20875 Software Engineering

Tutorial 4

First git project

1. If not already done, configure the name and email to be used in your commits. Configure your preferred editor to be used by git as well.
2. Create a directory called "`test_project`".
3. Inside the directory "`test_project`", create a git repository.
4. Add a file called "`main.c`" containing:

   ```
   #include <stdio.h>

   int main()
   {
           printf("Hello, world!\n");
           return 0;
   }
   ```

   and a "`Makefile`" (make sure to use TAB characters for indentation):

   ```
   main: main.o
           clang -Wall -O3 -o $(@) $(^)

   main.o: main.c
           clang -Wall -O3 -c -o $(@) $(<)
   ```

5. Build the executable "`main`". Create a "`.gitignore`" file that ignores that executable and any file ending in "`.o`".
6. Stage the source files ("`main.c`", "`Makefile`", "`.gitignore`") for a commit. Check your staged area with "`git status`".
7. Create the first commit for "`test_project`".

Second git project

You and your team (Alice, Bob, Carol, Dan, Eve) need to solve a series of optimization problem instances. To that end, you use the GNU MathProg language. However, writing many similar GNU MathProg files is tedious, and it would be convenient to write them using Python code. Your team decides to write a small utility module that makes this easier.

1. Alice creates a git repository called `mathprog`. In it, she creates a file `mathprog.py`, in which she writes a first test case: A small model is stored into a Python string, then simply printed. Subsequently, Alice, leaves the project. Clone Alice's repository `https://www.poirrier.ca/git/mathprog.git` and read her code.

2. Bob emails you. Starting from Alice's test case, he implemented a `MathProg` class, which calls `glpsol` to solve the instance passed as a string parameter. Bob did not create a branch, but says you can fetch his commit `79dcd1` in his repository `https://www.poirrier.ca/git/bob.git`. Review the code introduced by Bob.

3. Carol and Eve are tasked with reviewing and improving Bob's code. After reviewing Bob's work, Carol finds that Alice's initial test is no longer sufficient, a decides to generalize it. She creates a branch `carol_branch` for her code. Fetch `carol_branch` from her repository `https://www.poirrier.ca/git/carol.git` into a local branch. For simplicity, we will call this local branch `carol_branch` locally as well. Switch to it, review Carol's code. Then, merge that code into your `main` branch.

4. After committing her new test, Carol realizes that there is a small logic bug in Bob's code. When the parameter `nosolve` is `False` (the default), the `MathProg` does *not* solve the problem. The double negation may have been confusing, but it should be the opposite: when `nosolve` is `False`, the problem should be solved. Fix the issue, then create a commit for this code change, still on the `main` branch.

5. Dan observes that `glpsol`'s output mingles with the Python script's output, which may be confusing. He decides to indent `glpsol`'s output to separate it visually from the rest. Fetch Dan's code on his branch `dan_branch` at `https://www.poirrier.ca/git/dan.git`. Review Dan's contribution. Rebase it on top of your commit on the `main` branch, then merge it.

6. Meanwhile, Eve implemented a parser for `glpsol`'s output. As part of her work, she created a class `Array` that can hold multi-dimensional lists, and is used for storing solution values. The new class is also convenient for MathProg parameters, and she modifiess the test to take Fetch Eve's `eve_branch` at `https://www.poirrier.ca/git/eve.git`. Merge it onto the `main` branch, combining Eve's parser with Carol and Dan's improvements, as well as your bugfix. Resolve any conflicts that result.