Best Practices

1. Key Best Practices
   1. Conscious, proactive risk scanning and management
   2. Accountable Positions
   3. Peer reviews
   4. Risk-driven incremental delivery
   5. Binary mini-milestones
   6. Continuous integration and smoke test
   7. Regular status updates and brief meetings
   8. Assertions
   9. Team-oriented version control practices
   10. Defect estimation
   11. Rigorous risk-driven testing (especially test creation before coding)
   12. Focused prototypes
   13. Time Estimation/boxing
   14. Scheduling
   15. Process improvement
   16. Change control mechanisms
2. Accountable Position
   1. Accountability 🡪 Taking responsibility
   2. Lack of accountability increases risks items fall through cracks
   3. Risk officer
   4. Project manager (FT)
      1. Scheduling
      2. Handling mgmt./user interfaces
      3. Assigning mini-milestones
      4. Triage
      5. Coordinating meetings
      6. Driving peer reviews
      7. Facilitation
   5. Development team and lead (FT)
   6. Continuous integration
      1. At least one build master
   7. Testing
      1. Testing lead (FT)
      2. Tester (FT)
   8. Triage team (ad hoc)
      1. Directed triage
      2. Monitoring bug counts
   9. Lead should also be communicational position
3. Importance of Risk management (conscious, proactive risk scanning and management)
   1. Overall: “Today’s risk is tomorrow’s problem”
   2. If you do not aggressively attack project risk, project risks will aggressively attack you --- Weinberg
   3. Good risk management is what allows a company to undertake higher-yield and risker projects
   4. Allows avoiding many problems
   5. Allows resilience to inevitable problems
      1. Greater lead time
      2. Know strategy
   6. Bounds uncertainty
   7. Helps prevent implicit transfers of risk
4. Top Ten list of software risk item
   1. Personnel shortfalls
   2. Unrealistic schedules and budgets
   3. Developing the wrong software functions
   4. Developing the wrong user interface
   5. Gold plating
   6. Continuing stream of requirements changes
   7. Shortfalls in extremely famished components
   8. Shortfalls in extremely performed tasks
   9. Real-time performance shortfalls
   10. Straining computer science capabilities
5. Proactive Handling of Risks: Steps
   1. Risk identification (what risks are there)
   2. Exposure analysis (how much do these risks matter?)
      1. Determine risk exposure, Prioritization
   3. Handling (how do we deal with the important risks?)
      1. Avoid (seek another strategy that avoids risk)
      2. Assume (accept)
      3. Control (contingency planning and proactive preparation): some mixture of
         1. Mitigate (do something up front)
         2. Contingency planning (ready to do something if risk materializes)
         3. Transfer (insurance)
      4. Monitoring (risk scanning)
6. Contingency planning
   1. Advantage: only expend resources if risk materializes
   2. Need to act quickly if risk materializes
      1. Spot risk as far ahead of time as possible
      2. Close risk scanning
   3. Good for short lead-time measures
   4. Problems
      1. Resources (time, money) used for another task
      2. Too disorganized in reaction
      3. Too expensive or infeasible to react effectively
      4. People rely on contingency to save them
7. Mitigation
   1. Resources allocated, spent up-front
      1. Plenty of time to arrange (can get cheaper)
      2. Resources always spent
   2. Disadvantages
      1. People argue as wasteful
         1. Seek dual use for investment
   3. Good for long-lead time needs (hiring staff, investigate alternative component/technology/vendor)
   4. Generally, combine with contingency planning
8. What is expected of you
   1. Risk officer
   2. Risk scanning
   3. Updated risk plan
   4. Risk prioritization
   5. Postmortems
9. Offensive Programming: Try to get broken program to fail early, hard (assertions)
   1. Asserts: Actually quit the program
   2. Fill memory allocated with illegal values
   3. Fill object with illegal data just before deletion
   4. Set buffers at end of heap, so that overwrites likely trigger page fault
   5. Setting default values to be illegal in enums
   6. We will talk about assertions and error handling later this week
10. Problem handling approaches
    1. Return codes
       1. Caught condition: Common expected event
       2. Special language support: NO
       3. Debug code: yes,
       4. User code: yes
    2. Monadic Optional values
       1. Common expected event
       2. Yes
       3. Yes
       4. Yes
    3. Exceptions
       1. Unexpected rare events
       2. Yes
       3. Yes
       4. Yes
    4. Assertions
       1. Logic errors, dev, mistakes
       2. Yes
       3. Yes
       4. No
11. Common Errors Caught
    1. Failure to adhere to contract
       1. Precondition
       2. Postconditon
       3. Invariants
    2. Logical oversights by programmers
       1. Property of data structure
       2. Internal consistency of algorithm
    3. Example Checks
       1. Results of efficient and brute force algorithms =
       2. Heap is not corrupting
       3. Reference Passed or in data structure is non-null
       4. Results of new code or old code are equal
       5. That a relationship assumed by the algorithm does hold
       6. That no duplicate entries check methods work
12. Assertion Use
    1. Typical: some or all only occur in debug code
       1. Performance implications
       2. Don’t want to present arcane info to user
       3. Need to avoid relying upon side-effects
    2. For most robust code, may put in logic to handle some assertion failures explicitly
       1. Not all bugs are fixed before shipping
    3. Tie ins with logging recommended
    4. Make a habit of checking preconditions, post-conditions, invariants
    5. Do not use assertions to handle unusual (but quite possible) situation – may not be present in user code
13. Spiral Model (Risk-driven incremental delivery)
    1. Communication
    2. Planning
       1. Estimation
       2. Scheduling
       3. Risk analysis
    3. Modeling
       1. Analysis
       2. Design
    4. Construction
       1. Code
       2. Test
    5. Deployment
       1. Delivery
       2. Feed back
    6. Back to a
14. Spiral Model
    1. Risk-driven approach combining elements of
       1. Systematic approached of waterfall model
       2. Iterative and nimble elements of prototyping
    2. Each cycle leads to
       1. Decreasing in risk
       2. Increasing product definition
       3. May not be development per se
          1. Prototypes, specifications, versions of software, etc
       4. Requires rigorous risk assessment
       5. Replan after each iteration
15. Continuous Integration (continuous integration and smoke test)
    1. Viewed as project “heartbeat” / “synch pulse
       1. Import for convergence, feedback/planning, cooperation
    2. Typical components
       1. One-stop compilation and linking
       2. “Smoke test”
