CS 5150: Software Engineering

Schedule

Date	Lecture topic	Materials	Assignments
Jan 25	Lecture 1: Introduction Syllabus, projects, context	• Handout	• Read about the FAA AAS • Submit A1: Counting lines of code • Start forming teams around projects
Jan 27	Lecture 2: Projects and process Stakeholders, risk, development methodologies	• Handout	• Read *Software Engineering at Google*, Chapter 2: How to Work Well on Teams • Continue forming teams around projects
Feb 1	Lecture 3: Teams Team roles, developer strengths and growth, meetings, collaboration tools	• Handout	• Continue forming teams around projects
Feb 3	Lecture 4: Planning Feasibility, scheduling	• Handout	• Submit A2: First steps with Git • Finish forming teams around projects • Meet with clients • Start defining project plan
Feb 8	Lecture 5: Requirements I V&V, documentation, traceability	• Handout	• Meet with clients • Continue defining project plan
Feb 10	Lecture 6: Requirements II, orientation Elicitation, scenarios, use cases, code navigation	• Handout	• Finish writing project plan • Optional reading
Feb 11	**Project plan** due (11:59 PM EST)		
Feb 15	Lecture 7: Code forges & version control GitHub tools, kanban boards, centralized vs. distributed VCS, branch management, commits	• Handout	• Read any article on writing Git commit messages (e.g. How to Write a Git Commit Message) • Read *Software Engineering at Google*, Chapter 9: Code Review • Optional reading
Feb 17	Lecture 8: Code review Benefits of review, inspections, author & reviewer guidelines	• Handout	• Complete A3: Cleaning up commit history
Feb 22	Lecture 9: Models UML, use cases, sequence diagrams, data flow, state charts	• Handout	• Read Chapter 9: ITK in The Architecture of Open Source Applications, Volume II • Optional reading
Feb 24	Lecture 10: Architecture Deployment diagrams, component diagrams, architectural styles	• Handout • Slides	• Work towards report #2 • Optional reading
Mar 1	February break		

Date	Lecture topic	Materials	Assignments
Mar 3	Lecture 11: Architecture II Three tier, MVC, virtualization	• Handout • Slides	• Complete report #2
Mar 4	**Report #2** due (11:59 PM EST)		
Mar 8	Lecture 12: Presentations, user experience I Focus groups, UI prototypes	• Handout	• Submit peer evaluations • Schedule midpoint presentation
Mar 10	Lecture 13: User experience II Design principles, responsive design	• Handout	• Schedule/prepare midpoint presentation
Mar 15	Lecture 14: User testing, code tracing	• Slides	• Prepare/present midpoint presentation
Mar 17	Lecture 15: Retrospectives, web application frameworks Team retrospectives, Django, ORM, entity-relation models, CRUD, routes, templates, logging	• Slides	• Review midpoint scores on CMSX • Prepare/present midpoint presentation
Mar 22	Lecture 16: Program design UML class diagrams, UML sequence diagrams, design patterns, Observer	• Handout	• Read *Software Engineering at Google*, Chapter 8: Style Guides and Rules • Prepare/present midpoint presentation • Work towards report #3
Mar 24	Lecture 17: Design patterns, implementation Builder, Factory method, RAII, Visitor, Façade, static analysis, style	• Slides	• Prepare/present midpoint presentation • Complete report #3 • Submit peer evaluations for session 3
Mar 25	**Report #3** due (11:59 PM EST) Last day for **midpoint presentation**		
Mar 29	Lecture 18: Testing I Portability, test coverage, test style/scope/size	• Slides	• Read *Software Engineering at Google*, Chapter 11: Testing Overview • Optional reading • Submit peer evaluations for session 3
Mar 31	Lecture 19: Testing II Unit tests, test doubles, continuous integration	• Slides	• Optional reading
Apr 5, 7	Spring break		
Apr 12	Lecture 20: Testing III Continuous integration, dynamic analysis, debuggers, Valgrind, fuzz testing, performance, latency vs. throughput, Amdahl's Law	• Slides	
Apr 14	Lecture 21: Build systems Profiling, soak testing, task vs. artifact targets, Makefiles, Bazel	• Slides	• Read *Software Engineering at Google*, Chapter 21: Dependency Management • Complete report #4 • Submit peer evaluations for session 4
Apr 15	**Report #4** due (11:59 PM EST)		
Apr 19	Lecture 22: Dependency management Dependency networks, vendoring, SemVer, supply chain vulnerabilities	• Slides	• Submit peer evaluations for session 4
Apr 21	~~Lecture 23~~ In-class exam	• Sample questions	

