Reminders

- Respond to the project plan teamwork assessment on Moodle.
  - You won't get your project plan grades until everyone on your team responds.
- Tests
  - Grades will be posted this weekend.
  - Solutions will be on Moodle this weekend.
  - I'll hand the tests back Nov 20.

Any comments? Length? Difficulty?

More reminders

- Next week, I am traveling
  - Next Tuesday, Nov 13: no lecture

  Use time to work on project
  - Next Thursday, Nov 15: guest lecture
- For the homework (due 11/15), Wenzhe is available during office hours: Monday 2PM–3PM in CS 316
- I am available via email

Project status report

- Due Nov 17 on Moodle
- 1 per team
- Submit a 1-2 paragraph summary of your team's progress.
- Tell me what's done and if you are stuck on anything
- The goal is for me to help out, not to grade you

Today's plan

- Teamwork
- Debugging (especially in teams)

Working in Teams

- Why is teamwork hard?
- Not getting into each other's way
- Positive teamwork

Team pros and cons

- Benefits
  - Attack bigger problems in a short period of time
  - Utilize the collective experience of everyone
- Risks
  - Communication and coordination issues
  - Groupthink: diffusion of responsibility; going along
  - Working by inertia; not planning ahead
  - Conflict or mistrust between team members

Communication: powerful but costly!

- Communication requirements increase with increasing numbers of people
- Everybody to everybody: quadratic cost
- Every attempt to communicate is a chance to miscommunicate
- But *not* communicating will *guarantee* miscommunication

What about conflicts?

What can cause conflicts?

- Two people want to work on the same file
  - Google docs lets you do that

But…

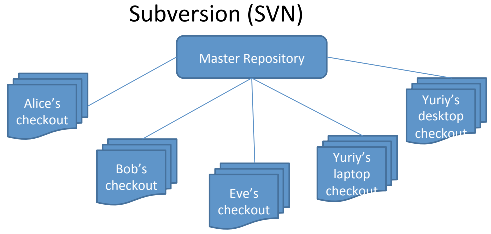
- What about same line?
- What about timing?
- What about design decisions?

Version control

Version control aims to allow multiple people to work in parallel.

Centralized version control

- (old model)
- Examples: Concurrent Versions System (CVS)
  Subversion (SVN)



Problems with centralized VC

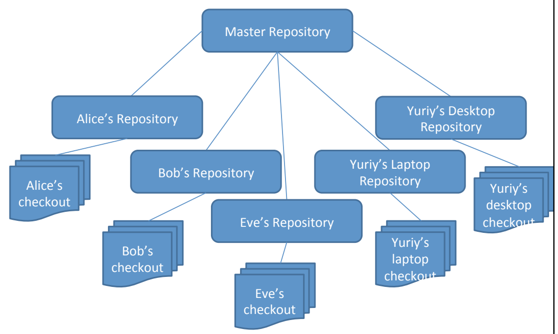
- What if I don't have a network connection?
- What if I am implementing a big change?
- What if I want to explore project history later?

Distributed version control

- (new model)
- Examples: Mercurial (Hg), Git, Bazaar, Darcs, …

- Local operations are fast (and possible)
- History is more accurate
- Merging algorithms are far better

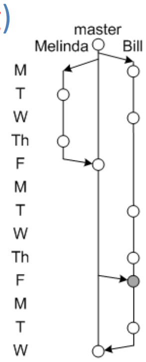
Distributed version control model



History view (log)

- Bill and Melinda work at the same time

- At the end, all repositories have the same, rich history



What VC does the cloud provide?

- code.google.com has SVN and Hg
- bitbucket.org has Hg
- github.com has git
- sourceforge.net has SVN, CVS, git, Hg, Bazaar

- You can run whatever you want of UW servers

Team structures

- Tricky balance among
  - progress on the project/product
  - expertise and knowledge
  - communication needs

"A team is a set of people with complementary skills who are committed to a common purpose, performance goals, and approach for which they hold themselves mutually accountable."

– Katzenbach and Smith

Common SW team responsibilities

- Project management
- Functional management
- Developers: programmers, testers, integrators
- Lead developer/architect ("tech lead")

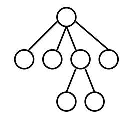
- These could be all different team members, or some members could span multiple roles.
- Key: Identify and stress roles **and** responsibilities

Issues affecting team success

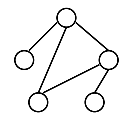
- Presence of a shared mission and goals
- Motivation and commitment of team members
- Experience level
  – and presence of experienced members
- Team size
  – and the need for bounded yet sufficient communication
- Team organization
  – and results-driven structure
- Reward structure within the team
  – incentives, enjoyment, empowerment (ownership, autonomy)

Team structure models

- Dominion model
  – Pros
    - clear chain of responsibility
    - people are used to it
  – Cons:
    - single point of failure at the commander
    - less or no sense of ownership by everyone



- Communion model
  – Pros
    - a community of leaders, each in his/her own domain
    - inherent sense of ownership
  – Cons
    - people aren't used to it (and this scares them)

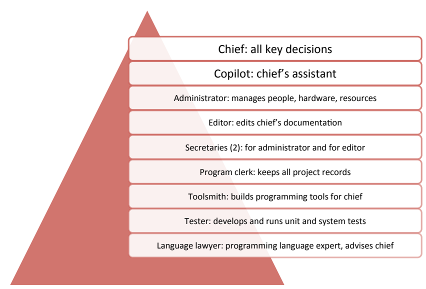


Team leadership

- Who makes the important product-wide decisions in your team?
  – One person?
  – All, by unanimous consent?
  – Other options?...
  – Is this an unspoken or an explicit agreement among team members?