          1. system-wide test of basic functionality
          2. Evolves over time
    3. Continuous integration argues for daily as lowest build frequency
16. Continuous Integration
    1. Building triggered by all developer checkin
    2. Can allow for pinpointing problems to checkins
17. Continuous Integration Advantages
    1. Cooperation: Greatly reduce integration headaches
       1. Reduced likelihood of merge conflicts
       2. Easier, less wasteful to fix if conflict occurs
       3. Allows bigger teams to function nimbly
    2. Quicker identification of problem. Check-in, bugs
    3. Helps identify state of project via smoke tests, availability of executable
    4. Improved estimation, flexibility for shipping
    5. Improves team and customer morale
    6. Helps force developers to fix bugs before continuing
18. One-step build
    1. Build script
    2. Clean
    3. Compile source code
    4. Integrate database
    5. Run test
    6. Run inspections
    7. Deploy software
19. One Solution
    1. Synch:
       1. Continuous build and smoke test
       2. Rules governing bug fixing vs. new devel
          1. Generally, developer should fix bugs on existing functionality before proceeding on to implement new functionality
       3. Stabilize
          1. Incremental and evolutionary delivery (“growing” software)
             1. Milestones, incremental development
          2. Mini-milestones
20. Synch and stabilize process
    1. CVS code (check out)
    2. Private Code under development (refresh against CVS)
    3. Unit tested code (refresh and diff against CVS)
    4. Unit tested code to be merged (merge code changes on Developer machine)
    5. Check-in Candidate Undergoing smoke test
    6. Code for check in waiting for check in window
    7. Update CVS code
    8. Official release from build
21. Clarifications
    1. Developers
       1. Work in private sandbox
       2. Do not necessarily check in work daily
    2. Developers primarily work against other people’s code from last stable build
       1. Generally will refresh every day
    3. Daily build is built from scratch
    4. Developers test code against recent checks before checking in own
    5. Gen. do not want to pass DB to customers
22. Maintaining the build
    1. Frequently maintaining daily build is a full-time job (possibly for serval people)
       1. Particularly tricky for large projects
    2. Code and make script development
    3. Requiring setting incentives/policing
       1. Requiring check in every few days
       2. Disincentive to break build (embarrassment, shouldering of build responsibility, etc.)
    4. Smoke test creation by testing department
    5. Require private smoke-testing before check-in
    6. Sometimes used as punishment for breaking build
23. Smoke Test (quick test)
    1. Breadth-oriented functionality testing
    2. Should be automated
       1. Cf continuous testing
    3. Purpose is not to discover new bugs, but to identify stability problems
       1. Want to be sure that new features won’t interfere with basic functionality relied on by other developers
24. Avoiding “breaking the build”
    1. Important that continuous integration be exception, not rule
       1. This includes smoke test
    2. Under pressure, devs, need cultural
    3. Under pressure, dev, need cultural pressure against “cutting corners” (unsafe check)
       1. Monetary fines
       2. Amusing penalties (goat horns)
       3. Taking over build
25. Barriers
    1. Hard to do early in project (start with rarer builds)
    2. Concerns that getting to point of smoke testing can require too much stub code
    3. Build of large projects demands heavy computational load
    4. Need to regularly update smoke test
    5. Careful oversight of build needed
    6. Red herring: Achieving daily integration requires considerable time (save much more time later)
26. Critical Piece: Feedback mechanism
    1. Text message
    2. Browser Plug-in
    3. Monitor
    4. Orb
    5. Sound
    6. Email
27. Duval build principles
    1. Commit code frequently
    2. Don’t commit broken code
    3. Fix broken builds immediately
    4. Write automated developer test
    5. All tests and inspections must pass
    6. Run private builds
    7. Avoid getting broken code
28. Testing for CI: Good Practices
    1. Automate unit test
       1. Such as NUnit or JUint
    2. Automate component tests
    3. Automate system test
    4. Automate functional tests
       1. Selenum for Web
       2. Abbot for GUI
    5. Categorize developer test
    6. Run faster tests first
    7. Write tests for defects
    8. Make component tests repeatable
29. DB Integration: Good Practices
    1. Automate database integration
    2. Use a local database sandbox
    3. Use a version control repository to share database asset
    4. Give developers the capability to modify the database
    5. Make the DBA part of the development team
30. Improve understanding of situation: directed triages
    1. Directed triage involves deliberately wading through bug reports to prioritize
    2. Typically full information lacking on project status because many bugs not examined for priority
    3. Directed triage can reduce uncertainty, enqueuer work
31. Element to be monitored (Binary mini-milestones)
    1. Task progress (mini milestones completed)
    2. Functionality output (LOC)
    3. Resource use (especially person hours)
    4. Build and release progress
    5. Quality (defect counts, reliability, performance, …)
    6. Morale
    7. Time passed
    8. Budget expended
32. Task Progress
    1. For mid-stage, binary completion of task (mini-milestones) is foremost metric
    2. Need shared understanding of meaning of reporting terms
       1. Implemented
       2. Unit and smoke tested
       3. Release nots updated
       4. Checked in
       5. Ready for QA
33. Fundamental reporting metrics
    1. Middle game:
       1. Completed work item
       2. Time spent working on items
       3. Prioritized defect counts
       4. Exit criteria status
    2. End game: defect stock, bugs fixed, found, incoming to triaged rate ratio, active bu time, bugs per developer, fault feedback ratio (FFR)
