Lab 05: ADTs, and more GIT

CMPT 145

Laboratory 05 Overview

Section 1: Pre-Lab Reading ▸ Slide 3
Section 2: Laboratory Activities ▸ Slide 26
Section 3: What to hand in ▸ Slide 47

Section 1

Pre-Lab Reading

A scenario that calls for multiple versions

Multiple versions are appropriate (1)

- Commercial scenario:
  - You are developing a large application, with lots of features.
  - You want a basic version with limited features, with a low price.
  - You want an advanced version with lots of bells and whistles, for a higher price.
- Many commercial software development projects are like this.

Multiple versions are appropriate (2)

- Development scenario:
    - You are developing a large application, with lots of plans for many features.
    - You plan to start with a simple version with very few features.
    - Each version of your application will add more and more features.
- This is the incremental and iterative development process.

Multiple versions are appropriate (3)

- Research scenario:
  - You have a data analysis script to answer a scientific question.
  - You realize that a slightly modified script could answer different scientific questions about a slightly different dataset.
  - You need multiple versions to address the different datasets.
- Many research projects have scripts created this way.

Example: The MM1 Queueing algorithm

- Did you ever wonder what would happen if the MM1 simulation used LIFO order instead of FIFO?
- Humans value fairness, and queues (FIFO) seem to be fair.
- Just how unfair would it be if customers were served in LIFO order?
- To answer, we could repeat the MM1 simulation using a Stack instead of a Queue.
    - This is an example of needing multiple versions for a research project.

A list of bad ideas

To answer the FIFO vs. LIFO question, we could do one of the following:

- Edit the MM1 program.
- Copy the MM1 program, and edit the copy.
- Edit the Queue ADT.

These are all terrible ideas!

Why editing is bad

- To answer the FIFO vs. LIFO question, we could edit the MM1 program.
- Changing a working program to have different behaviour is fine, as long as the old behaviour is no longer needed.
- In this experiment we want both behaviours:
  - MM1 with a FIFO queue (standard).
  - MM1 with a LIFO stack (experimental).
- Editing back and forth is a waste of programmers' time!

Why copying is bad

- To answer the FIFO vs. LIFO question, we could copy the MM1 program, and change the copy.
- Suppose there are errors you didn't notice before you copied.
    - Copying the program copies the bugs!
    - Twice as much code to fix!
- Suppose we want to add code to the MM1 program, say, to collect more data about wait times.
    - Copying the program forces us to modify both copies the same way.
    - Takes twice as long to make the changes.
- Having two copies of the same program means that you have twice as much code to worry about.

Why changing an ADT is bad

- We could edit the Queue ADT and change the code:
  - Make enqueue behave like Stack's push
  - Make dequeue behave like Stack's pop
- The would affect every program that already uses your Queue ADT!
- You'd have to change it back when you're done.
  - If you forget, scripts that were correct will have faults, and you might not remember why.

Summary of bad ideas for adaptable software

- Copying code is a bad idea.
  - More code means more errors, and more time debugging.
- Editing code repeatedly is a bad idea.
  - Wastes programmer time!
- Modifying an established ADT is a bad idea.
  - Changes the behaviour of every working application that uses it.

Two good ideas

- Abstraction: use an ADT to solve the problem.
- Version control: Keep two different versions.

ADTs for Adaptability

ADTs enhance Adaptability

- Software always changes!
- ADTs help software designers manage change and new demands
- We'll see how to use ADTs to make future changes easier!

Abstraction to the rescue!

- Stacks and Queues are similar, but not exactly the same.
- The similarity can be expressed by a Container ADT with the following operations.
  - Create a container
  - Add a value to the container
  - Remove a value from the container
  - Check the container's size, if it's empty
  - Peek at the upcoming value without removing it
- The Container ADT generalizes Stack and Queue.
- We'll create 2 different implementations of this ADT.

Creating an adapter ADT

- Here's one of the implementations: ContainerQ

```
# CMPT 145: ContainerQ
# Simple adapter for Queues
# documentation removed to conserve space on slides

import Queue as Queue

def create():
    return Queue.create()
def add(container,value):
    Queue.enqueue(container,value)
def remove(container):
    return Queue.dequeue(container)
def is_empty(container):
    return Queue.is_empty(container)
def size(container):
    return Queue.size(container)
def peek(container):
    return Queue.peek(container)
```

ContainerQ is an Adapter

- The ContainerQ operations call the Queue operations.
  - It does nothing else.
  - Makes the container behave like a FIFO Queue.
- We can edit the MM1 application so that it imports the Container ADT instead of the Queue ADT.
  - The code has changed, but the behaviour has not.
- Key idea: We can also create a ContainerS ADT, an adapter for the Stack ADT.
  - Makes the container behave like a LIFO Stack.
- ContainerQ and ContainerS have the same operations.