Date	Lecture topic	Materials	Assignments
Apr 26	Lecture 24: Delivery Release management, continuous delivery, feature flags	• Slides	• Work on A4: Dependency analysis
Apr 28	Lecture 25: Licenses and legal considerations Law, contracts, copyright, open source licensing, copyleft, patents, export restrictions, privacy	• Slides	• Submit A4: Dependency analysis
May 3	Lecture 26: Non-functional properties I: security Access control, communications security, injection attacks	• Slides	• Prepare final presentation
May 3	Lecture 27: Non-functional properties II: dependability & safety Software integrity levels, hardware reliability, watchdog timers	• Slides	• Present final presentation
May 10	Lecture 28: Professionalism Responsible disclosure, ethics, diversity, project failures, conclusion	• Slides	• Complete final report and handoff package • Submit peer evaluations for session 5 • Submit course evaluation
May 10	**Final report** due (11:59 PM EST)		

Reading assignments

Lecture 1

Read about the Federal Aviation Administration's Advanced Automation System project. Think about what aspects of the *software development process* led to it being a runaway failure.

1. Federal Aviation Administration (FAA) Advanced Automation System (AAS). Guide to the Systems Engineering Body of Knowledge.
2. The Ugly History of Tool Development at the FAA. Edward Cone. *Baseline*. 2002-04-09.
3. [optional] Why (Some) Large Computer Projects Fail. Robert N. Britcher. *Software Runaways*. 1998.

Follow-up questions

1. Identify the *developer*, *client*, *customer*, and *user* for this project.
2. Which process steps were handled poorly for this project?
3. Which of the general methodologies discussed in lecture most closely matches that used by this project? Was it an appropriate choice?

Lecture 2

Read *Software Engineering at Google*, Chapter 2: How to Work Well on Teams.

Lecture 6

For some examples of real requirements documents (from Sommerville's *Software Engineering*), optionally read:

- Mentcare requirements document
- Insulin pump requirements specification

Lecture 7

To learn best practices for writing commit messages, read any article on the subject (e.g. How to Write a Git Commit Message). These conventions are well-established, and work on your projects is expected to conform to them.

In preparation for next lecture, read *Software Engineering at Google*, Chapter 9: Code Review.

To learn more about version control, optionally read:

- *Software Engineering at Google*, Chapter 16: Version Control and Branch Management
- Why Google Stores Billions of Lines of Code in a Single Repository

Follow-up questions

1. Which of these is a good commit subject?
2. How should you respond to this proposed change in a review?

Lecture 9

A nice reference for (informal) design documentation for familiar software systems is the two volumes of The Architecture of Open Source Applications. To familiarize yourself with this resource (and witness a variety of model diagrams in context), please read Chapter 9: ITK in Volume II.

Note that much of the content in these volumes (including most of the diagrams in the above chapter on ITK) focuses on program design and development guidelines rather than system-level architecture. For some familiar architectural styles (including a layered view and a dataflow view), check out Chapter 7: GPSD in the same volume.

Lecture 10

For more examples of software architectural styles, optionally read:

- David Garlan and Mary Shaw, "An Introduction to Software Architecture" (1994)
- Ian Sommerville, *Software Engineering, Tenth Edition* – Presentations, Chapter 6

Lecture 18

Read *Software Engineering at Google*, Chapter 11: Testing Overview.

For examples of well-written style guides, optionally read the Google Style Guides for your preferred languages.

For an example of how to thoroughly test difficult corner cases in critical software, optionally read How SQLite Is Tested.

Lecture 19

For more information about brittleness, mocking, integration testing, and CI for Internet-scale applications, optionally read the following chapters in *Software Engineering at Google*:

- Chapter 12: Unit Testing
- Chapter 13: Test Doubles
- Chapter 14: Larger Testing
- Chapter 23: Continuous Integration

For a more traditional take on testing styles and formal test plans, optionally read *Better Embedded System Software*, Chapter 23: Testing and Test Plans.

Canvas assignments

A1: Counting lines of code

1. Install a tool for counting lines of code (recommendation: scc or tokei).
2. Choose a programming project or assignment that you implemented in the past (the larger the better, but at most two authors). Count the number of lines of code in the project and its test cases and answer the corresponding questions.
3. Choose a major application (whose source code is available) that you have used or heard of before (examples: Blender, Unreal Engine, Visual Studio Code, Firefox; do not use the example from lecture). Download its source code (if you have not used Git before, now would be a good time to install it and learn how to "clone" a repository).
4. Count the number of lines of code in the application and its test cases and answer the corresponding questions.
5. Respond to the reflection question.

