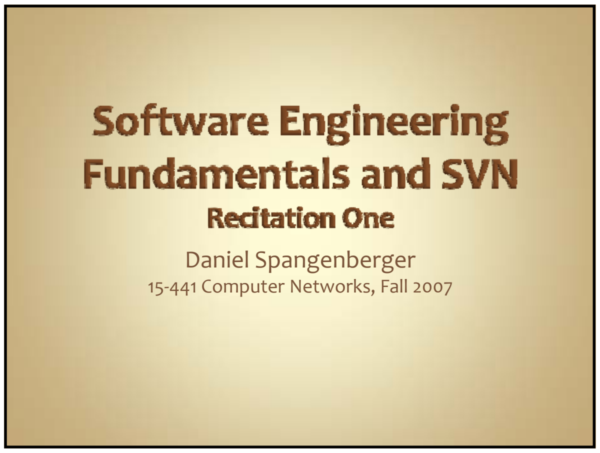
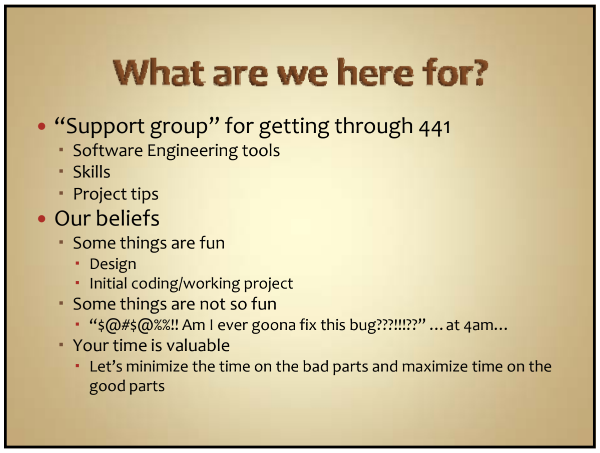


Software Engineering Fundamentals and SVN

Recitation One

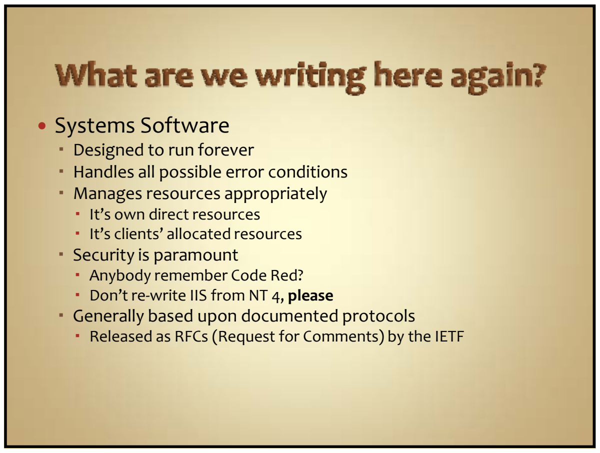
Daniel Spangenberger

15-441 Computer Networks, Fall 2007



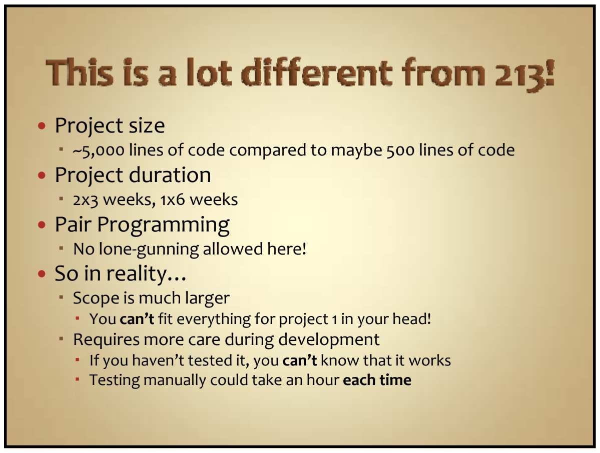
What are we here for?

