Collaborative Programming: Pair Programming and Reviews

and

Maintenance and Refactoring

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
  - An inexperienced coder can learn from an experienced one

- cons:
  - Some people don’t like it

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 leads to refactoring, followed by a additional reviews and eventually approval

Motivation for reviews

- Can catch most bugs, design flaws early.
- > 1 person has seen every piece of code.
  - Prospect of someone reviewing your code raises quality threshold.
- Forces code authors to articulate their decisions and to participate in the discovery of flaws.
- Allows junior personnel to get early hands-on experience without hurting code quality
  - Pairing them up with experienced developers
  - Can learn by being a reviewer as well
- Accountability. Both author and reviewers are accountable for the code.
- Explicit non-purpose:
  - Assessment of individuals for promotion, pay, ranking, etc.
  - Management is usually not permitted at reviews

Types of code review

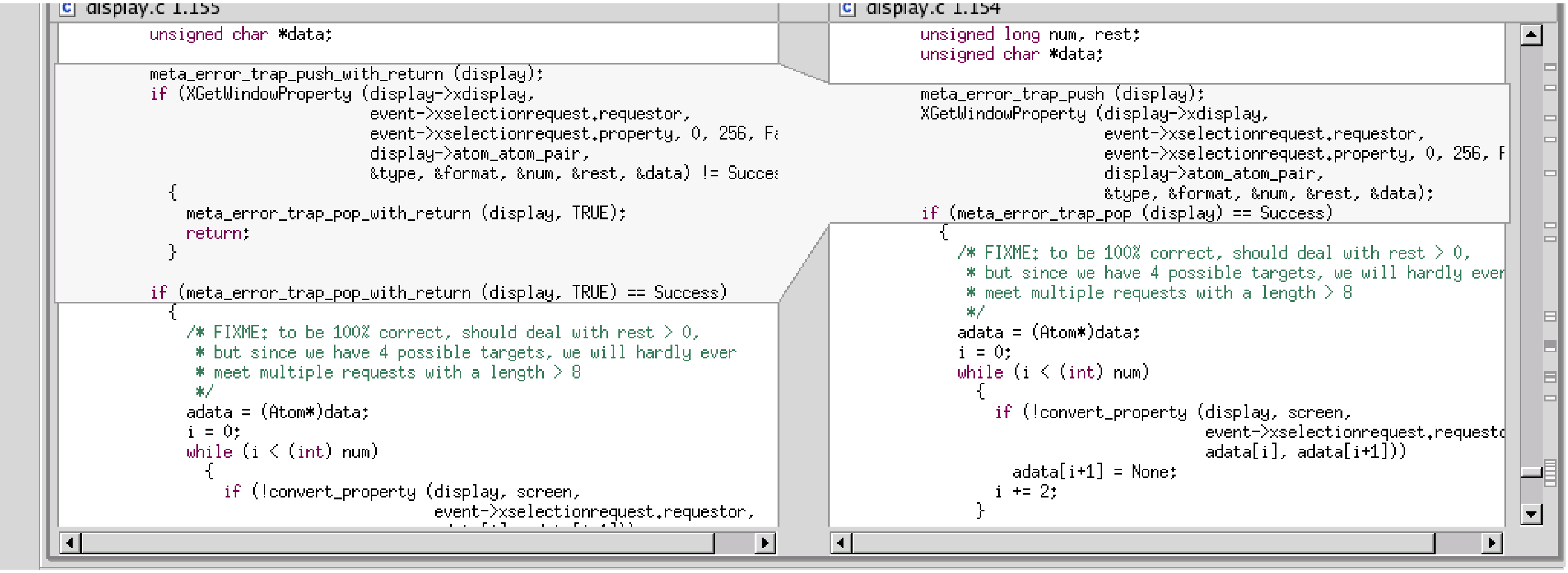
- What is reviewed:
  - A specification
  - A coherent module (sometimes called an “inspection”)
  - A single checkin or code commit (incremental review)
- Who participates:
  - One other developer
  - A group of developers
- Where:
  - In-person meeting
    - Best to prepare beforehand: artifact is 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
- Technique
  - Does developer present the artifact to a group?
  - Only identify defects, or also brainstorm fixes?
  - Sometimes, a specific methodology
    - "Walkthrough" = playing computer, trace values of sample data