Do not worry about being perfectly precise; a rough estimate is fine for the quantitative questions.

Questions

1. What is the origin of your personal project? (e.g. CS 2112 assignment, independent research, hobby project, hackathon, etc.)
2. How many lines of code were written for your project?
3. What fraction of those lines are code comments?
4. What fraction of those line correspond to tests?
5. What application did you choose to analyze?
6. How many lines of code make up that application?
7. How many of those lines are code comments or documentation? Note: some projects have a long license comment at the top of every source code file. Consider how much that might skew the results and how you could correct for it.
8. How many of those lines correspond to tests?
9. How does the proportion of documentation & test code differ between your personal projects and major applications? What do you think justifies the difference (or explains the lack of difference)? [2-3 sentences]

A2: First steps with Git

Fluency with version control tools like Git is essential in modern software engineering. And most CS 5150 projects will be managed within Cornell's GitHub server. This assignment ensures that you are able to use these tools.

At a high level, your task is to clone a repository, make and test a change to its code, and commit your change.

Procedure

1. **Log into Cornell's GitHub server** at https://github.coecis.cornell.edu/ using your Cornell credentials. This will establish an account for you on the server (allowing course staff to form teams and share repositories with you).
2. **Install a Git client**. Git support may be built into your IDE, but you will still need to be able to access Git's command line interface in order to perform the more advanced repository manipulations that will be required in this course. You must **configure your name and e-mail address** (use your Cornell e-mail), and you must be able to authenticate with the Cornell GitHub server (using SSH keys, for instance). If you have never done this before, follow the instructions at https://docs.github.com/en/enterprise-server@3.3/get-started/quickstart/set-up-git (remember to perform these steps on Cornell's Enterprise GitHub server, not on the public "github.com").
3. **Clone the "cs5150-sp22/scrambler" repository** from https://github.coecis.cornell.edu/cs5150-sp22/scrambler; e.g. `git clone git@github.coecis.cornell.edu:cs5150-sp22/scrambler.git`.
4. Choose your preferred language (C++, Java, or Python). **Compile, run, and test** the version of the "scrambler" code in that language. If you have a preferred IDE (e.g. NetBeans, VS Code, Eclipse, PyCharm), try to configure it to perform these tasks via its graphical interface. If you like to work on the command line, the README offers some tips.
5. **Modify the code** to use your NetID (instead of the instructor's) as the key in the scrambler application. Ensure that the application still compiles and runs. **Commit your change** to your Git repository; e.g. `git commit -a -m "Change hard-coded key to author's NetID"` (you do not need to "push" you change—you don't have permission yet).
6. **View your repository's log** (e.g. `git log --name-status`, or install and run `gitk`). **Confirm** that you only committed changes to the intended file(s) and that your name and e-mail address are correct. **Copy the hash of your commit**.
7. **Run the scrambler application** with an argument of "cs5150". **Copy the output**.
8. Complete this Canvas quiz.

If you have trouble with any of these steps, first try diagnosing the issue with your project teammates. If the whole team is stumped, escalate to Ed Discussion. TA and instructor office hours are available as well (though we each specialize in different operating systems).

Questions

1. Which version of the Scrambler application (C++, Java, or Python) did you choose to test and modify? (You may of course play with all three versions—it is informative to compare similar code in multiple languages. But you only need to report on one of them for this assignment.)
2. What development environment (e.g. NetBeans, VS Code, Eclipse, PyCharm) did you use to explore this version of the application? Answer "Command line" if you did not use an IDE or graphical editor.
3. When you ran the application's unit tests, how many tests were run (as reported by the test runner)?
4. What is the Git hash for the change you committed to your repository?
5. What is the output when scrambling the text "cs5150", using your NetID as the key?

A3: Cleaning up commit history

This assignment gives you practice with more advanced Git operations, including rebasing and conflict resolution. Your objective is to rewrite a branch's history so that it is suitable for merging to the trunk branch and is compatible with a one-way merge topology.