- "Support group" for getting through 441
  - Software Engineering tools
  - Skills
  - Project tips
- Our beliefs
  - Some things are fun
    - Design
    - Initial coding/working project
  - Some things are not so fun
    - "$@#$@%%!! Am I ever goona fix this bug???!!!??" … at 4am…
  - Your time is valuable
    - Let's minimize the time on the bad parts and maximize time on the good parts



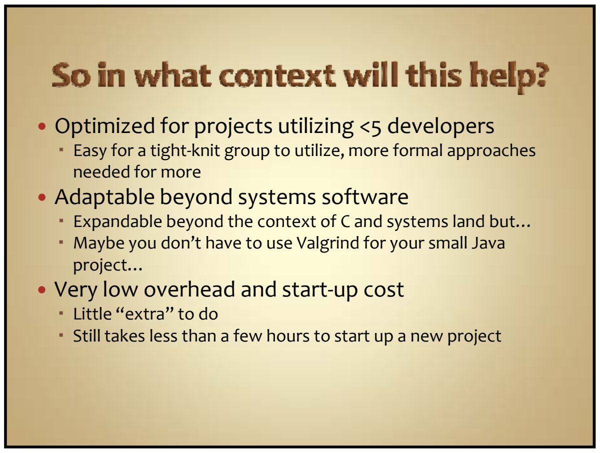
What are we writing here again?

- Systems Software
  - Designed to run forever
  - Handles all possible error conditions
  - Manages resources appropriately
    - It's own direct resources
    - It's clients' allocated resources
  - Security is paramount
    - Anybody remember Code Red?
    - Don't re-write IIS from NT 4, **please**
  - Generally based upon documented protocols
    - Released as RFCs (Request for Comments) by the IETF



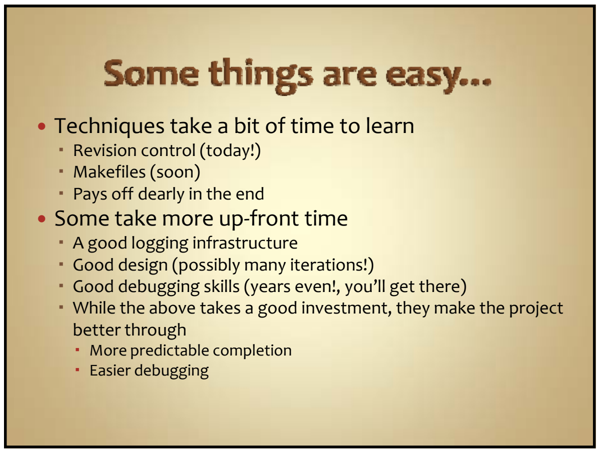
This is a lot different from 213!

- Project size
  - ~5,000 lines of code compared to maybe 500 lines of code
- Project duration
  - 2x3 weeks, 1x6 weeks
- Pair Programming
  - No lone-gunning allowed here!
- So in reality…
  - Scope is much larger
    - You **can't** fit everything for project 1 in your head!
  - Requires more care during development
    - If you haven't tested it, you **can't** know that it works
    - Testing manually could take an hour **each time**



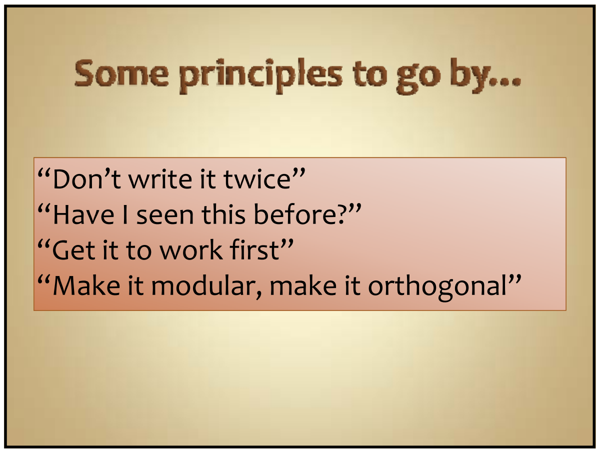
So in what context will this help?