34. Expected (Rigorous risk-driven testing)
    1. Early provision of code to tester
    2. Design for testability
    3. Test driven develop
    4. Unit testing by developers against specifications
    5. Use of mocking
    6. One or more test matrices relating test to features (requirements)
    7. Both manual and automated testing
    8. System test (including through UI)
       1. Thought through test case
       2. At least node-level coverage of pages/screens
    9. Converge testing
35. Expected of us
    1. Early provision of code to testers
    2. Design for testability
    3. Test driven develop
    4. Unit testing by developers against specification
    5. One or more test matrices relating tests to features
    6. Both manual and automated testing
    7. System tests
       1. Thought through test cases
       2. At least node-level converge of pages/screens
    8. Coverage testing
36. Suggestion
    1. Debug not just defect, but process (coding, testing, etc.)
    2. Use a bug tracking system
    3. Build for testability from the start
       1. Modify code to add
          1. Step tracing/logging
          2. Testing hooks
    4. Automate most tests
    5. Check test errors
    6. Explicitly consider risk
    7. Create prioritized schedule
    8. Consider containerizing
    9. Strive for clear, balanced bug reports
    10. Consider
        1. Pair testing
        2. Buddy testing
    11. Build
        1. Require that code is tested against smoke test before check in
        2. Unit test during build
    12. Communicate requirement changes
    13. Talking to dev (especially if repeatedly failing with fixed)
37. Suggestion 1
    1. Plan test environment
    2. Reset state following test
       1. Containerization (via Docker) as strong asset
    3. Create prioritized schedule
    4. Create release criteria
    5. Derive some tests from requirement
       1. Design some system wide tests as soon as requirements are avail
38. Suggestion 2
    1. Beware skimping tests on late check in (often most buggy)
    2. Plan for non-func.tests
       1. Stress testing
       2. Load testing
       3. Performance tests
       4. Memory footprint
       5. Scaling
       6. Concurrency
    3. Isolate test and dev environ.
    4. Perform exploratory tests
    5. Try to automate regression tests
    6. Beware defect “circulation” among devs
    7. Use testing tools
       1. Investigate early
       2. Evaluate on prototype or other system
    8. Code freezing n days prior to deliverable
39. Reviews (Peer reviews)
    1. More cost-effective than testing
       1. IBM found 3.5 hors/error for inspection removal vs. 15-25 hours/error for testing
    2. Easily pay for themselves (quality is free)
    3. More flexible than testing
       1. Need not wait for executable code
          1. Can perform at all stages of software engineering process
          2. Can be done early in the development of a component
       2. Can assess communications issues (clarity, style, commenting, etc.)
40. Importance of early reviews
    1. Requirements
    2. Early artifacts have disproportionate impact on development process
    3. Marketing documents
    4. UI design
    5. Design
    6. Unit implementation
    7. Unit testing
41. Other benefits of peer reviews
    1. Person reviewing the artifact (clarify understanding, learn code tricks, stylistic ideas)
    2. Person whose artifact is being reviewed
       1. Improving technique, learn
    3. Broader culture
       1. Spread of knowledge about code base
       2. Spread of knowledge of standards, coding styles
       3. Code written with other people in mind
42. Guidelines for reviews
    1. Keep impersonal: focus on artifact, not people
    2. Keep review team small (3-7)
    3. Try to identify – but not solve – problems during review
    4. Limited meeting to no more than 2 hours
    5. Require advanced preparation for formal reviews
    6. Be sensitive to cultural and human components
    7. Prioritize focus for more major issues
43. Spectrum of Formality 1
    1. One classification
       1. Inspection
       2. Team review