Procedure

You will continue to work on the `scrambler` repository you cloned for A2. Back in that assignment, you made a commit directly on the trunk; now that you know more about branch management, let's put that commit in its own feature branch and restore `master` to its original state:

1. Ensure that you have no pending modifications by running `git status`. Commit or discard any changes to tracked files.
2. Create a new branch (here named "a2") capturing the current state of your repository: `git checkout -b a2`.
3. Set your active branch back to trunk: `git checkout master`.
4. Reset your local "master" branch to match its original state: `git reset --hard origin/master`.
5. New commits have been added to the scrambler repository since you cloned it for A2. Update your copy by running `git pull`.

Now your clone should match the central server again, but your custom work for A2 was not lost—it still exists in your "a2" branch. Time to start working on A3, and this time you'll use a feature branch right away:

1. Checkout a local copy of the "assignment3" branch: `git checkout assignment3`. This branch represents some work-in-progress that you will be cleaning up.
2. Copy this state to a personal feature branch whose name matches your NetID: `git checkout -b <netid>` (your branch name must match your NetID, all lower case, in order to get credit for this assignment).
3. Compare the contents of this branch to trunk by running `git log` and `git log master` in side-by-side terminals (or `gitk` and `gitk master` for a graphical view). Note that your branch contains three commits not on master, while master

contains one new commit not in your branch.

Now for the cleanup operation. You need to resolve any conflicts with the current state of trunk and recompose the new work so that each commit represents a single logical change. In the end, your branch should consist of two commits on top of the latest trunk. As you perform each step, think about how the states of your **working copy**, **branch**, and **staging area** (aka “index”) are changing. Here is one procedure for achieving this (you will need to learn the details of these commands yourself; the free book Pro Git is a good place to start):

1. Rebase your commit on top of master and resolve all conflicts without losing any important bug-fixes. This will likely involve commands like `git rebase`, `git add`, and `git rebase --continue`, as well as modifying source code in your editor. Check that you achieved what you intended using `gitk`.
2. Split the top commit into two: one that adds the tests, and one that improves the usage message. This can be done with `git reset` and `git commit`. Write appropriate commit messages for any new commits. Again, check your work with `gitk`.
   For this assignment, the changes that belong in separate commits are conveniently in separate files. But in the future, if you wanted to split up changes that were made in the same file, you would use `git add -p`.
3. Squash the three commits that relate to refactoring the utility functions while keeping the usage commit separate. Use `git rebase -i`. As always, write appropriate commit messages and check your work.
4. If you have a C++ development environment with Boost.Test, run `make test` to ensure that all tests pass.

Finally, submit your cleanup by pushing your feature branch to GitHub: `git push -u origin <netid>`. You do not need to create a pull request.

A4: Dependency analysis

Modern software is often built on top of hundreds of dependencies, many of them brought in transitively. And with a trend towards fine-grained libraries, the web of trust underpinning your software’s reliability becomes vast. In this assignment, you will get a sense of the scale of the issue and will practice using some of the databases that assist developers and administrators in managing supply chain vulnerabilities.

1. Choose a moderately-large software application with at least a half-dozen direct dependencies (and presumably twice as many or more transitive dependencies). You may select your team project if you like (but if you do, try to identify different vulnerabilities from your teammates).
2. Identify the application’s direct dependencies. These may be configured in the build system or listed in the developer documentation. Note that web applications may list separate dependencies for their frontend and backend (you should count both), and that some projects have additional test or CI dependencies (you only need to worry about runtime dependencies).
3. Construct a reasonably-complete dependency network for the application and count the number of distinct dependencies (both direct and transitive). This will be easiest if dependencies are managed by the build tool. (Note that Bazel does not automatically import transitive dependencies, so a Bazel project's direct dependencies should cover the whole dependency network.)
4. Identify at least one security vulnerability with a CVE affecting your application’s dependency network. This may include the application’s runtime platform (e.g. JVM or Python interpreter).
   There are tools that integrate with some build systems (and artifact mirrors) to automatically identify vulnerabilities, e.g. OWASP (Java: Ant/Maven/Gradle/SBT), or you can search the National Vulnerability Database, or even download a list of all CVEs and write a script to search offline.
5. Determine the severity of the vulnerability (see the CVSS score in the NVD) and whether or not your current deployment of the application is affected.

Questions

1. What application did you choose to analyze?
2. List the direct dependencies you identified for this application.
3. How many total (direct + transitive), distinct dependencies did you identify in the application's runtime dependency network?
4. Which CVE did you identify as affecting one of your application's dependencies?
5. Which dependency was affected by this vulnerability?
6. What is the severity (e.g. CVSS 3.x score) of this vulnerability?
7. Is your current deployment of this application affected by this vulnerability? If not, does it predate the vulnerability or include a fix?