- Optimized for projects utilizing <5 developers
  - Easy for a tight-knit group to utilize, more formal approaches needed for more
- Adaptable beyond systems software
  - Expandable beyond the context of C and systems land but…
  - Maybe you don't have to use Valgrind for your small Java project…
- Very low overhead and start-up cost
  - Little "extra" to do
  - Still takes less than a few hours to start up a new project



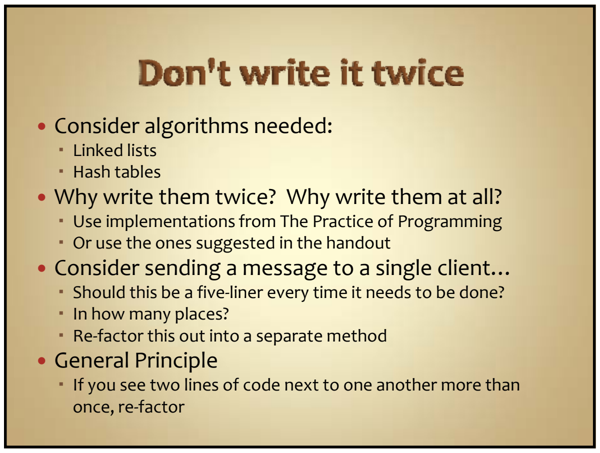
Some things are easy…

- Techniques take a bit of time to learn
  - Revision control (today!)
  - Makefiles (soon)
  - Pays off dearly in the end
- Some take more up-front time
  - A good logging infrastructure
  - Good design (possibly many iterations!)
  - Good debugging skills (years even!, you'll get there)
  - While the above takes a good investment, they make the project better through
    - More predictable completion
    - Easier debugging



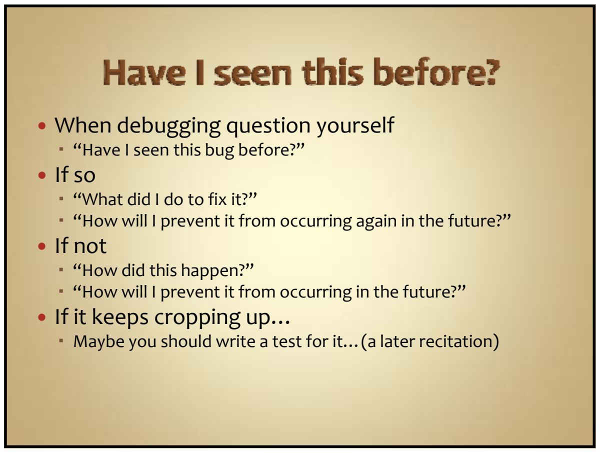
Some principles to go by…

"Don't write it twice"
"Have I seen this before?"
"Get it to work first"
"Make it modular, make it orthogonal"



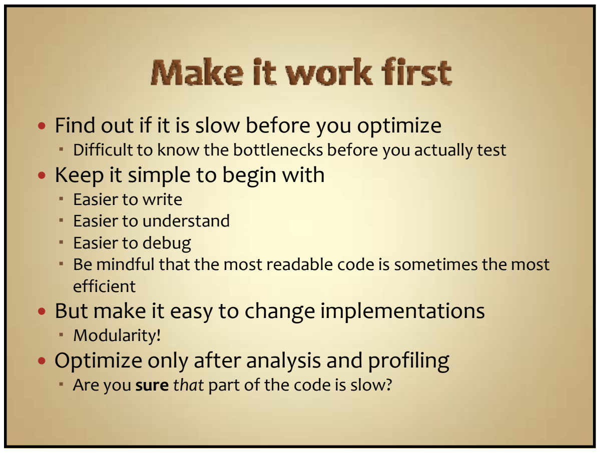
Don't write it twice

- Consider algorithms needed:
  - Linked lists
  - Hash tables
- Why write them twice? Why write them at all?
  - Use implementations from The Practice of Programming
  - Or use the ones suggested in the handout
- Consider sending a message to a single client…
  - Should this be a five-liner every time it needs to be done?
  - In how many places?
  - Re-factor this out into a separate method
- General Principle
  - If you see two lines of code next to one another more than once, re-factor



