Collaborative Programming: Pair Programming and Reviews

CSE 403

Pair programming

- **pair programming**: 2 people, 1 computer
  - take turns "driving"
  - rotate pairs often
  - pair people of different experience levels

- pros:
  - Can produce better code; notice problems faster
  - Inexperienced coder can learn from experienced ones
  - Reduces bus number
- cons:
  - Distracting (can't get into flow; but better documentation)
  - Can impede design work
  - Straightforward work doesn't require two people

Reviews

- **Review**: Other team member(s) read an artifact (design, specification, code) and suggest improvements
  - documentation
  - defects in program logic
  - program structure
  - coding standards & uniformity with codebase
  - enforce subjective rules
  - ... everything is fair game
- Feedback → refactoring → reviews → … → approval
- Can occur before or after code is committed

Analogy: writing a newspaper article

What is the effectiveness of…

- Spell-check/grammar check
- Reviewing your own article
- Others reviewing your article

Motivation for reviews

- Can catch most bugs, design flaws early
- > 1 person has seen every piece of code
  - Insurance against author's disappearance
  - Accountability (both author and reviewers are accountable)
- Forcing function for documentation and code improvements
  - Authors must articulate their decisions
  - Authors participate in the discovery of flaws
  - Prospect of a review raises your quality threshold
- Inexperienced personnel get experience without hurting code quality
  - Pairing them up with experienced developers
  - Can learn by being a reviewer as well
- Explicit non-purpose:
  - Assessment of individuals for promotion, pay, ranking, etc.
  - Management is usually not permitted at reviews

Motivation by the numbers

- Average defect detection rates
  - Unit testing: 25%
  - Function testing: 35%
  - Integration testing: 45%
  - **Design and code inspections: 55% and 60%**.
- 11 programs developed by the same group of people
  - First 5 without reviews: average 4.5 errors per 100 lines of code
  - Remaining 6 with reviews: average 0.82 errors per 100 lines of code
  - Errors reduced by **> 80%**.
- IBM's Orbit project: 500,000 lines, 11 levels of inspections. **Delivered early** with **1% of the predicted errors**.
- After **AT&T** introduced reviews, **14% increase in productivity** and a **90% decrease in defects**.

(From Steve McConnell's Code Complete)

Logistics of the code review

- What is reviewed:
  - A specification
  - A coherent module (sometimes called an “inspection”)
  - A single checkin or code commit (incremental review)
- Who participates:
  - One other developer
  - A group of developers
- Where:
  - In-person meeting
    - Best to prepare beforehand: artifact is distributed in advance
    - Preparation usually identifies more defects than the meeting
  - Email/electronic

Review technique and goals

- Specific focus?
  - Sometimes, a specific list of defects or code characteristics
    - Error-prone code
    - Previously-discovered problem types
    - Security
    - Checklist (coding standards)
      - Automated tools (type checkers, lint) can be better
- Outcomes:
  - Only identify defects, or also brainstorm fixes?

Code review variations

- **walkthrough**: playing computer, trace values of sample data
- **group reading**: as a group, read whole artifact line-by-line
- **presentation**: author presents/explains artifact to the group
- **offline preparation**: reviewers look at artifact by themselves (possibly with no actual meeting)

Common open source approach: incremental code review

- Each small change is reviewed *before* it is committed
- No change is accepted without signoff by a "committer"
  - Assumed to know the whole codebase well
  - Sometimes committers are excepted
- Code review can (d)evolve into a design discussion
- Good if you don't trust the programmer

Another approach: holistic group code review

- Review an entire component
  - Documentation is required (as is good style)
  - *No* extra overview from developer
- Each reviewer focuses where he/she sees fit
  - Mark up with lots of comments
  - Identify 5 most important issues
- At meeting, go around the table raising one issue
  - Discuss the reasons for the current design, and possible improvements
- Author addresses all issues in comments
  - Not just those raised in the meeting
- Better for discussing design, for training, for establishing norms
- Applicable if:
  - You trust the programmer and/or tests (impact of unreviewed code is likely to be small)
  - No time or expertise for incremental code reviews

Code Reviews at Google

"All code that gets submitted needs to be reviewed by at least one other person, and either the code writer or the reviewer needs to have readability in that language. Most people use Mondrian to do code reviews, and obviously, we spend a good chunk of our time reviewing code."