Abstraction helps

- When your app needs LIFO, import the Stack ADT.
- When your app needs FIFO, import the Queue ADT.
- When your app needs to swap between FIFO or LIFO, import a Container ADT.
- Swapping Queues and Stacks is now easy!
  - Just edit the `import` line!
- Creating the Container ADTs is an investment of time and effort, but it pays off in programmer time saved later.

Version Control for Multiple Versions

Version Control for Multiple Versions

- Software always changes!
- You may find you need two similar but distinct versions of your current project.
    - No, not in your first year assignments, but maybe in longer term projects (summer research, etc)
- We'll see how to use Version Control to manage multiple versions!

Version control and branching

- So far, we've used version control as sophisticated backup software.
- Git allows us to create multiple versions, called branches.
- When we create a branch, we are creating an exact copy of the whole project.
- When we make changes to a branch, the changes only affect that branch, not all branches.
- We can jump between branches at any time.
- We only see one branch at a time.

Version control to the rescue!

- We'll keep the FIFO queue version as the main version.
  - Git initializes the project creating a branch called master.
- We'll make a new branch, and edit the script to use a LIFO stack.
  - We'll name the new branch StackSim, to reflect its purpose.
- We can jump between these branches when we want to change behaviours.
  - To get the Queue simulation, we'll jump to the master branch.
  - To get the Stack simulation, we'll jump to the StackSim branch.

Version Control Terminology

commit (verb) to save the changes made.

commit (noun) the state of your files when you committed them.

branch (verb) create new copy of the files.

branch (noun) a set of files.

checkout branch-name (verb) to jump to the named branch.

checkout file-name (verb) to retrieve the named file from the most recent commit of the branch you are currently on.

master (noun) the default name for the initial branch created when git initializes a set of files.

Section 2

Laboratory Activities

ADTs for Adaptability: Activities

Activities Overview

This portion of the lab will have the following steps:

1. Create a new project for the MM1 simulation.
2. Setting simulation parameters.
3. Modifying MM1 to use ContainerQ, an abstraction for Queue ADT.
4. Create a new ContainerS ADT, an abstraction for Stack ADT.
5. Running the simulation with either ContainerS or ContainerQ.

Activity ADT Step 1: Preparation

- Download the following files from the Laboratory.
  - `MM1.py`
  - `Queue.py`
  - `Stack.py`
  - `Statistics.py`
  - `ContainerQ.py`
- Create a new project and add these files to it.

Activity ADT Step 2: Preparation

- Run `MM1.py` to be sure it's working. Use the following inputs:

`arrival_rate`:	1.9
`service_rate`:	2.0
`sim_length`:	100000

- We'll use those same settings for all of our experiments in today's lab.
- Run it a few times, and make note of the average values reported.

ACTIVITY ADT Step 3: Adapting ADTs

1. Open the file `ContainerQ.py`.
   - Its functions simply call the Queue ADT
   - `ContainerQ.py` adapts the Queue ADT.
2. Modify `MM1.py` so that it uses `ContainerQ.py`.
   - Important: Start with `import ContainerQ as Container`
   - Replace `Queue.enqueue()` with `Container.add()`
   - Replace `Queue.dequeue()` with `Container.remove()`
   - Replace all other Queue operations with Container operations.
3. Run `MM1.py` to be sure it's (still) working.
   - Use the same inputs as before, and be sure that it gives more or less the same output as before!

Activity ADT Step 4: A new container

1. Make a copy of `ContainerQ.py`, call it `ContainerS.py`
2. Modify all the functions in `ContainerS.py` so that it is an adapter for `Stack.py`
3. You'll need to change the import, and all 6 operations!
    - `ContainerS.create()` should call `Stack.create()`.
    - `ContainerS.add()` should call `Stack.push()`
    - Etc.

ACTIVITY ADT Step 5: Changing the simulation

- Modify `MM1.py` so that it uses `ContainerS.py`.
    - Hint: You should only need to change the import line!
    - Hint: `import ContainerS as Container`
- Run `MM1.py` a few times, and make note of the average values reported.

Summary: Adapting ADTs

- We have two adapters, ContainerS and ContainerQ.
  - Their operations have the same names, but different behaviours.
- We modified the MM1 script to use the Container operations.
- We can switch between the two behaviours by changing the import statement.