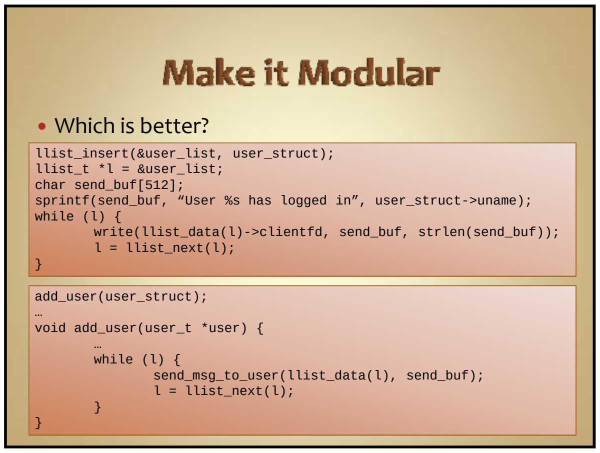
Have I seen this before?

- When debugging question yourself
  - "Have I seen this bug before?"
- If so
  - "What did I do to fix it?"
  - "How will I prevent it from occurring again in the future?"
- If not
  - "How did this happen?"
  - "How will I prevent it from occurring in the future?"
- If it keeps cropping up…
  - Maybe you should write a test for it…(a later recitation)



Make it work first

- Find out if it is slow before you optimize
  - Difficult to know the bottlenecks before you actually test
- Keep it simple to begin with
  - Easier to write
  - Easier to understand
  - Easier to debug
  - Be mindful that the most readable code is sometimes the most efficient
- But make it easy to change implementations
  - Modularity!
- Optimize only after analysis and profiling
  - Are you **sure** *that* part of the code is slow?

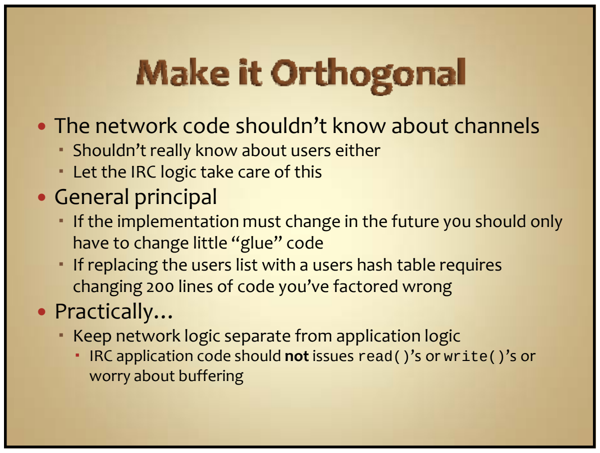


Make it Modular

- Which is better?

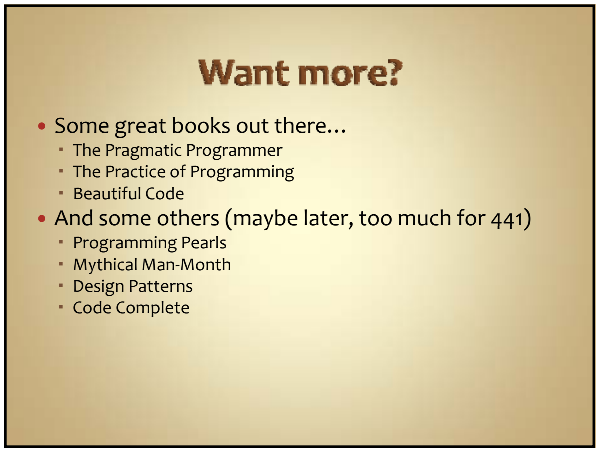
```
llist_insert(&user_list, user_struct);
llist_t *l = &user_list;
char send_buf[512];
sprintf(send_buf, “User %s has logged in”, user_struct->uname);
while (l) {
        write(llist_data(l)->clientfd, send_buf, strlen(send_buf));
        l = llist_next(l);
}
```

```
add_user(user_struct);
…
void add_user(user_t *user) {
        …
        while (l) {
                send_msg_to_user(llist_data(l), send_buf);
                l = llist_next(l);
        }
}
```



