Week 1 - Wednesday

COMP 3100

Last time

- What did we talk about last time?
- Course overview
- Began introduction to software engineering

Questions?

Back to Software Engineering

Questions when building software products

- What exactly should it do? What if people disagree?
- How does this product fit into the rest of the stuff the company does?
- How will users interact with the product?
- What parts should the product have?
- What languages should it be written in?
- What standards should we use to write it?
- How do we know if the program does what it's supposed to?
- How much time and money will it take to make it?
- What kind of documentation will it need?
- How will it change in the future?
- How far along are we in the process of making it?

More problems for large products

- The requirements themselves are huge
- The designs are large and complicated
- The code is long
- Testing gets harder because there's more to go wrong
- More people are on the project
  - Tracking progress gets harder
  - Communication gets harder
  - More managers are needed

History of software engineering

It was on one of my journeys between the EDSAC room and the punching equipment that … the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.

– Maurice Wilkes, Winner of the 1967 Turing Award

- Electronic computers were born in World War II
- Back then, getting the hardware to work was nearly impossible
- Programs were short by necessity
- In the 1950s, computers got much more powerful, allowing more complex programs
  - The field split into computer engineering (hardware) and computer science (software)
- By the late 1960s, software development projects with large numbers of people were running into problems
- Since then, people have worked hard on improving software engineering, but we still don't really know what we're doing

State of software development

- Most kinds of engineers have certification requirements
  - But not software engineers … yet
- In 2012, a survey of software projects found:
  - 39% were on time and on budget with expected features
  - 43% were late, over budget, or missing features
  - 18% totally failed
- Another study found that large IT projects were 66% over budget 33% of the time
- A series of interviews suggested that 75% of executives expected their software projects to fail

Sources of problems

- Software project managers haven't been educated in software engineering or project management
  - That's why you're here now
- Projects are underfunded and understaffed because managers are unwilling
- Well-planned, well-run projects can fail for technical reasons
  - Programming is still hard

Management

Management

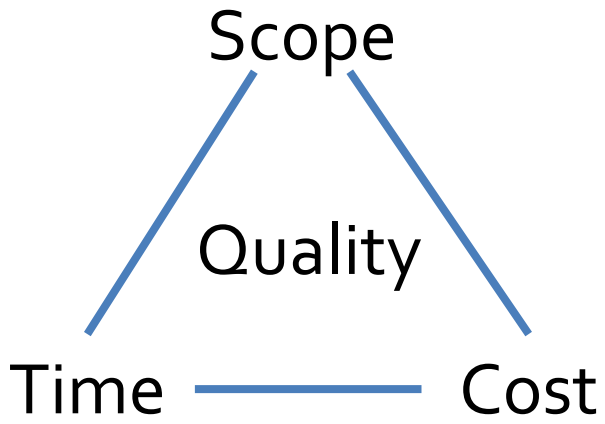
- A **project** is when you try to achieve a goal under time or money constraints
- **Management** is assembling, directing, and supporting human resources to do projects
  - People, at least Americans, don't like the idea of management
  - But it's impossible to do big projects without it

Aspects of a project that must be managed

- **Scope**
  - How much the project is trying to accomplish
  - **Creep** is the tendency for the work to increase
- **Time**
  - Must be reasonable for the project size
  - Must be monitored
- **Cost**
  - Similar issues as with time
- **Quality**
  - How good is acceptable?
  - Quality assurance must be done through the project, not just at the end
- **Resources**
  - Do you have the people (and tools) to get the job done?
- **Risks**
  - Have you planned for things going wrong?

Project management iron triangle

- There's a graphical depiction of project management used imply relationships between time, scope, cost, and quality



- This triangle is intended to indicate that you can't change scope, time, or cost without affecting the other two (at least if you want to maintain quality)
- Increasing scope means increasing time or cost (or both)
- It's obvious, but managers are sometimes tempted to push workers to work faster, for example, pretending there are no consequences

Software development methods

- Traditional methods
  - Careful planning and hierarchical leadership
  - Steps like requirement specification, design, implementation, testing, and maintenance
  - Example: Waterfall model
- Agile methods
  - Constant iteration
  - Self-directed teams
  - Minimal documentation
  - Example: Scrum
- Both methods are widely used and many successful teams use aspects of both
- The project for this class will mostly employ traditional methods because agile works best with experienced developers