Code reviews in industry

- Code reviews are a **very** common industry practice.
- Made easier by advanced tools that:
  - integrate with configuration management systems
  - highlight changes (i.e., diff function)
  - allow traversing back into history
  - E.g.: Eclipse, SVN tools



display.c 1.155

```
        unsigned char *data;

        meta_error_trap_push_with_return (display);
        if (XGetWindowProperty (display->xdisplay,
                                event->xselectionrequest.requestor,
                                event->xselectionrequest.property, 0, 256, Fa
                                display->atom_atom_pair,
                                &type, &format, &num, &rest, &data) != Succes
          {
            meta_error_trap_pop_with_return (display, TRUE);
            return;
          }

        if (meta_error_trap_pop_with_return (display, TRUE) == Success)
          {
            /* FIXME: to be 100% correct, should deal with rest > 0,
             * but since we have 4 possible targets, we will hardly ever
             * meet multiple requests with a length > 8
             */
            adata = (Atom*)data;
            i = 0;
            while (i < (int) num)
              {
                if (!convert_property (display, screen,
                                       event->xselectionrequest.requestor,
```

display.c 1.154

```
        unsigned long num, rest;
        unsigned char *data;

        meta_error_trap_push (display);
        XGetWindowProperty (display->xdisplay,
                            event->xselectionrequest.requestor,
                            event->xselectionrequest.property, 0, 256, F
                            display->atom_atom_pair,
                            &type, &format, &num, &rest, &data);
        if (meta_error_trap_pop (display) == Success)
          {
            /* FIXME: to be 100% correct, should deal with rest > 0,
             * but since we have 4 possible targets, we will hardly ever
             * meet multiple requests with a length > 8
             */
            adata = (Atom*)data;
            i = 0;
            while (i < (int) num)
              {
                if (!convert_property (display, screen,
                                       event->xselectionrequest.requesto
                                       adata[i], adata[i+1]))
                  adata[i+1] = None;
                i += 2;
              }
```

My approach

- Distribute code (or other artifacts) ahead of time
  - Common pagination
  - 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 takes all printouts and addresses all issues
  - Not just those raised in the meeting

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 right system?
- Verification: building system right?

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

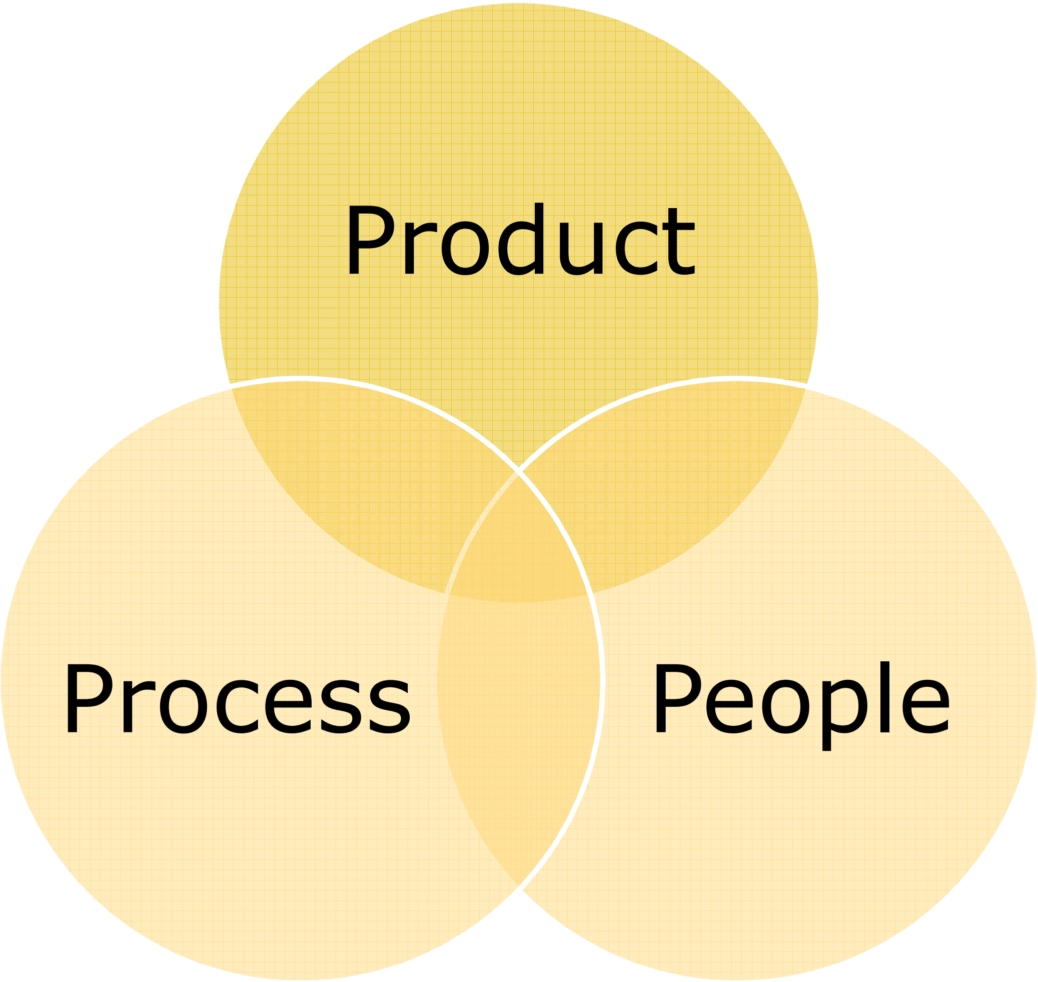
- Identifying errors?
- Confidence in the absence of errors?

