Programming for Engineers Portfolio : Quarter 3, Set 3

These exercises are intended to directly follow the tasks from the third lab of Q3. If you haven't done the lab, you are strongly recommended to do so first, as it shows how to do all the tasks, and also puts it in a more realistic context: lab2 spec

Before starting, merge this repository (elec40004-2019-q3-portfolio-ytc19-master) into your private repository.

1 - Create two new classes ProxyA_Impl1.cpp and ProxyA_Impl2.cpp

Convert the two implementations of `ProxyA` in `ProxyA_Impl1.cpp` and `ProxyA_Impl2.cpp` into classes called `ProxyA_Impl1` and `ProxyA_Impl2`.

You should end up with two headers `ProxyA_Impl1.hpp` and `ProxyA_Impl2.hpp` containing the class declarations, and two sources `ProxyA_Impl1.cpp` and `ProxyA_Impl2.cpp` containing the definitions. Both `ProxyA_Impl1.cpp` and `ProxyA_Impl2.cpp` should be compilable, but will fail to link with an error about the missing `main`.

2 - Convert `ProxyA` into an abstract base class

Convert `ProxyA` into an abstract base class.

There should be three pure virtual functions: `fff`, `ggg`, and `hhh`. Its destructor should be a virtual function with an empty body. It is up to you whether you want to have data members in the base class or the derived class, though you may want too looked ahead to task 9.

3 - Make `ProxyA_Impl1` and `ProxyA_Impl2` inherit from `Proxy`

Make `ProxyA_Impl1` and `ProxyA_Impl2` inherit from `Proxy`.

4 - Explicitly use the `ProxyA_Impl1` class

Modify the source files `Stub1.cpp` and `Stub2.cpp` so that they explicitly use the `ProxyA_Impl1` class.

Hint: you should be able to compile both `Stub1` and `Stub2` at this point using:

```
$ g++ ProxyA_Impl1.cpp Stub1.cpp -o Stub1_Impl1
$ g++ ProxyA_Impl1.cpp Stub2.cpp -o Stub2_Impl1
```

Commit the current project using a message containing the string `s3t4ExplicitImpl1`.

5 - Declare a factory function for `ProxyA` implementations

Create a header file called `ProxyA_factory.hpp` containing a function with the following declaration:

```
/*Create an instance of ProxyA, using the given
  string to select the concrete type:
- 'Impl1' : Returns a new ProxyA_Impl1
- 'Impl2' : Returns a new ProxyA_Impl2
*/
ProxyA *ProxyA_factory(const string &type);
```

6 - Implement a factory function that returns an implementation

Create a source file called `ProxyA_factory.cpp` which provides a definition of `ProxyA_factory`, using the semantics described in the declaration.

7 - Modify the source files to use the factory function

Modify the `Stub1` and `Stub2` source files to use the result of `ProxyA_factory("Impl1")`.

Note: After releasing the lab I realised this is not testable, as it gets overwritten by the next commit. So there is no way to tell if you actually did it, and it ends up as a free pass.

8 - Make the implementation a run-time choice

Modify `Stub1` and `Stub2` so that the program takes a single argument describing which implementation to use.

Hint: After you have done this, you should be able to do the following:

```
$ g++ ProxyA_Impl1.cpp ProxyA_Impl2.cpp Stub1.cpp ProxyA_factory.cpp -o Stub1
$ g++ ProxyA_Impl1.cpp ProxyA_Impl2.cpp Stub2.cpp ProxyA_factory.cpp -o Stub2
$ ./Stub1 Impl1 > Stub1.Impl1.txt
$ ./Stub1 Impl2 > Stub1.Impl2.txt
$ ./Stub2 Impl1 > Stub2.Impl1.txt
$ ./Stub2 Impl2 > Stub2.Impl2.txt
```

Create a commit with the string `s3t8RunTimeImplementation`

9 - Merging common functions

The function `ggg` is shared between the two implementations. Move the `ggg` function into a common function on the base class, and remove from the derived classes.

Note: reading back through the lab it is a bit opaque, but in order to make this work you also need the shared variables back into the base class as protected members. Otherwise they can't be accessed in the base class.

10 - Create/update test scripts to test the two programs

Create a script called `compile_and_run_stubs.sh` which will:

1. Compile the two stub programs into `Stub1` and `Stub2`.
2. Run each stub with the `Impl1` and `Impl2` implementations and capture the output.
3. Compare the output against reference outputs added to the repository.

