# CSE 403

Testing, part II (Quality Assurance),
bugs, and code reviews, part 2

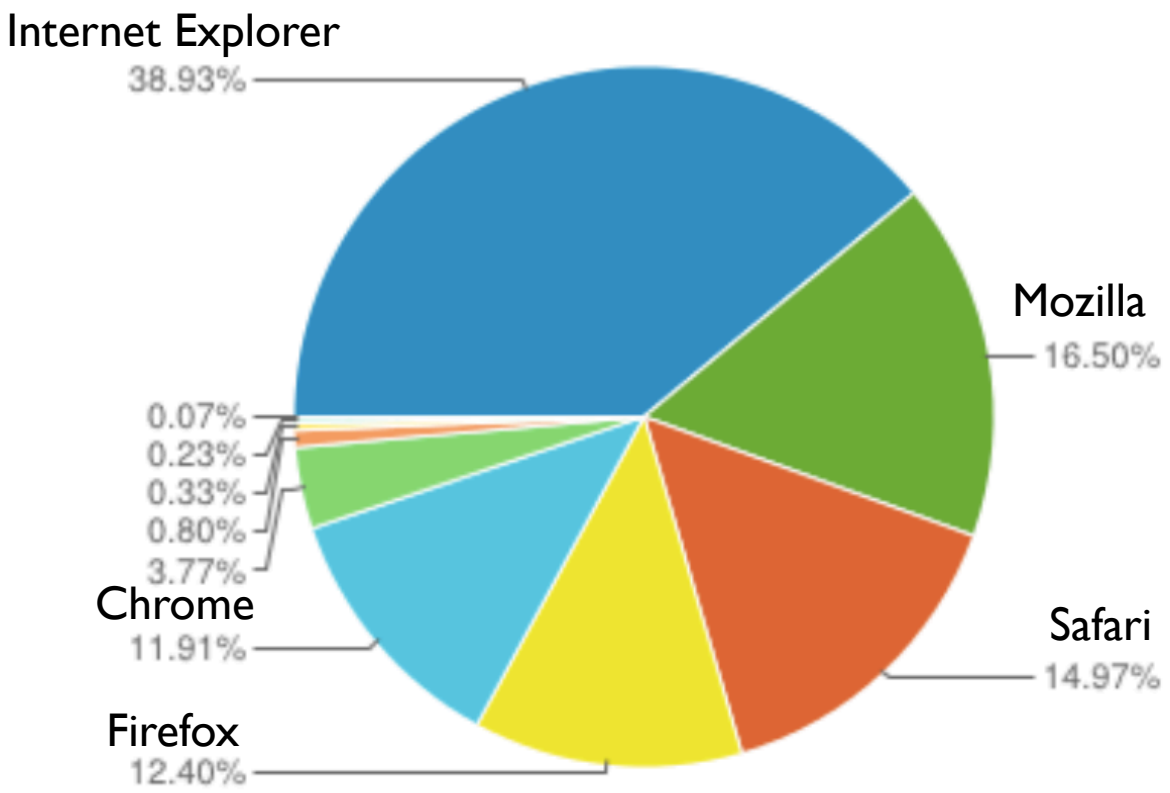
# Announcements

- Deliverables for 11/19 release (due 11:59PM)
  - Documents up to date: req, arch, design, schedule
  - Release notes
  - Status report for the **release** (not due until 11/20) -- content should be part of 11/21 presentation
- Presentations on 11/21: 7 minutes, same logistics as before. Deposit presentations (pdf) in dropbox by 11:59PM on 11/20
- Final release due on 12/03

