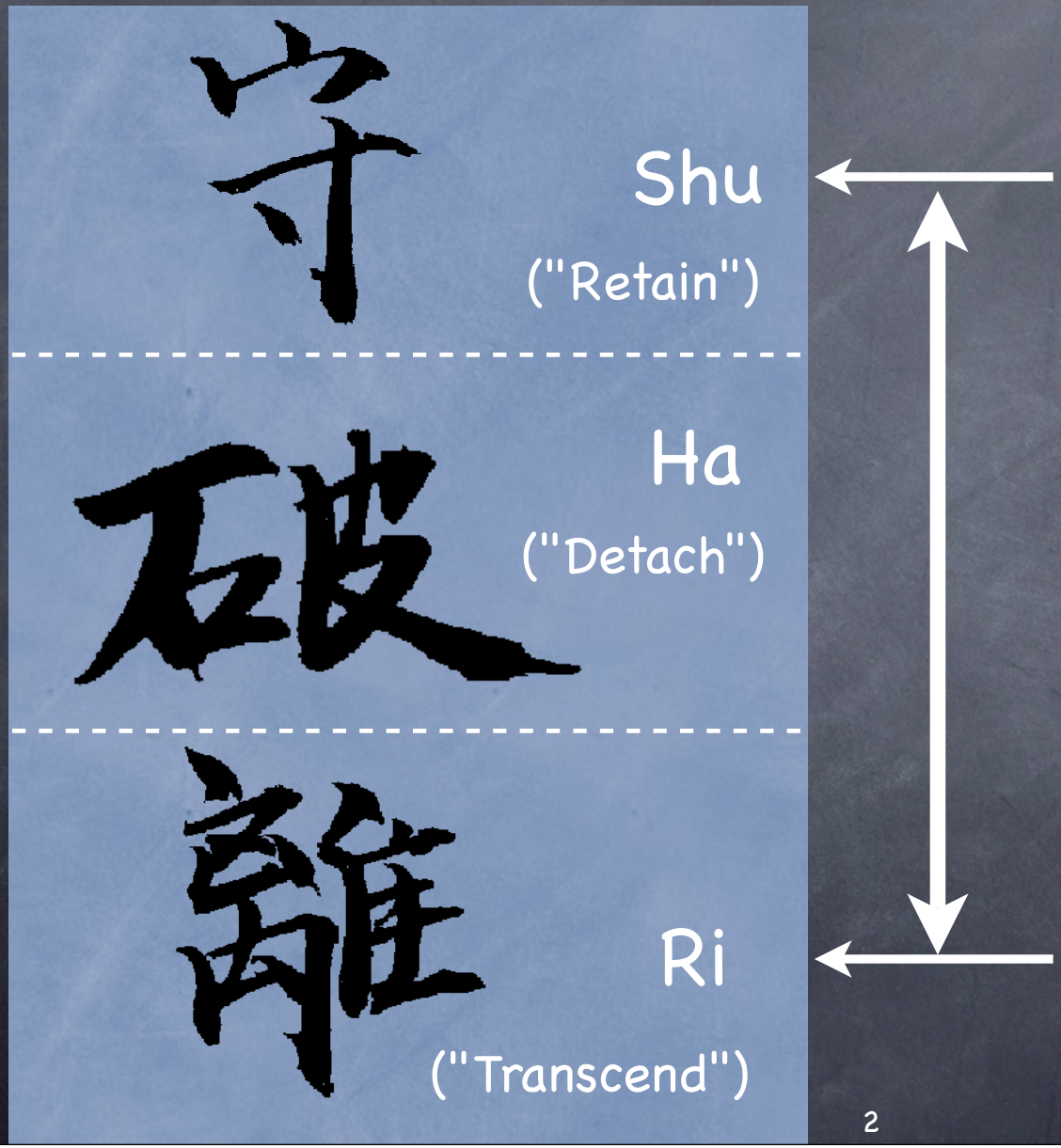
# Code Reviews for Fun and Profit

http://www.aleax.it/osc08_crev.pdf

# Audience levels for this talk






Google

# Contents of this talk

- code reviews: why don't we do enough?
- "Fagan inspections" vs lightweight CRs
- "too-light" CRs & their anti-patterns
- some "social aspects" of CRs
- what to check in CRs: readability and hard-to-test stuff (AUTOMATE all you can!-)
- tools and processes for CRs