Management

- Traditional methods rely on project managers while agile methods focus on self-directed teams
- Managers are responsible for:
  - **Planning**: Setting goals and requirements, deciding who does what when, estimating costs, etc.
  - **Execution**: Getting resources, training people, deciding processes
  - **Control**: Collecting and analyzing data and making adjustments
  - **Leading**: Motivating people, solving disagreements, supporting people

Requirements and design

- **Requirements** are functions or characteristics that software has
- **Customers** or **users** determine the requirements
- **Stakeholder** is a broad term that includes customers, users, developer, managers, and maybe the public
- **Designs** specify how the software system will meet the requirements
- Designs can look at a system from different aspects
- **Design patterns** are standard solutions to problems that have been useful in the past and can help structure designs

Implementation

- After the design is made, the software must be implemented in one or more programming languages
- **Compilers** and **interpreters** are used to run the programs
- **Editors** allow people to write code
- **Version control** tools let people track the evolution of the code
- **Code checkers** see if the code is meeting certain standards
- **Debuggers** help programmers find mistakes

Assuring quality

- Following certain practices can reduce defects
- Code reviews (literally looking through code) is one technique for finding defects
- Testing is another
  - Unit testing tools help us run test cases automatically
  - Code coverage tools make sure all parts of the program are being tested
- Integrated development environments (IDEs) combine tools for editing and testing code, doing version control, and lots of other things
  - IntelliJ IDEA
  - Eclipse
  - Xcode
  - Visual Studio
  - IDLE

Version Control

Version control

- For any large software project (and even small ones), it's valuable to have a way to track changes over time
- Such tools are called **version control systems**
- They allow:
  - Changes to be tracked over time
  - Developers to check code into repositories
  - Comparison of files over time
  - Documentation of changes made
- It's more than just a glorified backup system

Repository

- A **repository** is where all the development data is stored
  - Usually called **repos** by professionals
- Repositories include the current source code as well as a history of all the changes ever made
- For source code, most version control systems use **delta compression**, meaning that only the *differences* between files are stored
- Thus, hundreds of versions of your code can be stored without taking up hundreds of times the space

Actions

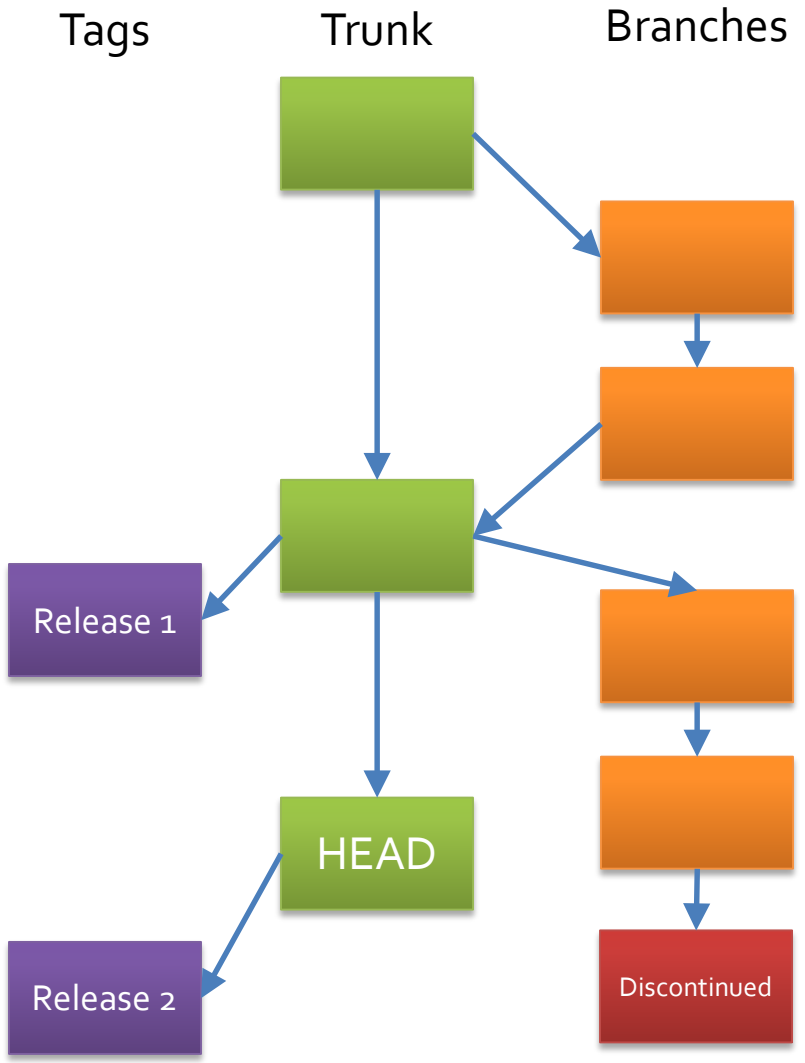
- **Committing** a file is adding its changes to a repository
- **Cloning** means creating a copy of another repository, including history
- **Merging** is combining two sets of files with independent changes into one set with changes from both
- **Pulling (or fetching)** copies the changes from an outside repository and adds them to the current repository
- **Pushing** copies the changes from the current repository to an outside repository