- 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
- …

How do we assure it?



How do we know we have assured it?

- Depends on “it”
- Depends on what we mean by “assurance”
- …

Exercise

"Code review" this checkin. 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-principal
   }
   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;
   }

}
```

Refactoring

Problem: "Bit rot"

- After several months and new versions, many codebases reach one of the following states:
  - *rewritten*: Nothing remains from the original code.
  - *abandoned*: The original code is thrown out and rewritten from scratch.

- Why is this?
  - Systems evolve to meet new needs and add new features
  - If the code's structure does not also evolve, it will "rot"

  - This can happen even if the code was initially reviewed and well-designed at the time of checkin, and even if checkins are reviewed

Code maintenance

- **maintenance**: Modification of a software product after it has been delivered.

  Purposes:

  - fix bugs
  - improve performance
  - improve design
  - add features

  - ~80% of maintenance is for non-bug-fix-related activities such as adding functionality (Pigosky 1997)

Maintenance is hard

- It's harder to maintain (someone else's?) code than write your own new code.
  - "house of cards" phenomenon (don't touch it!)
  - must understand code written by another developer,
    or code you wrote at a different time with a different mindset
  - most developers *hate* code maintenance
    - Why?

- Maintenance is how devs spend most of their time.

- It pays to design software well and plan ahead so that later maintenance will be less painful.
  - Capacity for future change must be anticipated

Refactoring

- **refactoring**: Improving a piece of software's internal structure without altering its external behavior.
  - Not the same as code rewriting
  - Incurs a short-term time/work cost to reap long-term benefits
  - A long-term investment in the overall quality of your system.

Why refactor?

- Why fix a part of your system that isn't broken?

  Each part of your system's code has 3 purposes:

  - 1. to execute its functionality,
  - 2. to allow change,
  - 3. to communicate well to developers who read it.

  - If the code does not do one or more of these, it *is* broken.

Low-level refactoring

Names:

- Renaming (methods, variables)
- Naming (extracting) "magic" constants

Procedures:

- Extracting code into a method
- Extracting common functionality (including duplicate code) into a module/method/etc.
- Inlining a method/procedure
- Changing method signatures

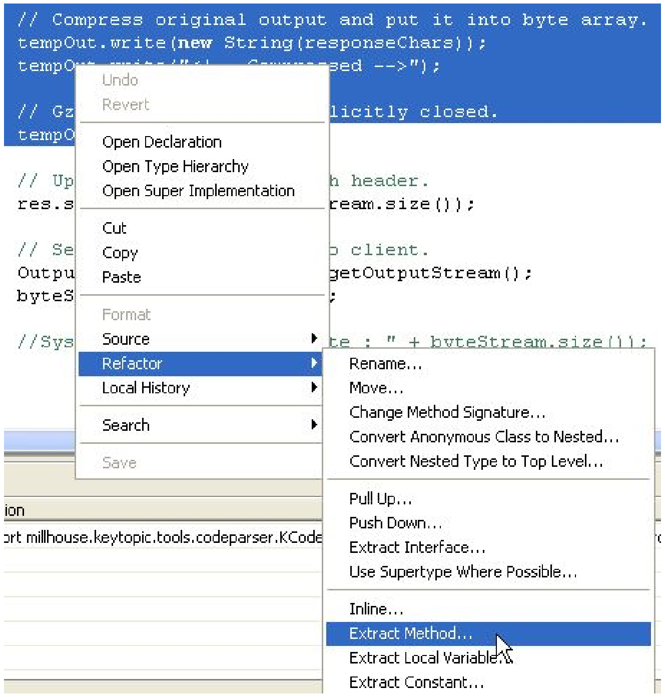
Reordering:

- Splitting one method into several to improve cohesion and readability (by reducing its size)
- Putting statements that semantically belong together near each other

  - See also http://www.refactoring.org/catalog/

IDE support for refactoring

- Eclipse / Visual Studio support:
  - variable / method / class renaming
  - method or constant extraction
  - extraction of redundant code snippets
  - method signature change
  - extraction of an interface from a type
  - method inlining
  - providing warnings about method invocations with inconsistent parameters
  - help with self-documenting code through auto-completion