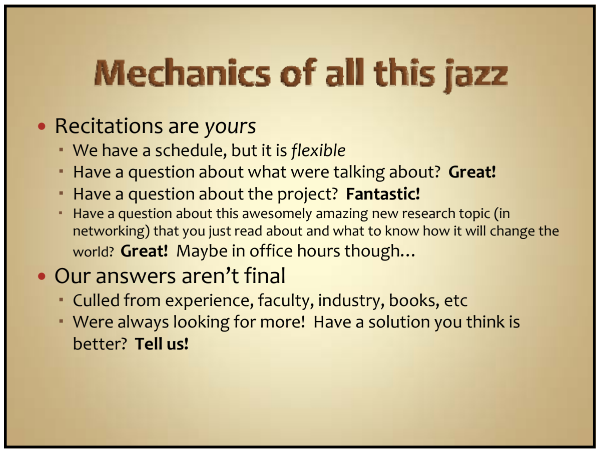
Make it Orthogonal

- The network code shouldn't know about channels
  - Shouldn't really know about users either
  - Let the IRC logic take care of this
- General principal
  - If the implementation must change in the future you should only have to change little "glue" code
  - If replacing the users list with a users hash table requires changing 200 lines of code you've factored wrong
- Practically…
  - Keep network logic separate from application logic
    - IRC application code should **not** issues `read()`'s or `write()`'s or worry about buffering



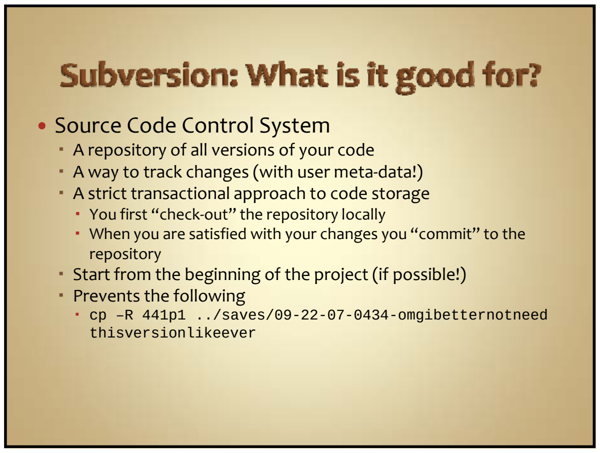
Want more?

- Some great books out there…
  - The Pragmatic Programmer
  - The Practice of Programming
  - Beautiful Code
- And some others (maybe later, too much for 441)
  - Programming Pearls
  - Mythical Man-Month
  - Design Patterns
  - Code Complete



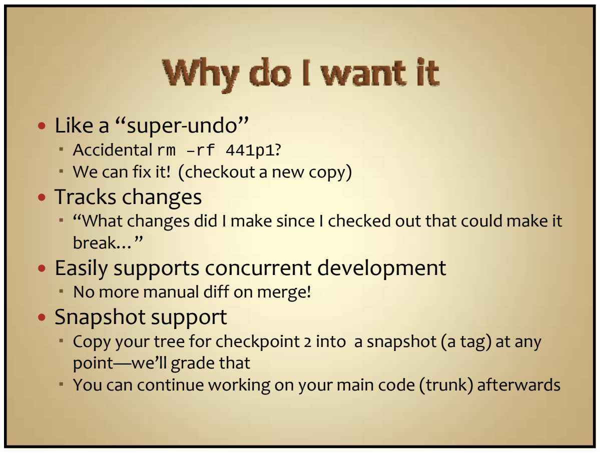
Mechanics of all this jazz

- Recitations are *yours*
  - We have a schedule, but it is *flexible*
  - Have a question about what were talking about? **Great!**
  - Have a question about the project? **Fantastic!**
  - Have a question about this awesomely amazing new research topic (in networking) that you just read about and what to know how it will change the world? **Great!** Maybe in office hours though…
- Our answers aren't final
  - Culled from experience, faculty, industry, books, etc
  - Were always looking for more! Have a solution you think is better? **Tell us!**



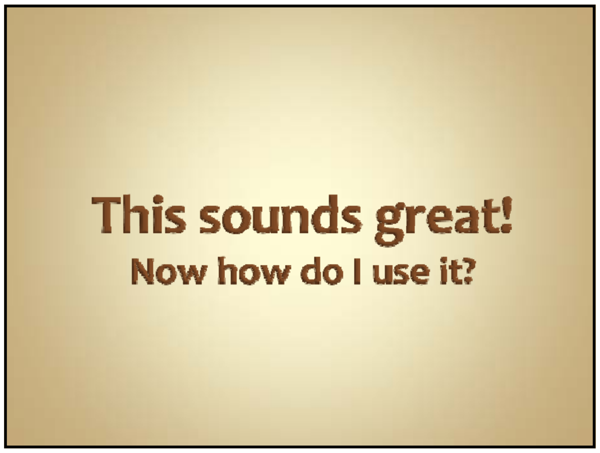
Subversion: What is it good for?