Merging

- Merging sucks
- You can try to avoid it, but ultimately, two people (or even yourself working on two different local copies) will make changes to the same files
- One of them will push their changes first
- The second will then have to merge
- Most systems are smart and can show those lines that conflict
  - If not, there are tools that can do that

Visualization of development

- Version control systems provide ways to organize the development process
- One such feature is a visualization of the development process
- The main sequence of development is called the **trunk**
- Code bases that diverge from main development (to work on a new feature) are **branches**
- **Tags** are snapshots of the code base in a particular state, often a release



Popular version control systems

- **Git**
  - Git is one of the most popular systems with a distributed model
- **SVN**
  - SVN is one of the most popular systems with a client-server model
- **Microsoft Team Foundation Server**
  - Microsoft always has to have its own thing
- **Mercurial**
  - Mercurial competes with Git as a distributed VCS
- **Perforce**
  - A big suite of tools that can do its own things or integrate with Git
- **CVS**
  - An old client-server tool that was popular until SVN overtook it

Git

History of Git

- Linus Torvalds, the creator of Linux, created Git in 2005
- Linux is a huge, distributed development project, and he needed a tool to organize the contributions people were making
- Torvalds hated CVS
  - He's a guy who hates a lot of things

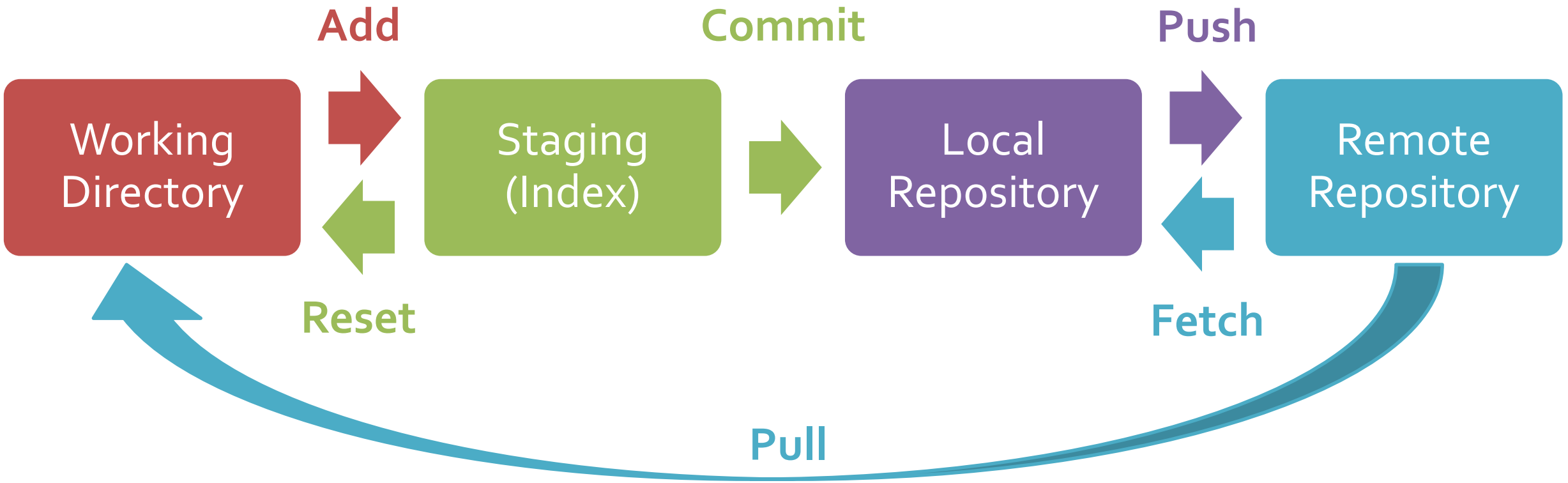
Philosophy of Git

- Git is a distributed VCS
- Every computer has a complete history of all the changes, ever
- There's no central server
- Programmers make changes and push them to or pull them from other repositories
- All operations are designed to be fast
- Torvalds did a pretty good job, but some common tasks are confusing

