Quality Assurance

Pragmatic Programmer Tip
Think about Your Work

Turn off the autopilot and take control. Constantly critique and appraise your work.

CSE 403, Spring 2007, Alverson
With material from Marty Stepp 403 lectures.
Readings

- Pragmatic Programmer: p 237-247 - Ruthless Testing

- Any junit/nunit/ruby test harness intro/tutorial:
  Examples:
  - http://open.ncsu.edu/se/tutorials/junit/
  - http://www.15seconds.com/issue/040922.htm
Outline

• Quality from the get-go

• Test
  o What makes a good tester
  o Types of testing

• Bugs
Quality Assurance

Things we can do to ensure we produce a high quality (low defect) product

Two main approaches to QA:
- Process: Build in quality from the start
- Test: Add quality through removing bugs
Ways to build in quality from the start

•

•
Ways to build in quality from the start

- Good design and planning
- Coding style guides
- Code reviews/walkthroughs
- Pair programming
- Constant attention to resources: time, memory, through measurement
From Apple’s C++ Style Guide:

- **Type names** – begin with a capital letter. ie. Book

- **Member names** - begin with an *f*, for "field." Member function names need only begin with a capital letter. ie.: fVisible, Draw.

- **Global names** (*including* static members of classes) - begin with a *g*. ie: gApplication, TGame::gPlayingField.

- **Local and parameter names** - begin with a word whose initial letter is lower case. ie.: i, thePort, aRegion.

- **Constant Names** - begin with a *k*. ie: kSaveDialogResID.

- **Multiple-word names** - the first word should follow the convention for the type of the name, and subsequent words should immediately follow, with the first letter of each word capitalized. ie. TSortedListclass
Interesting fact

Up to 4x the normal number of defects are reported for released products that were developed under excessive schedule pressure (Jones, 94)
Test – the most common QA practice

Two components of test:

**Verify:** “Did we build the system right?”

**Validate:** “Did we build the right system?”
What makes a good tester?

- Analytical
  - Ask the right questions
  - Develop experiments to get answers

- Methodical
  - Follow experimental procedures precisely
  - Document observed behaviors, their precursors and environment

- Brutally honest
  - You can’t argue with the data
How do test engineers fail?

- Desire to “make it work”
  - Impartial judge, not “handyman”
- Trust in opinion or expertise
  - Trust no one – the truth (data) is in there
- Failure to follow defined test procedure
  - How did we get here?
- Failure to document the data
- Failure to believe the data
Some testing jargon

Black box testing
Treats the system as atomic
Best simulates the customer experience

White box testing
Examines the system internals
Trace data flow directly (ie, in the debugger)
Bug report contains more detail on source of defect
May obscure timing problems (race conditions)
In black box, the tests are usually intended to cover the space of behavior.

Often developer driven

In white box, the tests are usually intended to cover the space of parts of the program.

Often tester driven
General types of testing

- Functional (include boundaries)
- Coverage/path
- Performance
- Security
- Stress/load
- Resource exhaustion
- Reliability/availability
- Usability
- Integration
- Validation
- User interface
- Web
- Environmental – heat/vibration
Boundary testing

- **boundary testing**: test inputs starting from known good values and progressing through reasonable but invalid to known extreme and invalid

  - Is this black or white box?

  - Imagine we are testing a Date class with a `getDaysInMonth(month, year)` method. What are some important conditions and good boundary tests for this method?
Boundary testing

- `getDaysInMonth(month, year)` method. What are some important conditions and good boundary tests for this method?
  - leap years
  - decimal, negative numbers
  - MaxInt, MinInt
  - Feb, September, …
Date class

• boundary tests
  
  getDaysInMonth(month, year)

  o

- boundary testing is *black-box*
  some ideas:
  - check for leap years (every 4th yr, no 100s, yes 400s)
  - try years such as: even 100s, 101s, 4s, 5s
  - try months such as: June, July, Feb, invalid values
Unit test tools to capture tests

- Capture your boundary tests in a test harness (jig) for easy, automated, running

- JUnit (java) and NUnit (c#), popular harnesses, plugin to eclipse – *Thursday’s section*

http://www.nunit.org/
http://www.junit.org/
Attributes of good unit tests

- Well-defined inputs and outputs
  - Consider environment as inputs
  - Consider ‘side effects’ as outputs

- Clearly defined initial conditions
  - Clearly described expected behavior

- Specific – small granularity provides greater precision in analysis

- Test must be at least as verifiable as feature
Coverage testing

- **coverage testing**: an attempt to use test input that will pass once over each path in the code
  
  - Is this black or white box?
  
  - What would constitute coverage testing for getDaysInMonth(month, year)?
Date class

coverage tests

getDaysInMonth(month, year)?

helps to see the code... but assume it deals with different days in the different months, and leap years

- coverage testing is white box

some ideas:
- error input: year < 1, month < 1, month > 12
- one month from [1, 3, 5, 7, 10, 12]
- one month from [4, 6, 9, 11]
- month 2
  - in a leap year, not in a leap year
Coverage analysis

- shows where you have holes in your test suite
- guides productive test development: try to get the most coverage with the least effort
- should you aim for 100% coverage?