```
// Compress original output and put it into byte array.
tempOut.write(new String(responseChars));
tempO                          sed -->");

// Gz                         licitly closed.
tempO

// Up                         h header.
res.s                         ream.size());

// Se                         o client.
Outpu                         getOutputStream();
byteS                         ;

//Sys                         te : " + byteStream.size());
```

Undo
Revert
Open Declaration
Open Type Hierarchy
Open Super Implementation
Cut
Copy
Paste
Format
Source
Refactor
Local History
Search
Save
Rename...
Move...
Change Method Signature...
Convert Anonymous Class to Nested...
Convert Nested Type to Top Level...
Pull Up...
Push Down...
Extract Interface...
Use Supertype Where Possible...
Inline...
Extract Method...
Extract Local Variable...
Extract Constant...
ion
ort millhouse.keytopic.tools.codeparser.KCode

Higher-level refactoring

- Refactoring to design patterns
- Exchanging risky language idioms with safer alternatives
- Performance optimization
- Clarifying a statement that has evolved over time or is unclear

- Compared to low-level refactoring, high-level is:
  - Not as well-supported by tools
  - Much more important!

Refactoring plan?

- When you identify an area of your system that:
  - isn't especially well designed
  - isn't especially thoroughly tested, but seems to work so far
  - now needs new features to be added

- What should you do?
  - Assume that you have adequate time to "do things right." (Not always a valid assumption in software...)

Recommended refactor plan

- When you identify an area of your system that:
  - isn't especially well designed
  - isn't especially thoroughly tested, but seems to work so far
  - now needs new features to be added

- What should you do?
  - Write unit tests that verify the code's external correctness.
    - (They should pass on the current, badly designed code.)
  - Refactor the code.
    - (Some unit tests may break. Fix the bugs.)
  - Add the new features.

"I don't have time to refactor!"

- Refactoring incurs an up-front cost.
  - many developers don't want to do it
  - most management don't like it, because they lose time and gain "nothing" (no new features)

- However...
  - well-written code is much more conducive to rapid development (some estimates put ROI at 500% or more for well-done code)
  - finishing refactoring increases programmer morale
    - developers prefer working in a "clean house"

- When to refactor?
  - best done continuously (like testing) as part of the SWE process
  - hard to do well late in a project (like testing)
    - Why?

Should startups refactor?

- Many small companies and startups skip refactoring.
  - "We're too small to need it!"
  - "We can't afford it!"

- Reality:
  - Refactoring is an investment in quality of the company's product and code base, often their prime assets
  - Many web startups are using the most cutting-edge technologies, which evolve rapidly. So should the code

  - If a key team member leaves (common in startups), ...
  - If a new team member joins (also common), ...