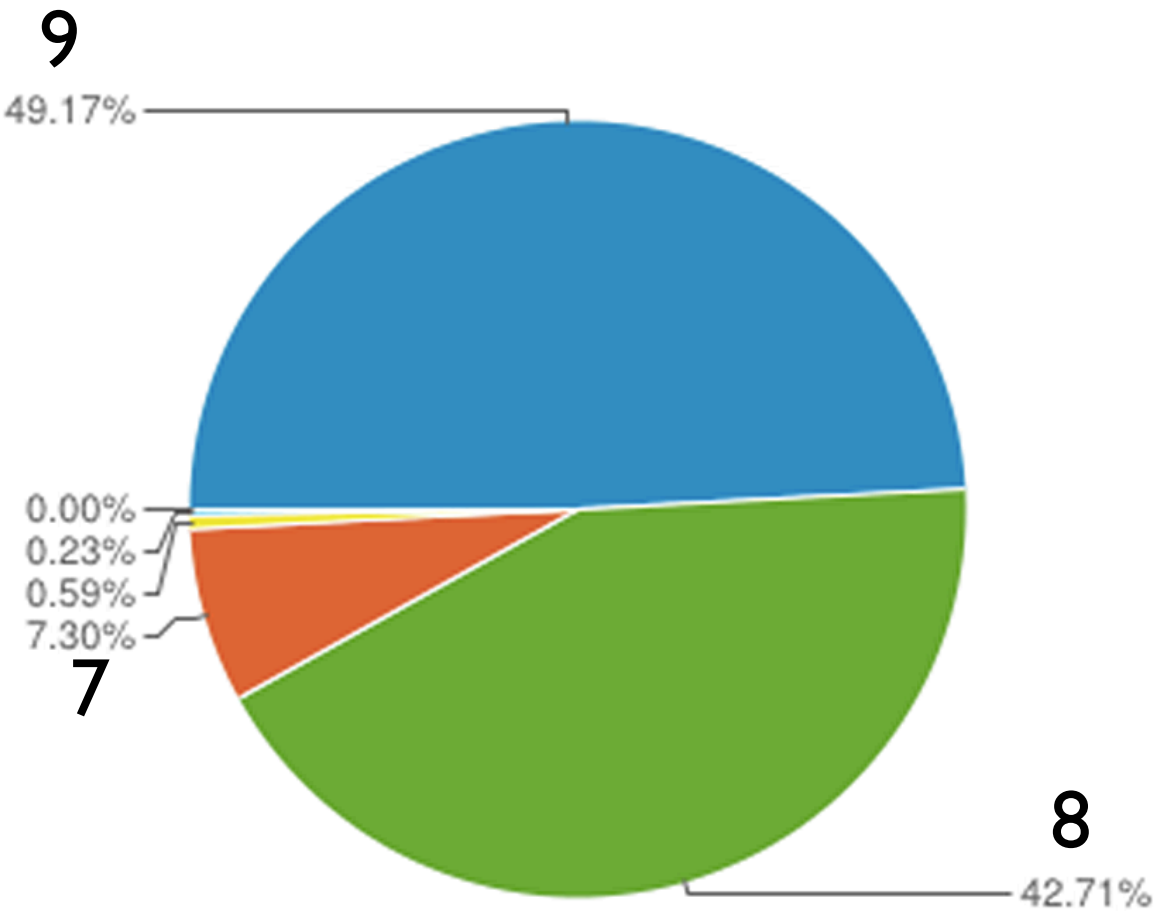
# Project Presentations, 11/21 (7 minutes)

- Operations Review by manager or project manager-- similar to weekly status but for the release cycle
  - What did you say you were going to do? What did you do? (1 minute)
  - What are you going to do in the next cycle? (1 minute)
  - What are the issues? (1 minute)
  - **What's your current status on platforms and browsers?**
  - Report on a metric -- or if not in place, scale from 1(disaster) - 10 (exceeding) and why (1 minute)
- Demo (1 minute)
- Questions and Answers (1 minute)

# Browser Usage



# Internet Explorer Usage



# Final release

- Project deliverable
  - Quantity: How much did you deliver?
  - Quality: How good was it?  (Design/UI/UX matters, bugs, usability, code)
  - Achievement of vision
- Software engineering principles
  - See wrap up and Joel on Software article
  - Answer all the questions in the wrap up (and more)

# Transparency (Compensation)
(tip of the day)

- You (nearly) have a job offer
- You are negotiating for compensation (mostly salary but...)
- How do you know if it's a good deal?