Which line shows productive test development?
Coverage tools

Plugins for eclipse that work with nunit, junit.

Google: nunit [junit] code coverage
Integration testing

Shows that the major subsystems that make up the project work well together.

- **big bang**: no stubs; do unit testing, then throw all parts together
- **bottom-up**: integrate upward into double, triple, quadruple module test
- **top-down**: test top layer (UI) first, then add layers to replace underlying stubs

- **big bang**: + faster (if everything works) - can be costly, error-prone
- **bottom-up**: + fewer stubs needed - tests UI last; UI is important!
- **top-down**: + focuses on user experience - needs many stubs
GUI testing

- testing a product that uses a GUI, to ensure it meets its written specifications

- difficulties:
  - many operations to test (ie. MS Wordpad has 325)
  - order of testing matters
  - regression testing is hard given GUI evolution
  - need to test on an array of browsers (web apps)
Approaches to GUI testing

• **automated UI testing**
  o scripts that use your app and look for failures
  o a black-box system test
  o Selenium: http://www.openqa.org/selenium/

• **manual tests**
  o human beings click through predetermined paths
  o need to write down the specific tests each time
  o http://members.tripod.com/~bazman/checklist.html

• **ad-hoc tests**
  o human beings are "turned loose" on the app to see if they can break it
Web app compatibility testing

• motivation
  o ensure that your web app is compatible with various browsers, platforms, etc.
  o ensure that your app's HTML code complies with web standards
  o ensure that you have no broken links or other HTML errors

• W3C HTML validator:
  http://validator.w3.org/

• W3C link checker:
  http://validator.w3.org/checklink

(let’s try it out on the 403 home page)
Load testing

creating demand on a system or device and measuring its response

- How many hits should the system be able to handle?
- What should be its performance under these circumstances?
- Will the system withstand abnormal load (stress testing)?

**tools**

Thoughts on test automation?

- Advantages

- Any limitations?
Pragmatic Programmer Tips

Test early, test often, test automatically

Coding ain’t done ‘til all the tests run

Find bugs once
BUGS!

Managing Bugs
What is a bug?

- Formally, a “software defect”
- System fails to perform to spec
- System causes something else to fail
- System functions, but does not satisfy usability criteria

If the system works to spec and someone wants it changed, that’s a feature request
What makes a good bug report?

Include:

- **Reproducible steps** – how did you cause the failure?
- **Observed result** – what did it do?
- **Expected result** – what should it have done?
- **Any collateral information**: return values/output,…
- **Environment**
  - OS version, env variables, compiler flags, …
  - Test platforms must be reproducible
  - “It doesn’t do it on my machine”
Defect tracking tools help management of data entry

Before reporting a bug, please read the bug writing guidelines, please look at the list of most frequently reported bugs, and please search for the bug.

**Bugzilla**

Enter Bug: TestProduct

Reported: alverson@cs.washington.edu
Version: other
Platform: PC
Initial State: NEW
Assign To: webmaint@cs.washington.edu
URL: http://

Product: TestProduct
Component: TestComponent
Priority: P1
OS: Windows
Severity: blocker

Tune in on Thursday for more about Bugzilla
And aid data evolution and retrieval

Value of a bug database is immeasurable, especially as the size of the project or number of customers grows:

• Don’t lose bugs
• Can pass them around easily
• Can prioritize
• Justify resources
• Justify your work
• Help indicate when the product ready for release (how?)
• Enables all sorts of cool metrics
Example: metrics showing quality

**Figure 16-1** Example of an “open defects” graph. Making this graph public emphasizes that controlling defects is a high priority and helps to keep potential quality problems under control.
More metrics – ready to release?

Open defects by priority (p1=high, p5=low)
Is this a reflection on the dev team?

Open defects by module

- **uml** [975]
- **java** [735]
- **core** [658]
- **web** [453]
- **editor** [385]
- **j2ee** [381]
- **xml** [306]
Classifying bugs

- **CRITICAL**: Prevents the customer from meeting their normal production schedules
- **URGENT**: Limits the customer's ability to meet their normal production schedules
- **MAJOR**: Does not interfere with the customer's normal production schedules
- **MINOR**: Little or no impact on users and can be bypassed easily
- **DES-HI/MED/LOW**: Feature requests
Another classification

- **Customer Facing**
  - **Severity**
    - Sev 1: crash, hang, data loss
    - Sev 2: blocks feature, no workaround
    - Sev 3: blocks feature, workaround available
    - Sev 4: trivial (e.g. cosmetic)

- **Internal**
  - **Priority**
    - Pri 1: Fix immediately
    - Pri 2: Fix before next release outside team
    - Pri 3: Fix before ship
    - Pri 4: Fix if nothing better to do 😊
A bug’s life (simplistic)

Bug activated

Triage Fix?

YES

Defect fixed - bug resolved
Fixed

NO

Won’t Fix
Not Repro
By Design
Postponed

Regression testing

Fixed?

YES

Bug closed

NO
Regression Testing

1. Ensure that the bug you just fixed, doesn’t reappear later though other mods
   - Add a test case to your suite for it!

2. Ensure that the fix you just added doesn’t break things working previously
   - Rerun the test suite before checkin
When can I ship?

Tune in next week!