Surgical/Chief Programmer Team

[Baker, Mills, Brooks]



Microsoft's team structure

[microsoft.com]

- **Program Manager**. Leads the technical side of a product development team, managing and defining the functional specifications and defining how the product will work.
- **Software Design Engineer.** Codes and designs new software, often collaborating as a member of a software development team to create and build products.
- **Software Test Engineer.** Tests and critiques software to assure quality and identify potential improvement opportunities and projects.

Toshiba Software Factory [Y. Matsumoto]

- Late 1970's structure for 2,300 software developers producing real-time industrial application software systems (such as traffic control, factory automation, etc.)
- Unit Workload Order Sheets (UWOS) precisely define a software component to be built
- Assigned by project management to developers based on scope/size/skills needed
- Completed UWOS fed back into management system
- Highly measured to allow for process improvement

Common factors in good teams

- Clear roles and responsibilities
  - Each person knows and is accountable for their work
- Monitor individual performance
  - Who is doing what, are we getting the work done?
- Effective communication system
  - Available, credible, tracking of issues, decisions
  - Problems aren't allowed to fester ("boiled frogs")
- Fact based decisions
  - Focus on the facts, not the politics, personalities, …

Motivation

- What motivates you?
  - Achievement
  - Recognition
  - Advancement
  - Salary
  - Possibility for growth
  - Interpersonal relationships
    - Subordinate
    - Superior
    - Peer
  - Status
  - Technical supervision opportunities

- Company policies
- Work itself
- Work conditions
- Personal life
- Job security
- Responsibility
- Competition
- Time pressure
- Tangible goals
- Social responsibility

Other?

De-motivators

- What takes away your motivation?
  - Micro-management or no management
  - Lack of ownership
  - Lack of effective reward structure
    - Including lack of simple appreciation for job well done
  - Excessive pressure and resulting "burnout"
  - Allowing "broken windows" to persist
  - Lack of focus in the overall direction
  - Productivity barriers
    - Asking too much; not allowing sufficient learning time; using the wrong tools
  - Too little challenge
  - Work not aligned with personal interests and goals
  - Poor communication inside the team

Today's plan

- Teamwork

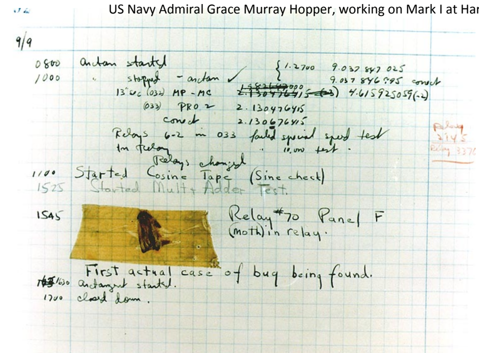
→Debugging (especially in teams)

Ways to get your code right

- Validation
  - Purpose is to uncover problems and increase confidence
  - Combination of reasoning and test
- Debugging
  - Finding out why a program is not functioning as intended
- Defensive programming
  - Programming with validation and debugging in mind
- Testing ≠ debugging
  - test: reveals existence of problem
  - debug: pinpoint location + cause of problem

A bug – September 9, 1947

US Navy Admiral Grace Murray Hopper, working on Mark I at Harvard



A Bug's Life

- Defect – mistake committed by a human
- Error – incorrect computation
- Failure – visible error: program violates its specification
- Debugging starts when a failure is observed
  - Unit testing
  - Integration testing
  - In the field

Defense in depth

1. Make errors impossible
   - Java makes memory overwrite bugs impossible
2. Don't introduce defects
   - Correctness: get things right the first time
3. Make errors immediately visible
   - Local visibility of errors: best to fail immediately
   - Example: checkRep() routine to check representation invariants
4. Last resort is debugging
   - Needed when effect of bug is distant from cause
   - Design experiments to gain information about bug
     - Fairly easy in a program with good modularity, representation hiding, specs, unit tests etc.
     - Much harder and more painstaking with a poor design, e.g., with rampant rep exposure

First defense: Impossible by design

- In the language
  - Java makes memory overwrite bugs impossible
- In the protocols/libraries/modules
  - TCP/IP will guarantee that data is not reordered
  - BigInteger will guarantee that there will be no overflow
- In self-imposed conventions
  - Hierarchical locking makes deadlock bugs impossible
  - Banning the use of recursion will make infinite recursion/insufficient stack bugs go away
  - Immutable data structures will guarantee behavioral equality
  - Caution: You must maintain the discipline