Operations in Git

- Once a repository is created in a directory, you can create files in that directory
- A file isn't tracked by Git until you **add** it
- A group of tracked files with changes can be **committed** to the local repository
- Then, you can choose to **push** those changes to another repository if you want other people to have them
- Sometimes, you don't want files to be tracked, so you can add their names (or wildcards like **`*.class`**) to a **`.gitignore`** file

Git process



It's also possible to reset to an earlier commit, overwriting the working directory, but it's confusing to put that arrow in.

Git cheatsheet

- There are many tools built on top of Git, but Git was designed for command-line operation
  - **`git init [project]`** — Create a new repo
  - **`git clone [url]`** — Copy a repo from the URL
  - **`git status`** — List all new and modified files
  - **`git add [file]`** — Stage the file to be committed
  - **`git commit -m "message"`** — Commit the staged files with the message
  - **`git fetch [bookmark]`** — Download changes from bookmarked repo
  - **`git merge [bookmark]/[branch]`** — Combine bookmarked repo into local branch
  - **`git pull`** — Fetch + merge
- There's a lot more, but that's enough to get yourself in trouble

Using Git with IntelliJ

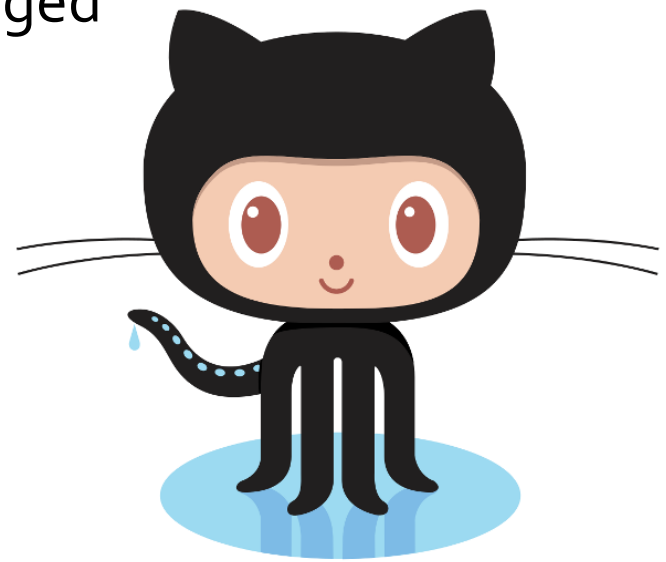
- IntelliJ has a GUI interface for Git
- It's easiest to use the **Get from Version Control** option when making a new project
  - If you're using GitHub, it's good to add your GitHub account to your IntelliJ
  - For plain-old Git, you can use the Repository URL
- No matter what you do, pay attention to where the files get stored
- Don't store stuff on lab computers since it'll get wiped
  - You can use OneDrive if you want to work on lab computers
- Once you've got the project set up in IntelliJ, you can use its built-in tool to talk to the GitHub repo
- The initial set-up is annoying, but then it's not bad
- Manage all files through IntelliJ, **not** by doing uploads through GitHub

Resources for Git

- Git is confusing!
- Don't be afraid to ask me questions
- But you can also Google
- There are some reasonably good videos to introduce Git:
  - https://www.youtube.com/watch?v=USjZcfj8yxE
  - https://www.youtube.com/watch?v=HVsySz-h9r4
- Git cheat sheet: https://www.jrebel.com/system/files/git-cheat-sheet.pdf

GitHub

- GitHub.com provides online repositories for code
  - Private repositories (except for education) are **not** free
  - Public repositories are free
- Git can be used without GitHub
- GitHub can even be used without Git (since it has support for SVN)
- GitHub has nice tools for:
  - Visualizing who's committing and how much they have changed
  - Issue tracking
  - Writing commit information and Read Me files
  - Pushing and pulling repos stored on GitHub
  - Creating webpages related to releasing software
- Ironically, Linus Torvalds hates GitHub



Upcoming

Next time…

- Friday is our first work day
- We'll get teams nailed down
- We'll get everyone added to GitHub
  - It's not a bad idea to make a GitHub account if you haven't already
- Hopefully, you'll pick your projects

Reminders

- Read Chapter 5: Software Product Requirements for Monday
- Start working on your projects!