# CSE 403

Testing, part II (Quality Assurance),
bugs, and code reviews, part 2

# Types of testing

- Unit testing
- Regression testing (different definition that David talked about)
- Coverage testing
- Monkey testing
- Code review (is this testing?)
- Performance testing, load testing, scalability testing (maybe later)

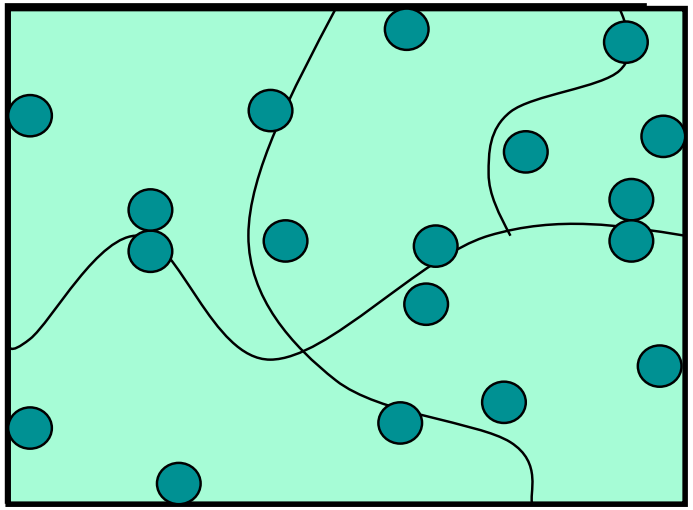
# Unit testing drill down

# Test cases

- How many?
- Can we test all cases?
- What tests do we want to run?
- How do we enumerate them?
- Can we partition the space?
- Tests can only reveal the presence of bugs/defects/problems -- not verify the correctness or absence of bugs

# Partition the input space to understand test cases you should write

- In theory, you only need to select one test in each partition
- In practice, this is hard -- problem specific
- I don't know how to do this!
- Edge cases are often the problem



# What's on the boundary? (i.e. what's an edge case?)

- Off by one errors
- Smallest, largest cases
- Zero, null, empty containers
- Null pointers
- Overflows in arithmetic
- Type errors
- Aliasing
- Exceeding maximum (buffer overflow)
- Circularity

# Test harness

- We'll use pytest
- You'll use the Django test execution framework?
- Can we really test the command line parsing?
- We'll test the triangle function code
  - Assuming we want to expose this interface

# Type errors (in Python)

- triangle function expects 3 integer arguments
- Types aren't explicitly declared
- Who should catch these errors?
- What do the following calls return?
  - triangle('a', 'a', 'a')
  - triangle ('z', 'z', 'a')
  - triangle('a', 'z' 'z')
  - triangle('a', 0, 2)
  - triangle ('a', 1, 2)

```
import sys

def triangle(x, y, z):
    if(x<=0 or y<=0 or z<=0
or (x+y<=z or x+z<=y or y
+z<=x)):
        print 'false'
    elif(x == y == z):
        print 'equilateral'
    elif(x==y or y==z or
x==z):
        print 'isoscele'
    else:
        print 'scalene'

if __name__ == "__main__":
    main()
```

# When should you write unit tests?

- Part of the design process -- helps you define the interface?
- After the design process
- After implementing
- After integration -- make sure there aren't many gaps
- After a bug is found

# Regression testing

- A regression suite is a collection of unit tests that you run to test the sanity of the system
- Typically automated
- A regression test is a unit test in the regression suite
- Slightly different than David's definition -- a unit test after a bug a found
- Run regularly (daily) and irregularly (before release, after big code changes before checking in)
- Gives confidence that things are still working or you didn't break anything
- You can't aggressively "re-factor" unless you have a good regression suite*
- *...if your refactoring changes the interfaces against which you run your test, this doesn't help -- your tests are broken

# Monkey testing

- Slang for functional/feature testing?
- Manually testing the application by clicking on buttons, filling out fields, observing results
- Walk through all the features
  - New
  - Old (regression)
- On all the platforms (**browsers**, backends, databases, etc.)
- Do it every release?
- Can this be automated?

# Code coverage

- Collection of tests so that every line of executable code gets executed
- Sanity test
- Especially important in non-statically checked/ typed languages(?)
- Exception paths are particularly interesting
- What are the success/failure conditions?
  - Uncaught exceptions, memory leaks, type errors(?)