Version Control for Multiple Versions: Activities

Activities Overview

This portion of the lab will have the following steps:

1. Create a new project, and initialize version control (Git).
2. Create a new branch for the version using a stack.
3. Become familiar with jumping between branches.
4. Create a new branch for the version using a queue.
5. Working with two different versions.

Activity: GIT Step 1: New Project

- Create a new project, initialize version control, add the MM1 files
    - The original ones from Moodle, not the previous exercise.
- Make sure the script executes properly, using FIFO queues as given.
- Add a new file, called README.txt, with the following text:

```
This version of MM1 uses a Queue. If this were not a lab
exercise, I would write more information about it.
```

- When everything is working, commit the version!

Activity: GIT Step 2: A branch!

- Create a new branch, named StackSim.
- In PyCharm: VCS menu, GIT, Branches..., New Branch
    - Because it's an exact copy, everything will look exactly the same as before.
    - New Branch will be greyed out if you haven't committed your files yet!
- Change the MM1 script to use Stack instead of Queue.
    - Edit the MM1 script directly! Don't bother with adapters this time.
    - Don't be afraid, your original code is saved under the master branch.
    - Edit the README.txt file (replace Queue with Stack).
- When everything is working, commit the version.

Activity: GIT Step 3: Moving between branches

- To jump between branches, we use "checkout".
  - In PyCharm: VCS Menu, Git, Branches…, Local branches, master, Checkout.
  - Checkout will swap your files so that they are exactly as they were before you made the StackSim branch.
  - Note: If you have edited but have not committed a change, git will not allow you to complete the checkout.
- View the MM1 script, and see the Queue is being used again.
- Jump between the two versions a couple of times (using checkout as above), to see how git works.

ACTIVITY: GIT Step 4: Another branch

- Checkout the master branch again.
- Make a new branch, called QueueSim.
  - This will be an exact copy of the master branch, with a descriptive name.
  - The name will remind you of its purpose, and will be symmetric with the StackSim branch.
- Now you can jump between 3 branches: master, StackSim, QueueSim.
- You can run either version without editing your code!
- Your project folder is not cluttered with variations.

ACTIVITY: GIT Step 5: Tidying up QueueSim

- Checkout the QueueSim branch.
- It should have the Queue ADT, but does not need the Stack ADT.
- Remove the Stack ADT file from PyCharm project.
  - Don't worry, this will not affect any other branch!
- Before you go on, check that everything is still working.
- Commit!

ACTIVITY: GIT Step 6: Tidying up StackSim

- Checkout the StackSim branch.
- It should have the Stack ADT, but does not need the Queue ADT.
- Remove the Queue ADT file, from PyCharm project.
    - Don’t worry, this will not affect any other branch!
- Before you go on, check that everything is still working.
- Commit!

Git Summary

- We have two different versions of the MM1 script on separate branches.
- By checking out branches, we can jump between versions.
- In PyCharm, the bottom of the window indicates the branch name, and lets your checkout different branches quickly!

Git helps those who help themselves

- Git won't understand why you need versions, only that you have several.
- When you have a project under version control, it's important to:
  1. Make very good commit messages.
  2. Have external documentation to describe your versions.
- If you do not do these things, you'll forget and regret!

More advanced use of Git

- We used Git to create two branches for two distinct versions.
- Git's branches can be used in other ways.
  1. A feature branch can be created for each new feature you add to the application.
  2. A development branch can be created while you fix a tricky bug.
- To do these tasks, we need to learn a bit more in a later lab.
- Find reasons to practice branching!

ACTIVITY: Reflection

- Answer the following questions with a sentence or two; put your responses in your `lab05-transcript.txt` file.

1. Does the average waiting time increase when you use LIFO instead of FIFO in the MM1 simulation?
2. Does the maximum waiting time increase when you use LIFO instead of FIFO in the MM1 simulation?
3. For the ADT activities, using ContainerS and ContainerQ, how hard was it to switch between versions?
4. For the Git activities, using branches QueueSim and StackSim, how hard was it to switch between versions?
5. Which approach to multiple versions do you prefer? Explain why with a sentence or two.

Section 3

Hand In

What To Hand In

Hand in your `lab05-responses.txt` file containing:

1. Your answers to the reflections activity on Slide 46.
2. Three git logs, copy/pasted as follows:
    - Checkout master, run `git --nopager log`
    - Checkout QueueSim, run `git --nopager log`
    - Checkout StackSim, run `git --nopager log`