CLIs, SCM, and VCSs (?!?)

CS 100: Introduction to the Profession

Michael Lee

ILLINOIS TECH | College of Computing

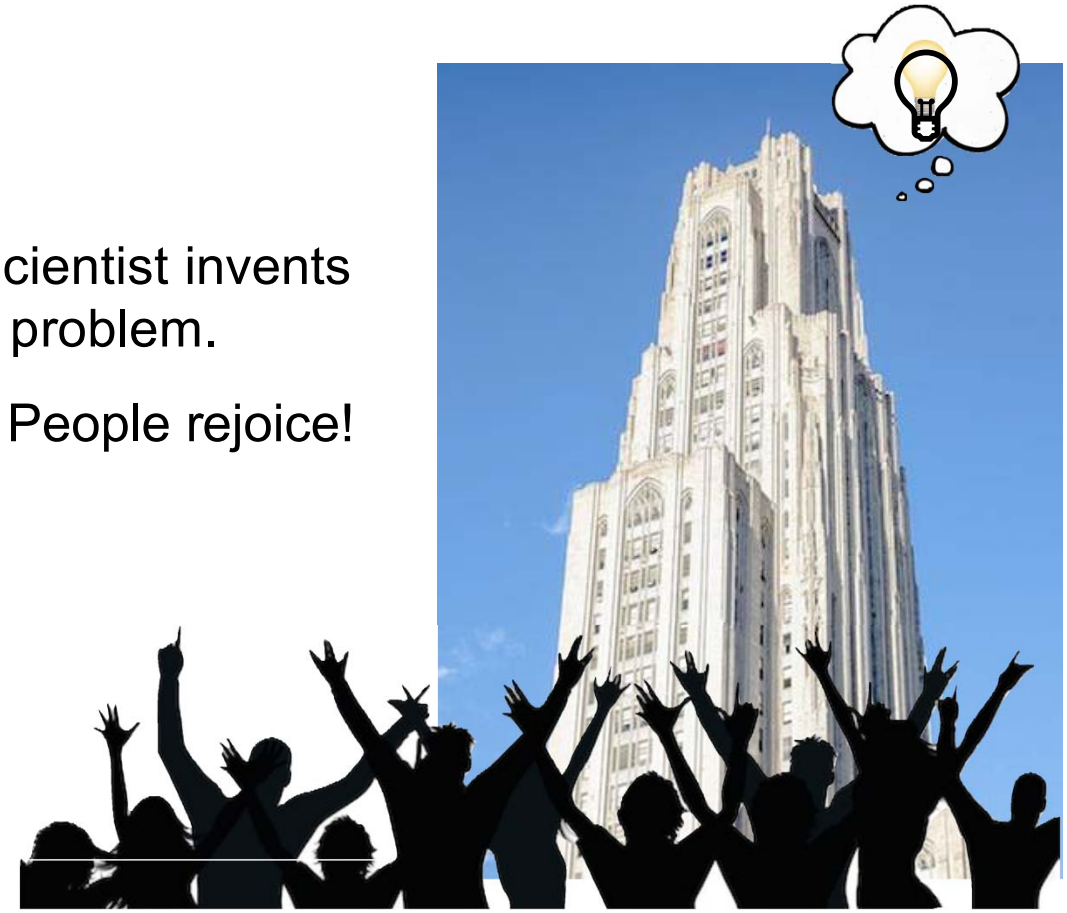
Agenda

- Development: Ideal vs. Reality
- Command Line Interfaces (**CLI**s)
- Software Configuration Management (**SCM**)
- Version Control Systems (**VCS**s)

§ 1. Development: Ideal vs. Reality

Ideal

- Step 1: Theoretical computer scientist invents a new algorithm to solve some problem.
- Step 2: The problem is solved. People rejoice!



Reality

- The algorithm needs to be implemented first
- The implementation of the algorithm is likely a **tiny piece** of the puzzle!

Software Engineering …

- Will likely require tons of supporting code (thousands/millions of LOC) written by many programmers
  - To connect to data sources and storage
  - To work with existing devices (I/O)
  - To interface with users
  - Etc.

Massive Complexity!

