Advanced System Programming

Assignment 1 CAF

Mohammad Ameen Kassem : 324259191

Gregory Guterman : 345956502

Link to GitHub Repo

Task 1 : Project structure

asp-caf-assignment/

+– deployment/

| +– Dockerfile # Development environment setup

+– Makefile # Build and development commands

+– assignment/ # Assignment source

+– caf/ # Python CLI application

| +– pyproject.toml # Python package configuration

| +– caf/ # CLI source code

| +– __main__.py # Entry point

| +– cli.py # Command-line interface

| +– cli_commands.py # Command implementations

+– libcaf/ # Core C++ library

| +– CMakeLists.txt # CMake build configuration

| +– pyproject.toml # Python package configuration

| +– libcaf/ # Python interface and higher-level repo operations

| | +– constants.py # Constants and configuration

| | +– plumbing.py # Low-level repo operations

| | +– ref.py # Reference handling

| | +– repository.py # Repository management and high-level API

| +– src/ # C++ source code

| +– bind.cpp # Python bindings

| +– blob.h # Blob object definitions

| +– caf.cpp/h # Low-level C++ implementation

| +– commit.h # Commit object definitions

| +– hash_types.cpp/h # Hashing implementations

| +– object_io.cpp/h # Object I/O operations

| +– tree.h # Tree object definitions

| +– tree_record.h # Tree record structures

+– tests/ # Test suite

+– caf/ # CLI tests

+– libcaf/ # Core library tests

Description:

1. DockerFile: This file is crucial for setup of the project, using this file Docker image is created, and hence we can use a unified operating system to run the project; moreover, we can use a Docker container to build an actual release product. Also, all the requirements are installed in Docker, so they keep your computer clean, and you have all the up-to-date versions of Python libraries.

2. MakeFile: This file is a “manager” of the project, to build a Docker, to run it, and to run pytests, we use a Makefile. It is not common to have a makefile for a Python project, but since we have part of the project in C++, it’s crucial to have one.

3. caf/caf/__main__.py: This file is an entry point to the project; it gives us an option to run caf as an installed command as well as a Python module. It has a Python mechanism to have a “cpp-like” main function using __main__. After entry, it calls the CLI module, which contains the main logic of the project.

4. caf/caf/cli.py: This file implements a command line interface on the hight level, actual command implementation are in cli_commands.py for the project, in it user commands are defined (init, commit, etc). The program utilize a argparce library to process command line input to the python. It uses a data structure to map command names to their functions and definitions.

5. caf/caf/cli_commands.py: This file contains the actual implementation of the command-line logic for the CAF, available to users. Each function in this module represents one command that the user can address through the CLI, such as initializing the repository, committing, creating/deleting branches ans etc. The program interacts with a libcaf library throught modules like repository.py and others. This file connects user commands to the internal functionality of the CAF system.

6. libcaf/libcaf/plumbing.py: This file contains low-level functions that interact directly with the core C++ code. It provides functionailty for reading and writing objects, files, hashing and etc, higher level modules rely on this module to perform their operations efficiently. In more simple words this file is like a bridge between C++ and Python parts of the project.

7. libcaf/libcaf/ref.py: This file defines different types of references in CAF. A reference maybe a direct hash to a commit, or simbolic reference that points to another reference such as branch name. The model also has read and write functions from files, also validating references formart. Essentialy, this part of the project is need to manage commits and branches within repository.

8. libcaf/libcaf/repository.py: This file is the core of the Python interface to the CAF repo. The Repository class handels all the methods repository has, such as initializtion, creating commits, showing difference and etc. It is a backend for CLI commands and uses plumbing, ref and C++ modules to perform its operations. This file is a main controller of the repository state and caf functionality.

9. libcaf/src/bind.cpp: This file contains Python bidings for the C++ code. It creates the _libcaf Python module and bind all the low-level C++ and data structures be accesible from Python, In practice, this is a bridge between Python and C++ backend.

10. libcaf/src/caf.cpp: This file implements the low-level content-addressable storage operation in C++. Here realized core functionality of hashing files, saving the objects store, reading, writing object data by hash also deleting stored information. It is connects to Python via pybind11 in bind.cpp.

11. libcaf/src/hash_types.cpp/h: This file implements hashing logic for Blobs, Trees and Commits. It communicates with caf.cpp and assist it to make content addressable storage work.