Second defense: correctness

- Get things right the first time
  - Don't code before you think! Think before you code.
  - If you're making lots of easy-to-find bugs, you're also making hard-to-find bugs – don't use compiler as crutch
- Especially true, when debugging is going to be hard
  - Concurrency
  - Difficult test and instrument environments
  - Program must meet timing deadlines
- Simplicity is key
  - Modularity
    - Divide program into chunks that are easy to understand
    - Use abstract data types with well-defined interfaces
    - Use defensive programming; avoid rep exposure
  - Specification
    - Write specs for all modules, so that an explicit, well-defined contract exists between each module and its clients

Third defense: immediate visibility

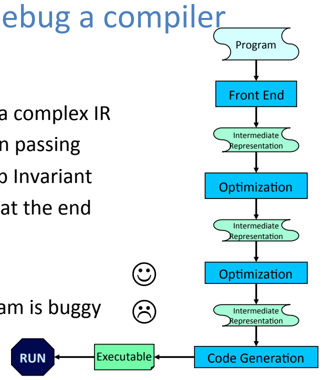
- If we can't prevent bugs, we can try to localize them to a small part of the program
  - Assertions: catch bugs early, before failure has a chance to contaminate (and be obscured by) further computation
  - Unit testing: when you test a module in isolation, you can be confident that any bug you find is in that unit (unless it's in the test driver)
  - Regression testing: run tests as often as possible when changing code. If there is a failure, chances are there's a mistake in the code you just changed
- When localized to a single method or small module, bugs can be found simply by studying the program text

Benefits of immediate visibility

- Key difficulty of debugging is to find the code fragment responsible for an observed problem
  - A method may return an erroneous result, but be itself error free, if there is prior corruption of representation
- The earlier a problem is observed, the easier it is to fix
  - For example, frequently checking the rep invariant helps the above problem
- General approach: fail-fast
  - Check invariants, don't just assume them
  - Don't try to recover from bugs – this just obscures them

How to debug a compiler

- Multiple passes
  - Each operate on a complex IR
  - Lot of information passing
  - Very complex Rep Invariant
  - Code generation at the end
- Bug types:
  - Compiler crashes
  - Generated program is buggy



Don't hide bugs

```
// k is guaranteed to be present in a
int i = 0;
while (true) {
    if (a[i]==k) break;
    i++;
}
```

- This code fragment searches an array a for a value k.
  - Value is guaranteed to be in the array.
  - If that guarantee is broken (by a bug), the code throws an exception and dies.
- Temptation: make code more "robust" by not failing

Don't hide bugs

```
// k is guaranteed to be present in a
int i = 0;
while (i<a.length) {
    if (a[i]==k) break;
    i++;
}
```

- Now at least the loop will always terminate
  - But no longer guaranteed that a[i]==k
  - If rest of code relies on this, then problems arise later
  - *All we've done is obscure the link between the bug's origin and the eventual erroneous behavior it causes.*

Don't hide bugs

```
// k is guaranteed to be present in a
int i = 0;
while (i<a.length) {
    if (a[i]==k) break;
    i++;
}
assert (i<a.length) : "key not found";
```

- Assertions let us document and check invariants

  Abort program as soon as problem is detected

Inserting Checks

- Insert checks galore with an intelligent checking strategy
  - Precondition checks
  - Consistency checks
  - Bug-specific checks
- Goal: stop the program as close to bug as possible

  Use debugger to see where you are, explore program a bit

Checking For Preconditions

```
// k is guaranteed to be present in a
    int i = 0;
    while (i<a.length) {
        if (a[i]==k) break;
        i++;
    }
    assert (i<a.length) : "key not found";
```

Precondition violated? Get an assertion!

Downside of Assertions

```
static int sum(Integer a[], List<Integer> index) {
   int s = 0;
   for (e:index) {
         assert(e < a.length, "Precondition violated");
         s = s + a[e];
   }
   return s;
}
```

Assertion not checked until we use the data

Fault occurs when bad index inserted into list

May be a long distance between fault activation and error detection

checkRep: Data Structure Consistency Checks

```
static void checkRep(Integer a[], List<Integer> index) {
   for (e:index) {
         assert(e < a.length, "Inconsistent Data Structure");
   }
}
```

- Perform check after all updates to minimize distance between bug occurrence and bug detection
- Can also write a single procedure to check ALL data structures, then scatter calls to this procedure throughout code

Bug-Specific Checks

```
static void check(Integer a[], List<Integer> index) {
   for (e:index) {
         assert(e != 1234, "Inconsistent Data Structure");
   }
}
```

Bug shows up as 1234 in list

Check for that specific condition

Checks In Production Code

- Should you include assertions and checks in production code?
  - Yes: stop program if check fails – don't want to take chance program will do something wrong
  - No: may need program to keep going, maybe bug does not have such bad consequences
  - Correct answer depends on context!
- Ariane 5 – program halted because of overflow in unused value, exception thrown but not handled until top level, rocket crashes…

Teamwork & debugging summary

- Work on the part of the project that excites you
- Make sure all necessary jobs are covered

- Do your best to
  - prevent errors in design
  - think hard before you write code
  - code to make bugs visible fast