# Bugs

# What are bugs?

- Algorithm, coding, type, logic errors for sure
- Incomplete/Missing functionality
- Mismatches between requirements and implementation
- Failure to meet strategic goals
- Ease of use difficulties
- User interface errors

# Categorizing bugs

- Severity : How "bad" is it?
- Priority: How important is it that we fix it (and when)
- Subsystem: Which module is the bug located, if known)
- Release: Which release(s) have the bug?
- Ownership: Who is responsible for fixing it?
- Status: Open, Assigned, Fixed, Closed
- Platforms: Operating systems, platform versions, browsers

# Writing bug reports

- Use a bug tracking tool
- Identify all the category information
- Written explanation of:
  - What the bug is
  - How to trigger it
  - Screen shots if available or applicable
  - Data to support trigger if appropriate
- Search for "good bug reports"; lots of good advice on the web

# Severity and Priority

- Severity: Critical, Major, Average, Minor, Exception
- Priority: Resolve Immediately, Give High Attention, Normal, Queue, Low Priority
- Severity X Priority ->25 choices!
- What do these mean?
- Argument in every company I've worked for
- Simplified from 2 dimensions (severity, priority) to 1: "show stopper," high, medium, low
- Overloaded terminology: Single dimension system called "priority"

# Reasoning about bugs

- Destroys data
- Crashes/Hangs system
- Feature not completed(?)
- User can't get task completed
- Inconsistent/confusing presentation
- Visual/presentation issues
- Annoyances
- CEO demands(?)

Red == show stopper
Orange == high
Yellow == medium
Green == low

# Bug tracking

- Tools (github, trac, bugzilla)
- Process
  - Who files them? (Mostly dev team and PMs but others too)
  - Project manager, product manager, or engineering manager oversees
  - Engineering manager and product manger negotiate
  - “Kill bugs” day
  - Features included? Wish lists?