- Source Code Control System
  - A repository of all versions of your code
  - A way to track changes (with user meta-data!)
  - A strict transactional approach to code storage
    - You first "check-out" the repository locally
    - When you are satisfied with your changes you "commit" to the repository
  - Start from the beginning of the project (if possible!)
  - Prevents the following
    - `cp –R 441p1 ../saves/09-22-07-0434-omgibetternotneed thisversionlikeever`



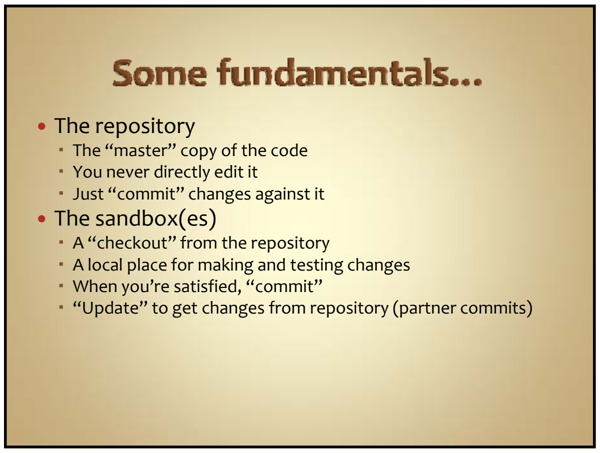
Why do I want it

- Like a "super-undo"
  - Accidental `rm –rf 441p1`?
  - We can fix it! (checkout a new copy)
- Tracks changes
  - "What changes did I make since I checked out that could make it break…"
- Easily supports concurrent development
  - No more manual diff on merge!
- Snapshot support
  - Copy your tree for checkpoint 2 into a snapshot (a tag) at any point—we'll grade that
  - You can continue working on your main code (trunk) afterwards



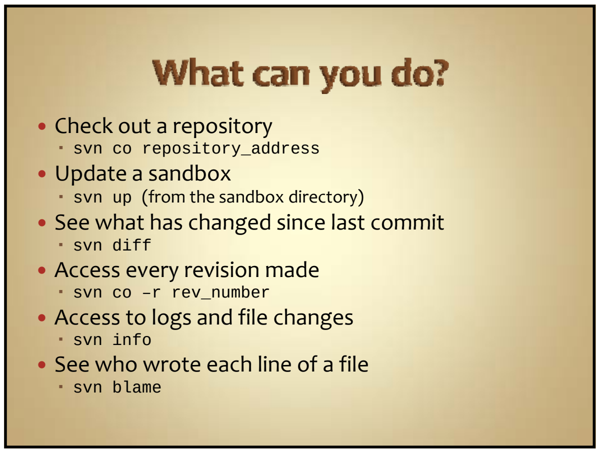
This sounds great!

Now how do I use it?



Some fundamentals…

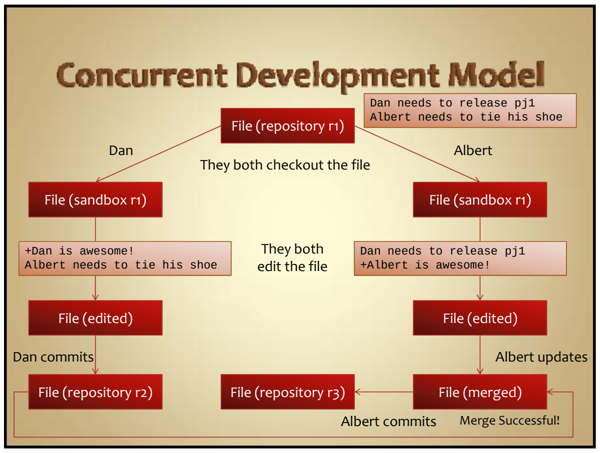
- The repository
  - The "master" copy of the code
  - You never directly edit it
  - Just "commit" changes against it
- The sandbox(es)
  - A "checkout" from the repository
  - A local place for making and testing changes
  - When you're satisfied, "commit"
  - "Update" to get changes from repository (partner commits)



