matching interface
    3. Allows to declaratively set “expectations” regarding behaviors of mocked class
       1. Ordering of calls/expectations
       2. What to do when called
       3. Max/min number of times that called
14. Motivation
    1. Software development gives rise to tremendous amounts of details workflow complexity
       1. Defect reporting and tracking: description, status, repro
       2. Client change requests to be examined
       3. Notification of updates that are required
       4. Requests for peer review of artifacts
    2. Automated software can greatly facilitate the resolution of these issues
    3. Maintaining electronic information regarding these issues can provide important support for decision making