       3. Walkthrough
       4. Pair programming
       5. Peer desk-check, passaround
       6. Ad hoc review
    2. Different terms used in different place
       1. Important thing is functional distinctions
44. Less Formal Reviews
    1. Peer desk check, pass around
       1. This involves preparation time for people but not meeting
    2. Walkthrough
       1. No preparation time – just see for the first time when being presented
    3. Team review
       1. Like inspection, but combined roles, author often leads meeting, inspector request to discuss sections of interest; may have more discussion on solutions
45. Inspection
    1. Plan inspection to address project and inspection objective
    2. Inspect upstream documents first
    3. Begin inspecting documents early in their lives
    4. Check against source and related documents
    5. Prepare and inspect at your organization’s optimum rates
    6. Focus on major defects
    7. Measure your benefits from inspections
    8. Emphasize defect prevention and process improvement
    9. Use serious, quantitative entry and exit condition
46. Size Tradeoffs
    1. More inspectors
       1. Catch more bugs (greater “synergy”)
          1. Despite pre-meeting preparation, some studies suggest large fraction of bugs found in meeting
       2. Make meeting harder to schedule
       3. Slow meeting progress
          1. Can stall programmer’s work
    2. Sometimes do servals parallel inspections rather than big shared inspection
47. A generic inspection Process
    1. Planning
    2. Planning for the meeting
       1. Ori enation meeting
       2. Preparation
       3. Review meeting
       4. Rework
       5. Another review needed
       6. Work unit close – out
48. Stage planning
    1. Participants review material on own before meeting
    2. Moderator assigned at this point
    3. Author contributes objective for inspection
    4. Based on historic data moderator estimates number of meetings required to do reviews of desired scope
    5. Moderator
       1. Invites participants
       2. Helps author prepare package of material for inspections
       3. Distributes package to participants several days ahead of time
49. Stage Overview
    1. Often a separate meeting
    2. Author more informally describes perspective on product
    3. Sometimes the inspection package is distributed during this meeting
    4. Sometimes skip if
       1. Participants already familiar with product
       2. Overview can be described in package
50. Stage Preparation
    1. Most preparation centers around inspection package
       1. The deliverable to be inspected
       2. Standards and requirements and specifications
       3. Typo list issue log
       4. Work aid to help identify defects
          1. E.g. common defects for this sort of deliverable
       5. Test documentation to verify this deliverable
51. Stages meeting 1
    1. Deliverables
       1. Inspection summary report
       2. Work product appraisal
       3. Information to communicate to mgmt.
       4. Issues log
       5. Indication of what changes are needed to complete inspection process
    2. May stop inspection if identified errors are to serious to make it worth it to continue
52. Meeting participant roles
    1. Author
    2. Moderator
    3. Reader
    4. Inspectors
    5. Recorder
    6. Typically, 3-4 participants
53. Stages rework
    1. Author addresses most items in issues log
    2. Sometimes issue log items get assigned to others
    3. Sometimes just log defects in defect control system to be followed up later
    4. Result
       1. Updated work product
       2. Annotated issue log indicating resolution for each item
54. Stage follow up
    1. Often with moderator as verifier
    2. Verifier confirms that changes have been successfully made
    3. Baselining of changed deliverable into SCCS
55. Stage Causal analysis
    1. This basically uses inspection process to improve
       1. The development process
       2. The inspection process
    2. Focus on process improvement and not on people
    3. Try to identify root cause of defects