-- Amanda Camp, Software Engineer, Google

Code reviews at Yelp

“At Yelp we use review-board. An engineer works on a branch and commits the code to their own branch. The reviewer then goes through the diff, adds inline comments on review board and sends them back. The reviews are meant to be a dialogue, so typically comment threads result from the feedback. Once the reviewer's questions and concerns are all addressed they'll click "Ship It!" and the author will merge it with the main branch for deployment the same day.”

-- Alan Fineberg, Software Engineer, Yelp

Code reviews at WotC

"At Wizards we use Perforce for SCM. I work with stuff that manages rules and content, so we try to commit changes at the granularity of one bug at a time or one card at a time. Our team is small enough that you can designate one other person on team as a code reviewer. Usually you look at code sometime that week, but it depends on priority. It's impossible to write sufficient test harnesses for the bulk of our game code, so code reviews are absolutely critical."

-- Jake Englund, Software Engineer, MtGO

Code reviews at Facebook

"At Facebook, we have an internally-developed web-based tool to aid the code review process. Once an engineer has prepared a change, she submits it to this tool, which will notify the person or people she has asked to review the change, along with others that may be interested in the change – such as people who have worked on a function that got changed.

At this point, the reviewers can make comments, ask questions, request changes, or accept the changes. If changes are requested, the submitter must submit a new version of the change to be reviewed. All versions submitted are retained, so reviewers can compare the change to the original, or just changes from the last version they reviewed. Once a change has been submitted, the engineer can merge her change into the main source tree for deployment to the site during the next weekly push, or earlier if the change warrants quicker release."

- Ryan McElroy, Software Engineer, Facebook

Software quality assurance (review)

- What are we assuring?
- Why are we assuring it?
- How do we assure it?
- How do we know we have assured it?

What are we assuring?

- Validation: building the right system?
- Verification: building the system right?

- Presence of good properties?
- Absence of bad properties?

- Identifying errors?
- Confidence in the absence of errors?

- Robust? Safe? Secure? Available? Reliable? Understandable? Modifiable? Cost-effective? Usable? …

Why are we assuring it?

- Business reasons
- Ethical reasons
- Professional reasons
- Personal satisfaction
- Legal reasons
- Social reasons
- Economic reasons
- …

Code review exercise

What feedback would you give the author? What changes would you request before checkin?

```
public class Account {
   double principal,rate;   int daysActive,accountType;

   public static final int STANDARD=0, BUDGET=1,
   PREMIUM=2, PREMIUM_PLUS=3;
}

...

public static double calculateFee(Account[] accounts)
{
   double totalFee = 0.0;
   Account account;
   for (int i=0;i<accounts.length;i++) {
      account=accounts[i];
      if ( account.accountType == Account.PREMIUM ||
           account.accountType == Account.PREMIUM_PLUS )
         totalFee += .0125 * (       // 1.25% broker's fee
            account.principal * Math.pow(account.rate,
            (account.daysActive/365.25))
            - account.principal);   // interest
   }
   return totalFee;
}
```

Improved code (page 1)

```
/** An individual account.  Also see CorporateAccount. */
public class Account {
   private double principal;
   /** The yearly, compounded rate (at 365.25 days per year). */
   private double rate;
   /** Days since last interest payout. */
   private int daysActive;
   private Type type;

   /** The varieties of account our bank offers. */
   public enum Type {STANDARD, BUDGET, PREMIUM, PREMIUM_PLUS}

   /** Compute interest. **/
   public double interest() {
      double years = daysActive / 365.25;
      double compoundInterest = principal * Math.pow(rate, years);
      return compoundInterest - principal;
   }

   /** Return true if this is a premium account. **/
   public boolean isPremium() {
      return accountType == Type.PREMIUM ||
             accountType == Type.PREMIUM_PLUS;
   }
```

Improved code (page 2)

```
    /** The portion of the interest that goes to the broker. **/
    public static final double BROKER_FEE_PERCENT = 0.0125;

    /** Return the sum of the broker fees for all the given
    accounts. **/
    public static double calculateFee(Account[] accounts) {
       double totalFee = 0.0;
       for (Account account : accounts) {
          if (account.isPremium()) {
             totalFee += BROKER_FEE_PERCENT * account.interest();
          }
       }
       return totalFee;
    }

}
```