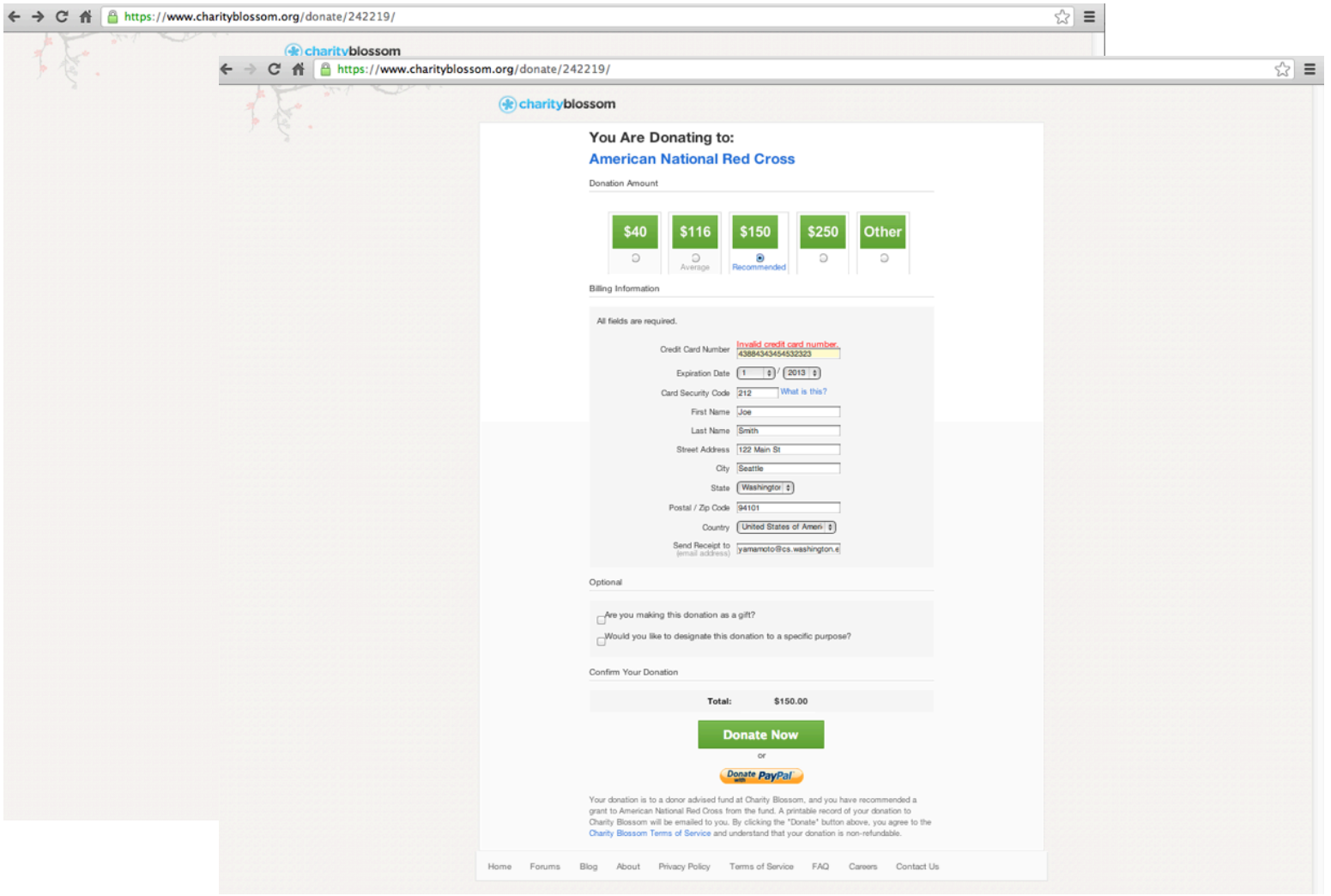
# Example bug report

- Priority: Show stopper
- Subsystem:  Donation processing
- Release:  20121031
- Ownership:  yamamoto
- Status: Assigned
- Platforms: All browsers

# Bug report example

- What: No donations can be completed because credit card field is never recognized as valid, even when a correct credit card number is entered
- How: Go to any donation page (e.g. https://www.charityblossom.org/donate/242219/) and correctly type in valid values for all fields and hit "Donate Now"
- System always reports, "Invalid credit card number"

# Screen shots



# Cost of bugs

# Cost of bugs

Bug in Knight Capital Group/NASDAQ costs $440 million resulting from glitch in facebook IPO.


facebook

# Cost of bugs

In 1988, Robert Tappen Morris exploits buffer overflow in gets() library call to cause $100 million in damage

# Cost of bugs

In 1999, Mars Climate Orbiter lost in space.
Cost: $665 million.
Bug: Teams used both metric units and English units.

# Cost of bugs

Y2K bug. Not enough digits to store the year, causing computers to think 2000 was 1900. Cost: $297 billion

# Cost of bugs

Therac-25 delivers100 times intended radiation (according to Wikipedia) and 6 deaths

# Aside: Stuxnet

- Exploitation of bug in Microsoft Windows to propagate a worm
- Target is a Siemens control system
- Initial infection through USB thumb drive
- Damaged Iranian Uranium Enrichment infrastructure
- Compromised digital certificates
- Supposedly a very sophisticated test system replicating target infrastructure
- Who writes this software? Should the bugs be patched? Bug vs. feature?
- All very current and murky
- Caveat: All of this is based on information from Wikipedia and the New York Times. Trust at your own risk.

# Elevator design bug

- Walk in
- Turn around
- Choose the floor you are going

Question: Which side has the elevator buttons?

# Elevator design bug

It's not on the left...

but on the left in the right elevator.

It's on the right...

in the left elevator...

It's not intuitive, most can't remember this, and they get it wrong about 50% of the time.

# Evevator design bug

- Don't know if it was designed this way (if it was, what was the reason?)
- Maybe an oversight (a bug)
- When was the problem discovered?
- Fix is to put buttons on the right (or left) in both elevators -- usage consistency
- Cost to fix it pretty cheap at design time
- Probably pretty expensive once it was installed
- Ongoing cost (penalty) for not fixing the bug
- Is there a positive unintended consequence?