# Code Reviews

- identified very early (generations ago!) as a great way to enhance code quality
  - way cheaper than having customers find those bugs "in the field"... or even QA!-)
  - also catches problems testing and static analysis can't (clarity, readability, names)
- widely acknowledged "best practice"
- so why is it sometimes "more honored in the breach than in the observance"?-)
  - reviews done spottily or not at all
  - "rubber-stamp" reviews...

Google

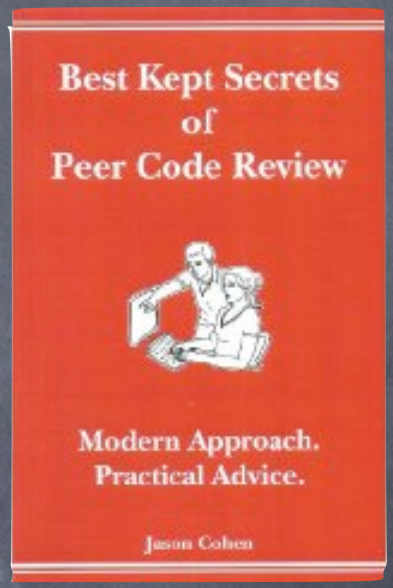
# "Fagan inspection"

- VERY heavy-weight part of very heavy-weight, high-ceremony processes
- requirement docs, test plans, architectural design, &c, are "inspected" as well as code
- phases: planning, overview meeting,
  - {preparation, inspection meeting, rework, follow-up verification} 1+ times
  - "moderator" decides; ~6 people/meeting
- high-formality -> very high cost, unsuitable except in high-formality/rigid processes
  - ...which have other limits/problems too;-)

Google

# Where's the ROI?

- smartbearsoftware.com
- they want to sell you their products ("Code Collaborator" &c) and services, BUT, they do so with clarity, transparency, and honor, providing lots of good free supporting materials
- case studies, analysis, biblio, ...
- summary: Fagan is good, but lightweight is better (esp. w/ good tool support of course;-)






Google

# Don't be too lightweight (1)

- "no process at all"? reviews are NOT your top problem, then!-)
  - worst: no version control system...!
  - next worst: no automated tests...!
  - then: no accepted "team style", no auto checks for it, no bug/feature tracker, ...
- FIRST fix any such gaping, bleeding wounds,
- THEN proceed to worrying about code reviews!-)

# Don't be too lightweight (2)

- If you have "just enough" process & tools, but no space in them for code reviews
  - so, they happen sporadically (if at all)
    - and/or are often "rubberstamp"...
  - maybe "not for the Big Guys"...?
  - "pair programming instead"...?
  - "TDD instead"...!?
- so THIS is the right state from which to enhance your process!-)

# PP vs CRs

- pair programming is great, BUT,
- not really a substitute for code reviews!
  - the pair can easily get "synchronized"
  - some things are clear/obvious to both, as "they've been there at creation", but...
    - may not be clear to others who weren't there (may need comments, &c)
    - may hide subtle problems ("given enough eyeballs, all bugs are shallow": 4 may not be enough!-)
- best practice: do *both* PP *and* CRs!

Google

# TDD vs CRs

- test-driven development is great, BUT,
- ABSOLUTELY no substitute for code reviews!
  - leaves you w/great unit tests (yay tests!)
    - tests that also help document the code
    - many kinds of bugs WILL be caught
  - BUT: no guarantee of clarity, readability, consistent naming, ...
    - AND: some kinds of bugs often escape
- best practice: do *both* TDD *and* CRs!