- Much of software engineering is about **managing complexity**
- We'll explore computing models for managing complexity in the future
- This lecture, we will explore some practical tools for doing so

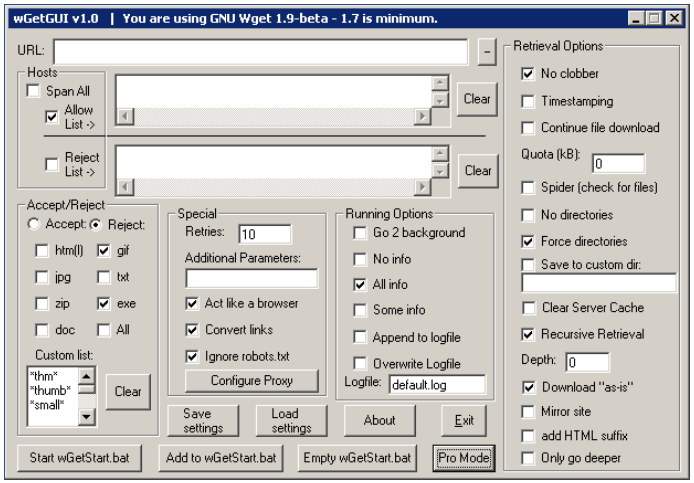
§ 2. Command Line Interfaces (CLIs)

Graphical User Interfaces (GUIs)

- Menus, windows, buttons, etc.
- Primary UI for Windows / Mac / Android / iOS / web apps / etc.
- Intuitive (?) — or at least relatively approachable
- But, takes a lot of work to build!
  - On top of underlying functionality

UI/UX and HCI

- Many overlapping fields dedicated to understanding “good” design
  - Modern HCI draws from psychology, neuroscience, etc.
- The more complex the software, the harder to make an intuitive UI!
  - Balance between surfacing features and ease-of-use



Alternative: Plain Text

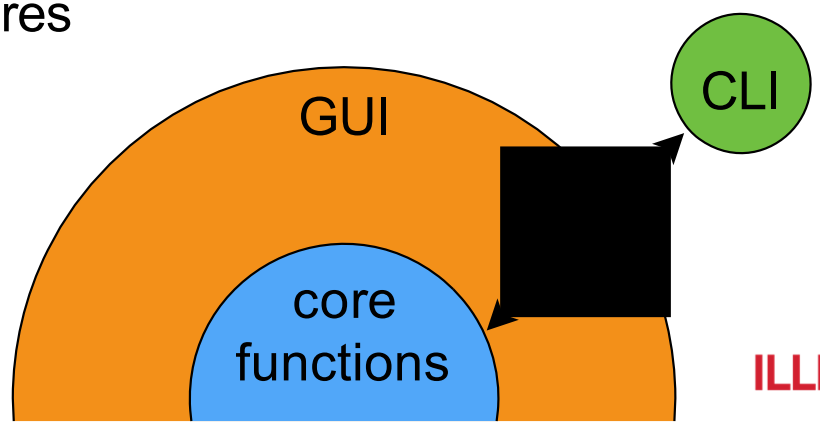
- Any modern language supports straightforward character-based I/O
  - And make it easy to process "command-line arguments”

```
public class App {
    public static void main(String[] args) {
        for (String arg : args) {
            System.out.println(arg);
        }
    }
}
```

```
> java App fold my laundry
fold
my
laundry
```