# Finding bugs

- During requirements
- During development
- During testing
- In the field (Really?)
- Examining the logs

# When should you look for/find bugs?

- Conventional wisdom says sooner is better
- Conventional wisdom says sooner is cheaper
- But in current times, is this still true?
- Depends on the application...

# Code reviews

# Code reviews

**Code review** is systematic examination (often known as peer review) of computer source code.

My addition: It's (usually) a human process

# Code reviews: why?

- Find bugs
- Evaluate architecture and design
- Share knowledge
- Share best practices
- Propagate “standards”

# How and when?

- Formal
- Informal: Pair programming, side by side debugging
- Whenever you do a checkin
- When you fix a bug (especially when close to a release -- e.g. after “code freeze”)
- After you compile, before you execute(?)

# Formal

- From the "dinosaur age" of programming?
- Line by line review, often of large bodies of code
- 1 minute/line, plus 1minute/line prior to code review
- Reader, author of the code, recorder, 2 reviewers
- Reviewing 60 lines of code costs 10 people-hours
- Useful?

# Reviewing triangle

```
import sys

def triangle(x, y, z):
    if(x<=0 or y<=0 or z<=0 or (x+y<=z or x+z<=y or y+z<=x)):
        print 'false'
    elif(x == y == z):
        print 'equilateral'
    elif(x==y or y==z or x==z):
        print 'isoscele'
    else:
        print 'scalene'

if __name__ == "__main__":
    main()
```

- Are degenerate triangles valid?
- Is 0 a valid length?
- "isosceles" vs. "isoscele"?
- type checking -- caller or callee?
- print vs. return value?
- comments?
- coding standards?
- sort x, y, z into a, b, c --> only need to check if (a < 0 and a + b < c) for invalid triangles

# Reviewing triangle

```
def main():
    print 'Line segments:',str(sys.argv[1:])
    x = atoi(sys.argv[1])
    y = atoi(sys.argv[2])
    z = atoi(sys.argv[3])
    triangle(x, y, z)


if __name__ == "__main__":
    main()
```

# Type check in the callee

## Original

```
import sys

def triangle(x, y, z):
    if(x<=0 or y<=0 or z<=0 or (x
+y<=z or x+z<=y or y+z<=x)):
        print 'false'
    elif(x == y == z):
        print 'equilateral'
    elif(x==y or y==z or x==z):
        print 'isoscele'
    else:
        print 'scalene'

if __name__ == "__main__":
    main()
```

## Proposed revision

```
def triangle(x, y, z):
    if((isinstance(x,int) or isinstance(x,float))
and (isinstance(y,int) or isinstance(y,float)) and
(isinstance(z,int) or isinstance(z,float))):
        if(x<=0 or y<=0 or z<=0 or (x+y<=z or x
+z<=y or y+z<=x)):
            print 'false'
        elif(x == y == z):
            print 'equilateral'
        elif(x==y or y==z or x==z):
            print 'isoscele'
        else:
            print 'scalene'
    else:
        print 'false'
```

# Type check in the caller

## Original

```
def main():
    print 'Line
segments:',str(sys.argv[1:])
    x = atoi(sys.argv[1])
    y = atoi(sys.argv[2])
    z = atoi(sys.argv[3])
    triangle(x, y, z)


if __name__ == "__main__":
    main()
```

## Proposed revision

```
def main():
    if len( sys.argv ) != 4:
        print "must call with 3 line segment
lengths"
        sys.exit(1)
    # will raise ValueError if inputs are not
numeric
    segments = map( float, sys.argv[1:] )

    triangle(*segments)


if __name__ == "__main__":
    main()
```

# How important are code reviews?

- Formal -- maybe, maybe not
- Peer-to-peer, informal -- important
- Knowledge transfer -- useful but infrequent
- Rigid process ("code review before every bug fix") -- sometimes and maybe
- As usual, "it depends" -- YMMV

# Take aways

- Testing (unit, regression, coverage, monkey)
- Specifying unit test cases is hard
- But there are still more (load, integration, system, response, scalability ...)
- Bugs:  what are they?, categorization, bug tracking systems/processes/policies
- Code reviews