# "Not 4 the Big Guys"? (1)

- excessive "reverence" for authority, fame or seniority can inhibit "juniors'" reviews
  - unlikely to be an issue in US geek culture
  - however, watch out (esp. other cultures!)
  - can also cause "rubberstamp reviews"
- antidote: "don't criticize, ASK"
  - no: "this will break when the arg == 0"
  - yes: "what happens when the arg == 0?"
    - frame it as LEARNING about things
    - may prompt a fix, a comment, ...

# "Not 4 the Big Guys"? (2)

- "Big Guys" sometimes have fragile egos...!
  - e.g., when they see perfection as a state,
  - not as a goal + a process to move towards it!-)
  - big negative effect on team spirit
    - may overshadow BG's contributions
- "don't criticize, ASK" can help w/this too
  - less likely to trigger defensive reflexes
  - try moving towards that style in general

# "Rubberstamp review" (1)

- may be "excessive reverence"
  - "if HE did it this way, I can't question it!"
  - easy to counter: cast reviewing as a way to learn better technique &c
    - which it actually IS, crucially & often!
- occasionally seen: the reverse effect
  - way-picky interminable back-&-forths
  - often, mostly about bikeshedding
  - counter: focus on team-spirit
    - and: *making forward progress*!

# "Rubberstamp review" (2)

- may be "lack of buy-in"
  - reviewer grudgingly agrees to perform reviews "as a chore", doesn't believe they're actually worth their time
  - worst: "swapping" rubberstamp reviews
  - need evangelism, supporting data!
- (also sometimes causes reverse-effect)

# The Social Side of CRs (1)

- only model I've ever seen work: everyone gets their work reviewed, every time
  - everyone learns AND everyone teaches
  - you don't every morning stop to think and decide "do I really need to brush my teeth today"? You make it a HABIT!-)
  - think of CRs as part of "code hygiene"!-)



Google

# The Social Side of CRs (2)

- generally best: every review is open to whole team, everyone is heartily invited to comment, but one designated reviewer "owns" the review (and follow-up to check defects are clarified & fixed)... like for any other action item!-)
- potential problem: "reviewer shopping"
  - social problems are best solved socially and culturally
  - however, sometimes a techie fix can help
    - e.g., random reviewer assignment



Google

# What you DON'T check

- Do NOT use CR time to check for such things as formatting issues &c
  - your team's style MUST be auto-checked by lint-like or IDE tools; if you're doing manually what's easily automated, EEK!-)
  - same for unit-test coverage &c...

A U T O M A T E !

  - also, no need to focus as much on stuff that unit-tests (&c) would catch (but...)

# So WHAT do you check?

- check, particularly, those issues that tools will "never" catch: readability, clarity, understanding, significant&consistent names
- plus, focus on hard-to-test-for issues...:
  - quality of tests
  - proper error handling
  - resource-leak issues
  - security issues
  - multi-tasking
  - performance
  - portability

# Readability &c: docs first!

- comments & other internal docs...:
  - match code but never "just repeat it"
  - are good, concise, correct English (or whatever language the project is in!-)
  - use names consistent with those used in the code & generally terminology well suited to the programming lang (int vs integer, bool vs Boolean, ...)
  - *point to* docs on complex algorithms or external docs (specs, libs &c), DON'T repeat such things in the middle of code

# Readability &c: non-docs

- code is clear, readable, concise (but not TOO terse)
  - and respects DRY (Don't Repeat Yourself)
- names are meaningful & consistent
- UI, if any, is clear and follows the whole project's style (*especially* error/log info!!!)
  - *appropriate* info in error msgs & logs!
- no "reinventing the wheel": *reuse*!
  - the clearest code (and the least likely to break) is the code that's NOT there;-)

# Hard-to-test issues (1)

- beyond test coverage, are corner and error cases well tested (w/mocks, DI, &c)?
- error handling: if language has exceptions, are they handled properly? if not, are all return values checked for error cases?
- any memory leaks (or equivalent in GC languages)? any other resource leaks?
  - is everything properly cleaned up along all paths? including error ones? tests?
- any security issues? SQL injection, XSS, buffer overflow, ...

