CS 246E Final Project Design Document

Leyang Zou

December 13, 2024

1 Introduction

In a 2015 interview, Linus Torvalds said the following:

> So I'd like to stress that while [Git] really came together in just about ten days or so … it wasn't like it was some kind of mad dash of coding. The actual amount of that early code is actually fairly small, it all depended on getting the basic *ideas* right. And that I had been mulling over for a while before the whole project started.

If there's one thing I have learned over my time working on myvc, it is that Torvalds was correct about the ideas being just as important, if not more so, than the actual implementation. About half of the time I spent working on the project consisted of no coding at all; instead, I was sitting with a notebook open mulling over the design of the whole thing. How was I going to support reading and writing entities that needed to be persistent on the file system? How was I going to avoid the practice of writing C code and calling it C++?

When I had (eventually) arrived at a sufficient response to these questions, the implementation of myvc was not exceedingly challenging. Sure, I spent a number of hours debugging my SHA1 and diff implementations, and ran into several different internal compiler errors when trying to use modules, but in general all the pieces fit together nicely - once they were well-designed.

I spent a lot more time than planned working on this project, and was close to not finishing. To be honest, I still seriously doubt that all of the bugs were caught during testing, which makes the final state of myvc quite fragile. However, I did learn a lot about both version control and object-oriented design while making it, and have no regrets proposing it as my project.