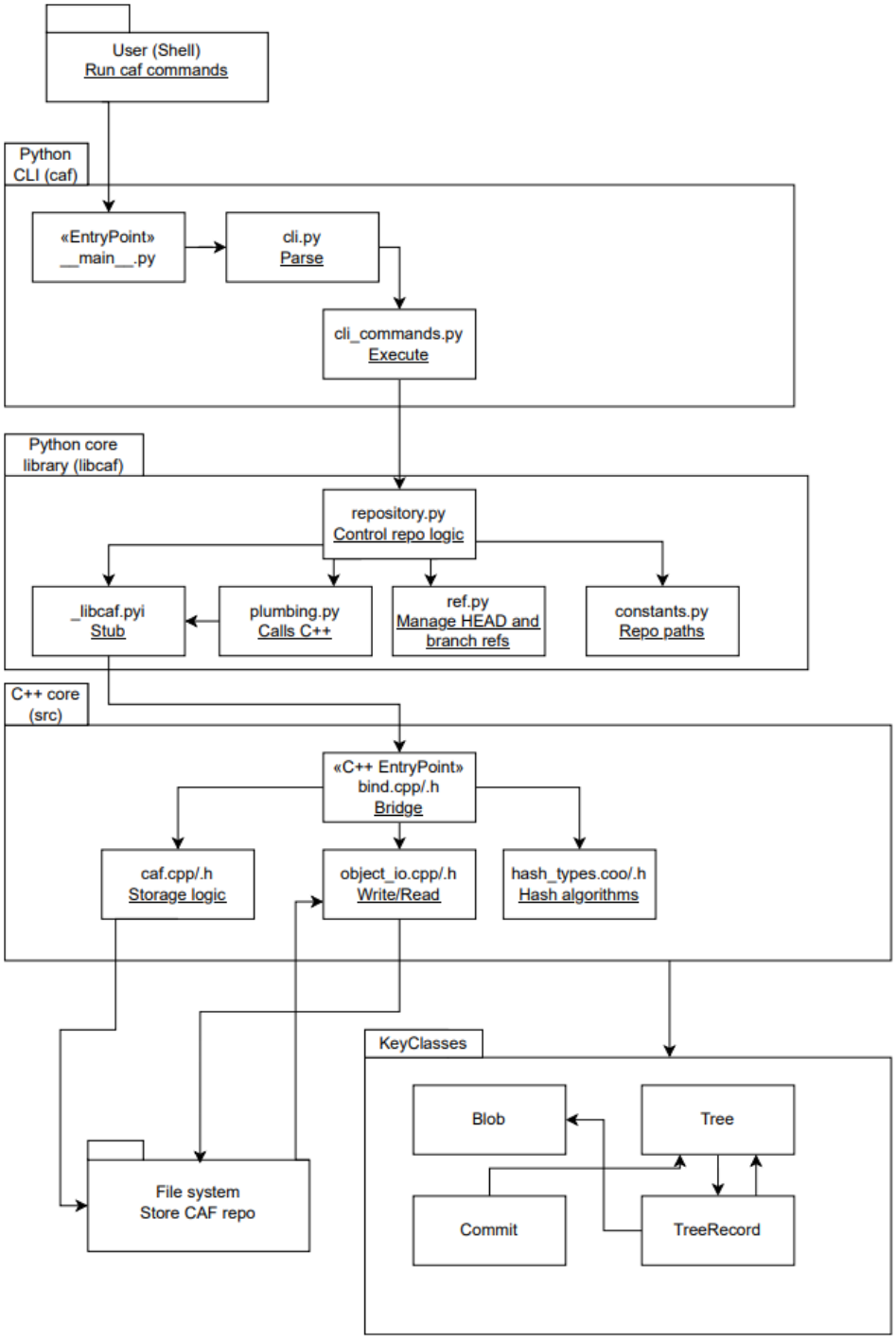
12. libcaf/src/object_io.cpp/h: This file manages input and output operations, how commits and trees are stored on and loaded from the disk. It writes object in a binary format to a file, identified by respectful hash.

13. libcaf/src/CMakeLists.txt: This file is a build for C++ part of the project.

14. test/caf/cli_commands: This is a seriece of files that test the CLI commands, that user interracts with, using pytest framework.

15. test/libcaf: This is a seriece of files that test the libcaf library, in partucluar methods that are available to the user and crutial to the CAF workflow, using pytest framework.