# Hard-to-test issues (2)

- multi-tasking (shudder...;-): any race conditions? possible deadlocks? be VERY defensive here...!
  - (if feasible, architect appropriately...)
- performance: any premature optimizations? However, also enforce "waste not want not" (no easily avoided overhead if "the fast way" is just about as simple;-)
- any portability issues? what platforms has the code been fully tested on?

# Tools & Processes

- lightweight CRs should be doable remotely and at convenient times for all involved
  - face-to-face/over-the-shoulder style has pluses, but is intrinsically higher-weight
    - plus, no useful "audit trail" is left
    - still might be good in "sprints"/"spikes"
  - remote-but-synchronized (IM, IRC and other "chat" approaches) may be usable
    - if no timezone &c issues...
- email DOES meet these basic needs...!

# Code Reviews by email

- definitely not a "shiny new tool"...;-)
- however, it has many clear pluses
  - universally available (web & otherwise)
  - typically very customizable user-agents
  - programs are also easily customized to:
    - automatically send e-mails on triggers
    - receive e-mails and act upon them
- any "shiny new tool" SHOULD be designed to cooperate smoothly w/email CRs!-)

# email CR workflow (1)

- VCS (or reviewee) starts a CR by mailing main reviewer (CC the team) with text and/or pointer to change-set ("patch", diff, &c)
  - pointer/identifier typically very useful (depending on VCS capabilities), as it may allow easy viewing of diffs on whole files
  - but, diff text is often a good "hook" for reviewer comments!
  - so, I'd suggest using both, when feasible

# email CR workflow (2)

- ideally, CR mailing should happen BEFORE actual commit/push of change-set to the codebase -- upholds trunk/head/tip quality
- if that's unfeasible (due to VCS limitations), consider a "staging repository" or branch for "committed but unreviewed" changes
  - only commit to trunk/head once the review is complete and satisfactory
- distributed VCS' flexibility allows for many different workflows, of course

# email CR workflow (3)

- reviewer comments on regions of the diffs
  - asking for clarifications,
  - suggesting possible changes,
  - pointing out definite problems (and thus implicitly demanding changes)
    - question-style may be best...
- others may offer similar feedback
- author MUST solve each issue to the reviewer's satisfaction: reviewer rules!
- ...whence the "reviewer shopping" issue;-)



Google

# changeset size for CRs

- aim for changesets of about 200 lines (depending on your language's terseness;-), INCLUDING comments (which need CR too!)
- smaller may obviously be needed (for simple bug fixes, tiny feature additions)
- bigger is harder to review well... try HARD not to exceed about 400 lines, PLEASE...

# Duration of CR sessions

- don't spend more than 60-90 minutes reviewing: effectiveness "drops off a cliff" around about that time!
  - "habituation effects" byte really hard
  - alas, there's no "getting in the zone" for CRs anywhere to the extent it can happen for coding or debugging sessions
- similarly: no more than 1 review/half day (1 in the morning, 1 in the afternoon)
- sometimes there will be pressure, of course...

# shiny new tools (OSS only)

- Rietveld (see http://code.google.com/p/rietveld/ and codereview.appspot.com)
  - hosted on GAE, so you don't even have to provide your own server...;-)
  - VERY "shiny new" at this time, still;-)
- Review Board (http://review-board.org/)
- Codestriker (http://codestriker.sourceforge.net/) -- in perl!
- Java Code Reviewer (http://jcodereview.sourceforge.net/ -- actually in Python and usable for non-Java reviews;-)

# Q & A

http://www.aleax.it/osc08_crev.pdf



Google