Design Reviews and Code Reviews

*

Design/Code Reviews

- Design review: meeting where authors present design with goal of quality by benefiting from the experience of the people attending the meeting
- Code review: held once the design has been implemented
  - In the Agile/Scrum context, since design and implementation occur together, they might be held every few iterations

Design/Code Reviews

- Shalloway*: formal design and code reviews often too late in process to make big impact
- Recommends instead have earlier, smaller meetings: "approach reviews".
  - A few senior developers assist team in coming up with an approach to solve the problem
  - Group brainstorms about different approaches
- If do a formal design review, suggests 1st hold a "mini-design review" to prepare

*Alan Shalloway, *Agile Design and Code Reviews*, 2002, www.netobjectives.com/download/designreviews.pdf

Code Reviews
Can Check Comments too

- Challenge: keeping comments consistent with changes, refactoring
- Code review is one place to ensure comments make sense
- Advice on comments:
  - Document what is not obvious from code
  - Raise level of abstraction
  - Explain why did something

Good Meetings: SAMOSAS

(Photo by K.S. Poddar. Used by permission under CC-BY-SA-2.0.)

- Start and stop meeting promptly
- Agenda created in advance; no agenda, no meeting
- Minutes recorded so everyone can recall results
- One speaker at a time; no interrupting talker
- Send material in advance, since reading is faster
- Action items at end of meeting, so know what each should do as a result of the meeting
- Set the date and time of the next meeting

Minutes and action items record results of meeting, start next meeting by reviewing action items

*

Reviews and Agile?

- Pivotal Labs – pair programming makes review superfluous
- GitHub – *pull requests* replace code reviews
  - 1 developer requests her code to be integrated with code base
  - Rest of team see request and determines how affect their code
  - Any concern leads to online discussion
  - Many pull requests/day
    => many minireviews/day

Finishing Coverage
and
Dealing with Legacy Code

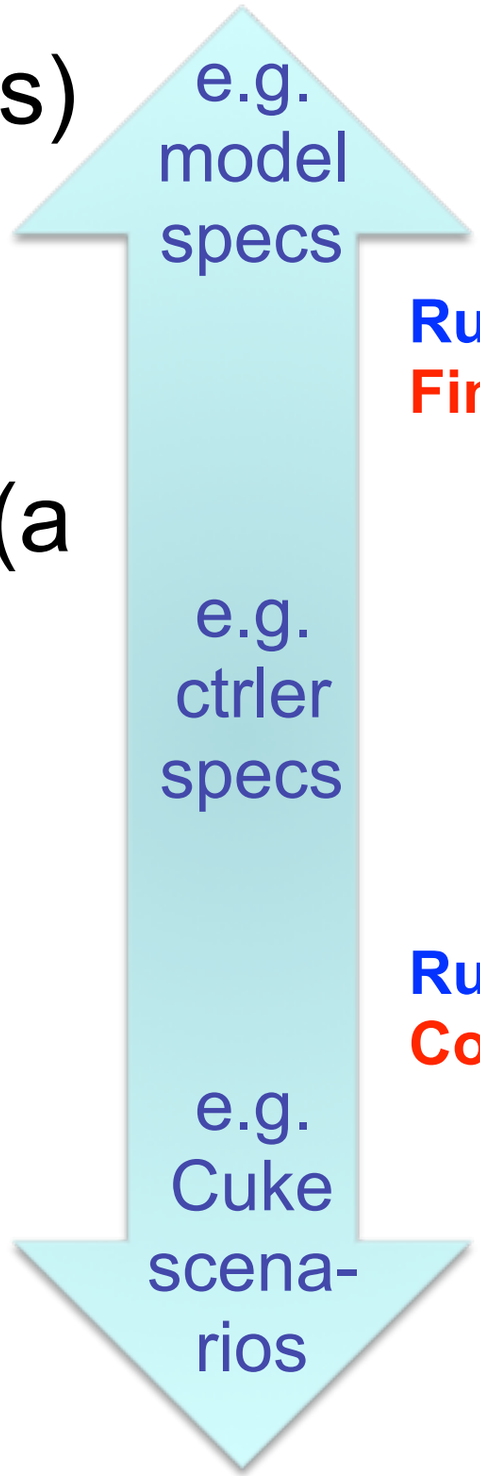
Measuring Coverage—Basics

```
class MyClass
 def foo(x,y,z)
  if x
   if (y && z) then bar(0) end
  else
   bar(1)
  end
 end
 def bar(x) ; @w = x ; end
end
```

- S0: every method called
- S1: every method *from every call site*
- C0: every statement
  - Ruby SimpleCov gem
- C1: every branch in both directions
- C1+decision coverage: every *subexpression* in conditional
- C2: every path (difficult, and disagreement on how valuable)

What kinds of tests?

- Unit (one method/class)
- Functional or module (a few methods/classes)
- Integration/system



High coverage

Runs fast
Fine resolution

Many mocks;
Doesn't test interfaces

Few mocks;
tests interfaces

Runs slow
Coarse resolution

Low coverage

Other testing terms you may hear

- Mutation testing: if introduce deliberate error in code, does some test break?
- Fuzz testing: 10,000 monkeys throw random input at your code
  - Find ~20% MS bugs, crash ~25% Unix utilities
  - *Tests app the way it <u>wasn't</u> meant to be used*
- DU-coverage: is every pair <define x/use x> executed?
- Black-box vs. white-box/glass-box

Going to extremes

- "I kicked the tires, it works"
- "Don't ship until 100% covered & green"
- TRUTH: use coverage to identify untested or undertested parts of code
- "Focus on unit tests, they're more thorough"
- "Focus on integration tests, they're more realistic"
- TRUTH: each finds bugs the other misses

TDD vs. Conventional debugging

Conventional	TDD
Write 10s of lines, run, hit bug: break out debugger	Write a few lines, with test first; know immediately if broken
Insert printf's to print variables while running repeatedly	Test short pieces of code using expectations
Stop in debugger, tweak/set variables to control code path	Use mocks and stubs to control code path
Gah! I thought for sure I fixed it, now have to do this all again	Re-run test automatically

- Lesson 1: TDD uses same skills & techniques as conventional debugging—but more productive (FIRST)

- Lesson 2: writing tests *before* code takes more time up-front, but often less time overall

TDD Summary

- Red – Green – Refactor, and always have working code
- Test *one* behavior at a time, using seams
- Use it "placeholders" or pending to note tests you know you'll need
- Read & understand coverage reports
- "Defense in depth": don't rely too heavily on any *one* kind of test