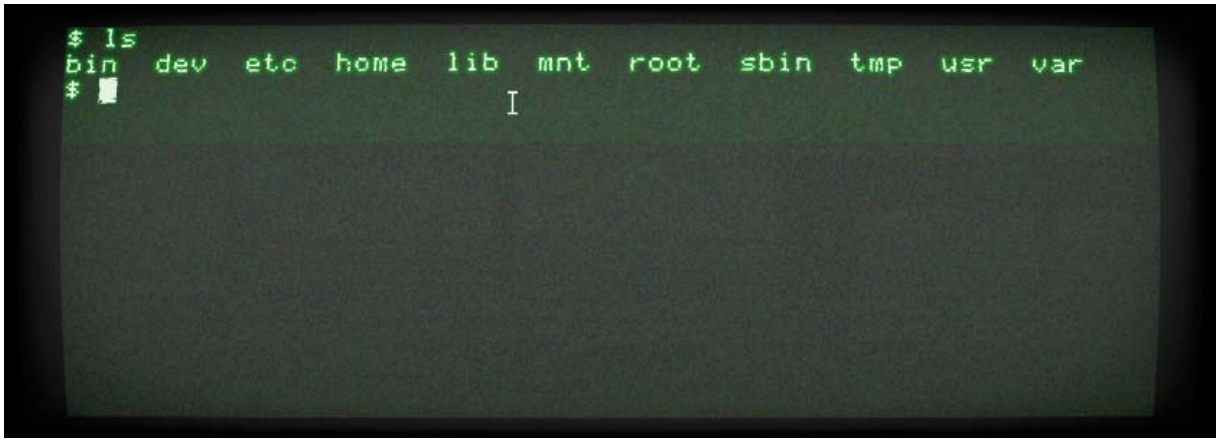
Alternative: Plain Text

- Low barrier to entry, and trivial to support new “commands”
  - vs. needing to build new menus / buttons / etc. into GUI
- When new features are added to software, can be easily surfaced via CLI
  - May take time to trickle up into GUI
  - “Power user” features



CLIs

- The OG UI
  - aka “REPL” — Read-Eval-Print Loop
  - aka “Shell”



UNIX Shells

- UNIX = family of operating systems dating back to 1960s
  - GNU Linux is a modern UNIX
- Shell = low-level CLI for the OS
  - Encapsulates functions provided by the OS and included utilities
- Popular modern shells:
  - sh, bash, tsh, zsh, fish

§ 3. Software Configuration Management (SCM)

A Typical Project

- Multiple developers with independent dev machines / environments
- Source code in one or more languages
- Many pre-built libraries with interdependencies
- One or more deployment targets

Essential Issues

- Devs must be able to use and contribute to shared project resources
- Code must be built/tested regularly, in a consistent environment
- Libraries must be synchronized and set up correctly (dependency hell)
- All deployment targets must be set up and tested

Some HARD Problems

- What if dev machines/envs are very different? (e.g., Mac vs. Windows)
- What if devs independently modify the same files?
- How to make sure devs are all using the same libraries / versions?
- Who runs the tests? When? How to communicate results?
- How do we know the whole project builds and will work on the deployment targets?

SCM

- Vast category of tools and processes for **configuring & managing** software projects
- “Meta” development — everything besides actually writing the code
  - Many buzzwords, e.g., “DevOps”
- Some “developers” (e.g., project managers) ONLY work with these tools

SCM Subcategories

- Build automation
- Test automation
- Version control
- Continuous integration
- Virtual environments
- Containers / Virtual Machines / Emulators

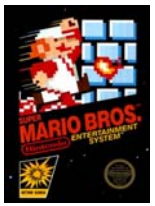
Jenkins

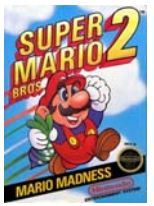
Focus on VCSs

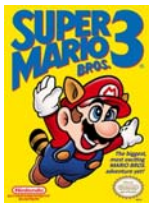
- You will use a tool from every SCM category eventually
- VCS is one of the most important to learn early on!
  - Used by professors for assignment distribution/submission
  - Allow you to check out and contribute to open source projects
  - Helps you work on extracurricular programming projects
  - Vital industry tool! (Companies need you to know how to use them)

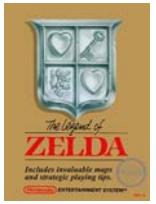
§ 4. Version Control Systems (VCSs)

Analogy: game save slots









- 1985-90: Super Mario Bros 1-3
  - 5-10 hours of gameplay; **no savepoints**
- 1986: Legend of Zelda
  - First console game with saves! Limited save points/slots
- 1989: Phantasy Star II
  - 250 hours of gameplay! Only save at Inns; few slots
- Mid 90s and onwards:
  - Lucky, lucky, lucky kidz

Scenarios

- Save progress so as not to start all over on reboot / power loss
- Checkpoint before boss fight to try different approaches & cut losses
- Save at scenic points, cool milestones, pre-cut-scenes, etc.
  - Log progress, show to friends, revisit (really?)