Commit the changes with a message including `s3t10ProxyDynamic`, and push to your private repo.

Submission

Updated as per lecture: the due date due just after the mid-term week: Mon 17th Feb at 22:00.

Submission of the portfolio is via github - the contents of your `private` portfolio repository will be pulled, and then the state of the repository at each of the tagged commit points will be examined.

There will also be a route for you to submit via blackboard as a backup, but the primary method is via github.

Sanity checks

Sanity checks will be performed at two points, which will: - Pull the currrent version of the repo - Check out the code at each of the named repo - Check that it compiles

The intended sanity check time-points are:

1. Wed 12th, 9:00

2. Thu 13th, 9:00
3. Fri 14th, 9:00

You don't need to do anything to take part; the current state of the repos will be pulled at those time, and at some point later the results of the check will be sent back.

Checking

It is recommended that you create a fresh clone of your repository, and then try checking out the named commit at each point. So the process would be (in a fresh clone):

1. Use `git log` to find the last commit containing the keyword.
2. Note the hash of the commit you want.
3. Use `git checkout HASH` to get back to that particular point, and check it looks as expected.

Tip: you can find all commits with a specific keyword using:

```
$ git log --all --grep KEYWORD
```

What if my history is messed up?

It's possible that as you went through the stages that you forgot to commit a particular file, or realised that there was a mistake. Source control doesn't really want you to go back in time and change things, as it is trying to keep a record of how the repository evolved.

The only thing that matters for assessment is the state of the files at each of the specified commit points. More specifically, it only cares about the *last* commit that contains the given keyword. The assessment will also be positivist - it only looks for files that it expects to be there, and ignores other files which might not exist.

So if you have an existing special commit that you don't like, any following commit with the same keyword will "override" it.

So if you find a mistake, you have two main ways that let you fix it:

1 - Create new commits on top of the old one

If you prepare exactly the state of the directory you *wanted* at a particular stage, you can simply commit again with the same message. This is the least destructive, and quite easy to do.

So for example, in this lab there are three commit checkpoints: `s3t4ExplicitImpl1`, `s3t8RunTimeImplementation`, and `s3t10ProxyDynamic`.

Let us imagine that you have a local copy of your repo in `~/q3-portfolio`, and want to fix-up the three commit checkpoints for just this set in some way. A reasonable approach is:

1. For each checkpoint KEYWORD in {`s3t4ExplicitImpl1`, `s3t8RunTimeImplementation`, and `s3t10ProxyDynamic`}:
    a. Use `git log --all --grep KEYWORD` to find the last commit hash with that keyword.
    b. Used `git checkout HASH` to move to that particular commit. Your repository should now reflect the state of the commit at that hash.
    c. Create a new directory `~/q3-portfolio/q3/s3/tmp/KEYWORD`
    d. Copy the contents of the `~/q3-portfolio/q3/s3` into `~/q3-portfolio/q3/s3/tmp/KEYWORD`
    e. Check and edit the files in `~/q3-portfolio/q3/s3/tmp/KEYWORD` until you get exactly the files you want, in the state you want.
2. Use `git checkout master` to get to the latest commit you made. If you do `git log` you should find that you are at the very latest commit again.
3. For each checkpoint KEYWORD in {`s3t4ExplicitImpl1`, `s3t8RunTimeImplementation`, and `s3t10ProxyDynamic`}:
    a. Copy the contents of `~/q3-portfolio/q3/s3/tmp/KEYWORD` back into `~/q3-portfolio/q3/s3`
    b. Add any files that were missed before using `git add`
    c. Create a new commit with a message including KEYWORD.

After this is complete, your last three commits should reflect exactly the file states that you wanted. The previous commits are still in there, but the later ones will override them.

2 - Force push

You can effectively start from scratch, by creating a fresh clone of the master repository, re-doing your work from the beginning (or copying in existing files for each stage), and then *force* pushing the new version up to your repo. This is considered more dangerous, as it will completely replace the history of the remote repository with your new repo's history - essentially you add a `--force` flag to git push.

If you take this approach, then make sure that you have a back-up of your private repository. Also be careful about the different directories for different sets - the repo considers all of of them to be part of the same history.

Once you're completely happy that your local repo is exactly what you want, you can use:

```
$ git push <remote-name> --force
```

Then switch to the github GUI and you should see the remote version is now the same as your local version.