Task 2 : Project Diagram



The diagram shows the main modules with their main responsibility underlined in the box , I seperated the diagram into four layers , User , Python CLI , python core, c++ core , (I also included the keyclasses in c++ with a package of their own), and fileSystem which isn't really a layer but it was required to show the interactions , an arrow start from a file that uses the file at

the end of the arrow , so that now looking at the diagram shows the controlflow of the execution, I also stated where the entrypoints were , showed data flow and stated where the bridge between python and c++ was , they communicate using pybind implemented in bind.cpp which is compiled into _libcaf, and the information passed between them are the keyclasses, the repository state is stored in text files on the disk, and they are retrieved when python needs them he calls c++ that reads and returns the stored objects, errors like when a file is missing are thrown as c++ exceptions and using pybind they are converted into python exceptions that are caught in the python core and finally reach the user.

Task 4 : Bug Fixes

Fixed tests 1 – 3 that were failing due to unsorted dictionary iteration in repository.py file. Names of the test are:

• tests/libcaf/test_diff.py::test_diff_nested_trees — FAILED

This test creates two directories dir1 and dir2. In the second commit, file_a.txt in dir1 is modified,and file_b.txt in dir2 is replaced by file_c.txt, so both directories are considered modified. The test expects the first entry in the modified list to be dir1, but the actual value was dir2. This happens because the diff tree children were appended in traversal order, since we dont have a garanty to sort in lexicographical order by directory name. To fix this, I added a step in diff_commits that recursively sorts diff.children at every node by record.name, such that dir1 stored before dir2 in the modified list.

Link to Commit

• tests/libcaf/test_diff.py::test_diff_moved_file_added_first — FAILED

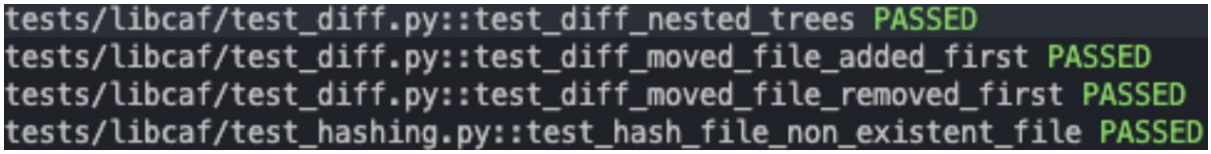
This test also sets up dir1 and dir2, but then moves file_a.txt from dir1 into dir2 as file_c.txt. As a result, both dir1 and dir2 are reported as modified. The test expects modified[0].record.name to be dir1, but it was dir2 instead. Again, the problem was that the diff children were stored in the order they were discovered during tree traversal, without any explicit sorting, so the order of modified directories was not deterministic. To fix this, I added a step in diff_commits that recursively sorts diff.children at every node by record.name, such that dir1 stored before dir2 in the modified list.

Link to Commit

• tests/libcaf/test_diff.py::test_diff_moved_file_removed_first — FAILED

Here the setup is similar, but file_b.txt is moved from dir2 into dir1 as file_c.txt. This again makes both dir1 and dir2 modified. The test asserts that the first modified directory is dir1, but modified[0].record.name was dir2. The issue is the same: the order of diff nodes reflected traversal order rather than a sorted order by directory name. To fix this, I added a step in diff_commits that recursively sorts diff.children at every node by record.name, such that dir1 stored before dir2 in the modified list.

Link to Commit



Fixed test 4 that was failing due to wrong tree record storage in tree.h file.

tests/libcaf/test_objects.py::test_tree_entries_are_canonicalized — FAILED

This test checks that tree values are store in lexicographical order by their names, not by order they are added or any other random order. The test constructs three Tree objects with the same

files a_file, a_file, c_file but in different orders.The test expects all three trees to to be stored in alphabetical order. However, the actuall implementation of tree.h used unordered_map, which does not preserve any order, we used map to preserve the order and hence we get a tree entries stored in order by name as the test expects.

Link to Commit

Task 5 : Tag implementation

Research summary :

Git tags are used to mark specific points in the repository history, it is a static, immutable reference to the commit. It's difference from a branch is that the branch is a mutable pointer that automatically moves forward with next commit. Tag permanently pointer to a specific commit hash and dont change. Tags also differ from just commits, tag is a name for this commit, not the commit itself. Most comman use case of tag is to mark release versions of a product. Git stores tags in refs/tags directory, while lightweight tags are simple pointer of commit, annotated tags also store metadata, that makes them a choice for formal release.

Design summary :

We implemented the tags feature by extending the project while matching the same structure that was already there, each tag is stored as a file under refs/tags and it contains the commit hash for the tag , we added the functions create/delete tag , tags in the python core that toggle adding deleting and listing the existing tags , on the CLI side we added the same functions that parse the arguments and use the core for execution.