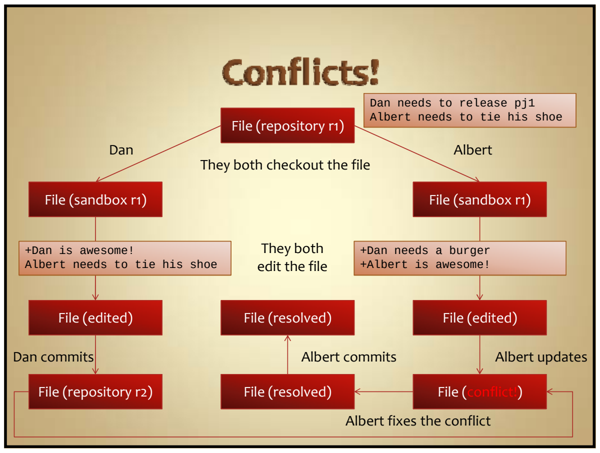
What can you do?

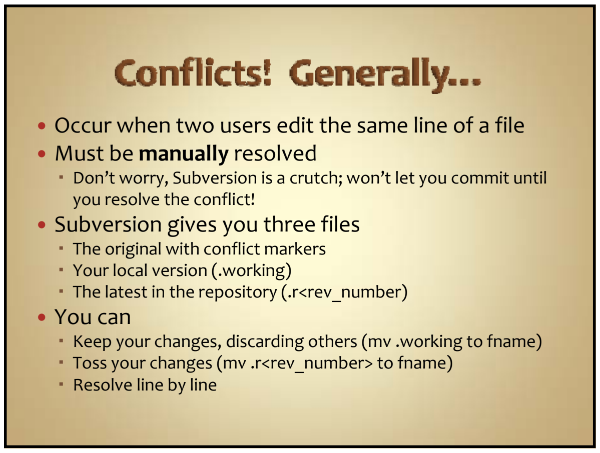
- Check out a repository
  - `svn co repository_address`
- Update a sandbox
  - `svn up` (from the sandbox directory)
- See what has changed since last commit
  - `svn diff`
- Access every revision made
  - `svn co –r rev_number`
- Access to logs and file changes
  - `svn info`
- See who wrote each line of a file
  - `svn blame`

Concurrent Development Model



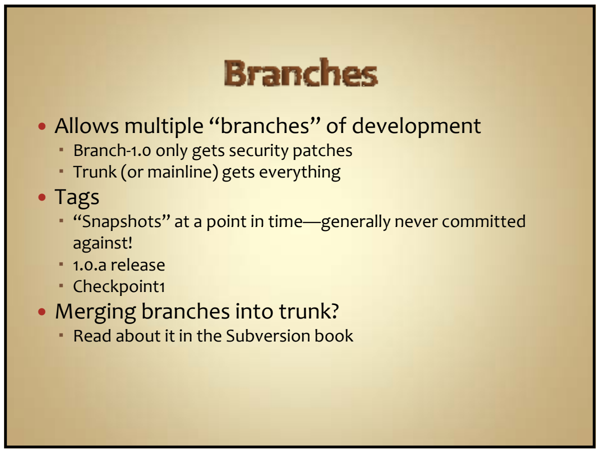
Conflicts!