Nitpicks

- How can I tell where a save was performed? (Save-file hell)
- How can I tell what happened between saves?
- Why can’t I “edit” a previous save and have it affect my current game?
- Why can’t I start a new game and merge in stuff with other saves?
  - “Achievements” are a kind of hack for this
- Why can’t I cooperatively play with friends and merge our saves?
  - What happens if someone screws up?

The ultimate save system

- Saves show up on a clear timeline, where context and differences between saves are clearly displayed
- “Branches” for experimenting with different paths through the game
  - Can just drop a branch or “merge” it in to another
    - E.g., work on a side quest in one branch, then merge into main branch after completing it
- Collaborative (multiplayer) support (how?!)
- Curated game-save collections that I can download / explore / update

We invented the VCS!

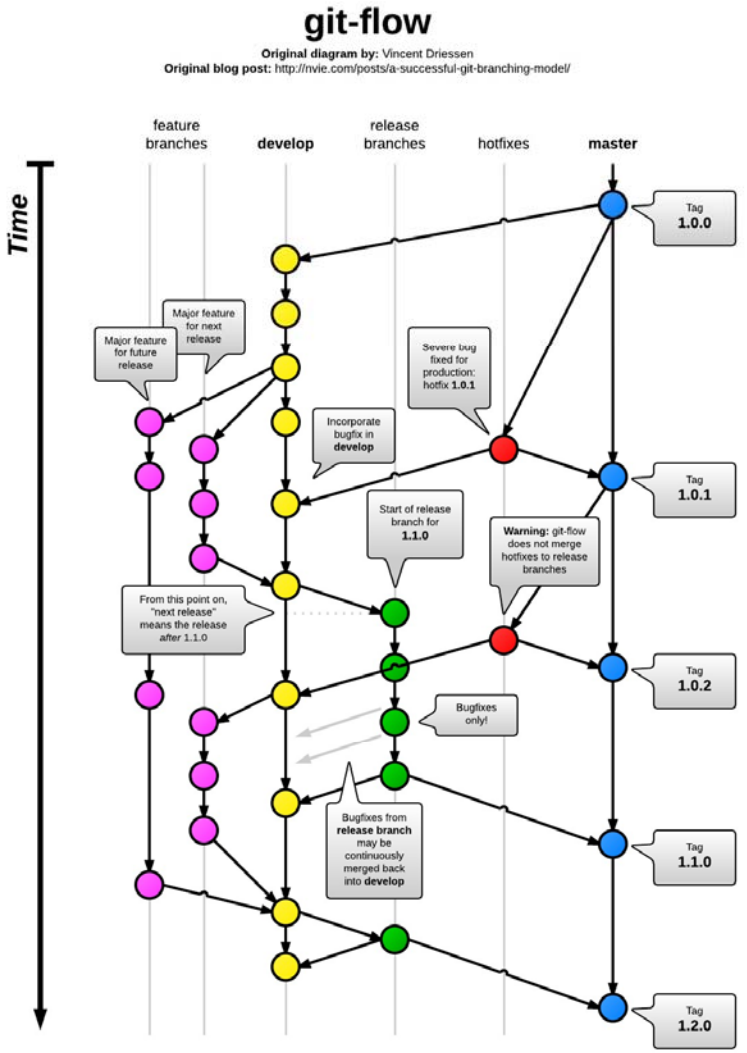
- Version Control Systems track changes over time made to a set of files by one or more authors
  - A collection of changes is called a **version** or revision
    - Associated with the author who **commit**ted it, along with a **timestamp** and **log message**

Modern VCSs

- Early systems: CVS, Subversion
  - Centralized design: made sense with all authors committing to a server housing a single “repository”
- Current systems: Git, Mercurial
  - Distributed design: every author has their own clone of the repository
    - Push/Pull changes to remote repositories
      - Project maintainer(s) manage the authoritative repository

VCS workflow

- Developers agree on conventions for committing and working on branches
- Typically involve keeping a pristine “mainline” branch
- E.g., Git workflow



§ Summary