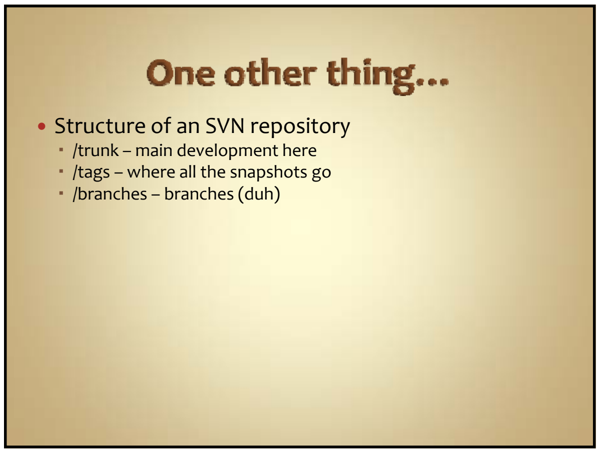
Conflicts! Generally…

- Occur when two users edit the same line of a file
- Must be **manually** resolved
  - Don't worry, Subversion is a crutch; won't let you commit until you resolve the conflict!
- Subversion gives you three files
  - The original with conflict markers
  - Your local version (.working)
  - The latest in the repository (.r<rev_number)
- You can
  - Keep your changes, discarding others (mv .working to fname)
  - Toss your changes (mv .r<rev_number> to fname)
  - Resolve line by line



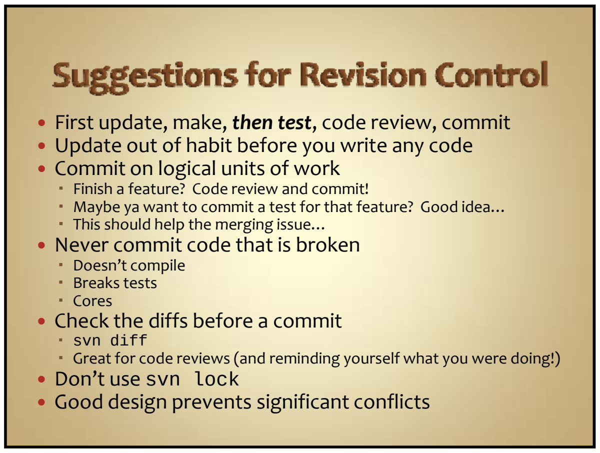
Branches

- Allows multiple "branches" of development
  - Branch-1.0 only gets security patches
  - Trunk (or mainline) gets everything
- Tags
  - "Snapshots" at a point in time—generally never committed against!
  - 1.0.a release
  - Checkpoint1
- Merging branches into trunk?
  - Read about it in the Subversion book



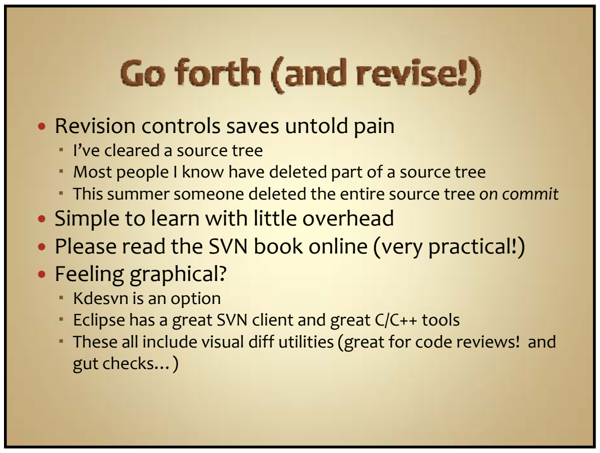
One other thing…

- Structure of an SVN repository
  - /trunk – main development here
  - /tags – where all the snapshots go
  - /branches – branches (duh)



Suggestions for Revision Control

- First update, make, ***then test***, code review, commit
- Update out of habit before you write any code
- Commit on logical units of work
  - Finish a feature? Code review and commit!
  - Maybe ya want to commit a test for that feature? Good idea…
  - This should help the merging issue…
- Never commit code that is broken
  - Doesn't compile
  - Breaks tests
  - Cores
- Check the diffs before a commit
  - `svn diff`
  - Great for code reviews (and reminding yourself what you were doing!)
- Don't use `svn lock`
- Good design prevents significant conflicts



Go forth (and revise!)

- Revision controls saves untold pain
  - I've cleared a source tree
  - Most people I know have deleted part of a source tree
  - This summer someone deleted the entire source tree *on commit*
- Simple to learn with little overhead
- Please read the SVN book online (very practical!)
- Feeling graphical?
  - Kdesvn is an option
  - Eclipse has a great SVN client and great C/C++ tools
  - These all include visual diff utilities (great for code reviews! and gut checks…)