2 UML Diagram

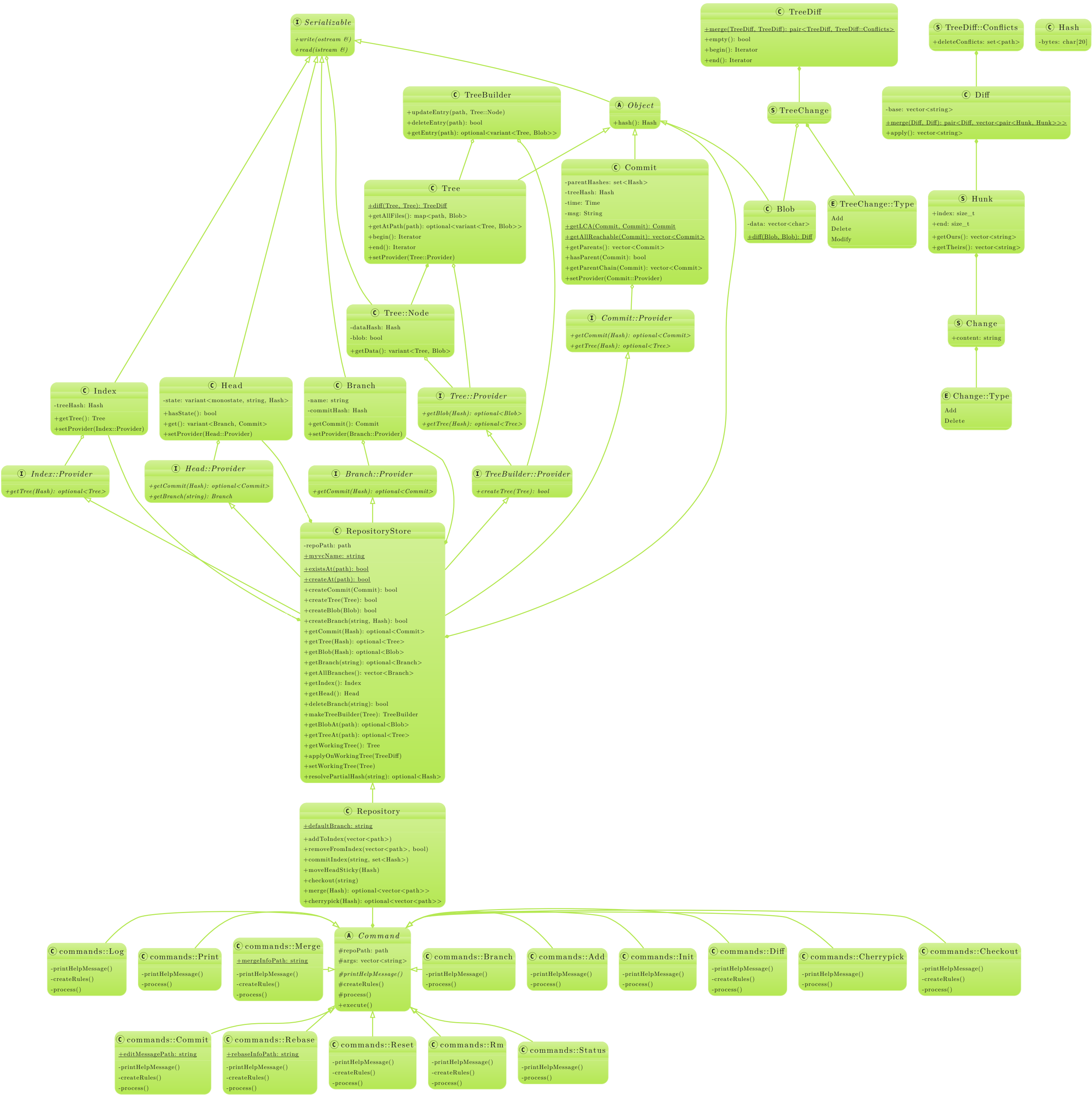

Figure 1: The updated UML diagram for myvc.

2.1 Deviations

My final UML diagram contains many deviations from the UML diagram submitted for due date 1. Specifically, the follow aspects have changed:

- `Synced` was renamed to `Serializable` and the `sync()` method was removed. The reason for this change is described in 3.2.1.
- `CommitAccessor`, `TreeAccessor`, etc., were renamed to `Commit::Provider`, `Tree::Provider`, etc., as I felt this name better described their purpose.
- `::Provider` interfaces no longer depend on any update operations, and the `Blob::Provider` class was removed because it was unnecessary. This is to support the new design outlined in 3.2.2.
- Many responsibilities were moved from the `Index` and `Head` classes to the `Repository` class as these operations are only ever performed as part of larger operations involving multiple entities.
- Added the `TreeDiff`, `TreeBuilder`, and `Repository` classes (and related structs). `TreeDiff` represents a diff between two `Tree` objects while the purposes of the other two are described in 3.2.3 and 3.2.4 respectively.
- Replaced `CommandExecutor` with `Command` and one subclass for each subcommand. The reason is described in 3.2.4.
- `Hash` now stores a `char[20]` instead of a `String`, because SHA1 hashes are fixed-size.

3 Design and Structure

3.1 Structure Overview

myvc is designed based on an MVC architecture, with the following roles:

- The **model** is represented by classes like `Blob`, `Commit`, `Branch`, `Head`, and `TreeDiff`, which collectively represent the internal state of a repository. Classes belonging to the model also support simple operations on their data; for example, `Commit::getAllReachable` is a static method that returns all commits reachable from a given commit.
- The **view** is represented by the abstract `Command` class and its subclasses `Status`, `Add`, `Diff`, `Merge`, etc., which are responsible for parsing command line arguments, validating them, and formatting output displayed to the user. All classes belonging to the view are segregated in the `myvc::commands` namespace.
- The **controller** is represented by the RAII class `RepositoryStore` and its subclass `Repository`, which facilitate the interaction between the view and the model. `RepositoryStore` provides low-level operations that interact with the repository, such as fetching a commit or updating a branch. On the other hand, `Repository` provides common high-level operations such as adding to the index, creating a new commit at HEAD, etc.

Classes in the model are responsible for knowing how to serialize and deserialize themselves from files in the file system through implementing `Serializable` (see 3.2.1), and will fetch their data *lazily* through the use of an abstract `Provider` interface implemented by `RepositoryStore` (see 3.2.2). However, `RepositoryStore` is responsible for managing the actual fetching, caching fetched objects, and syncing them.

One key part of the model is the `Hash` class, which represents a SHA1 hash that is widely used for identifying objects. My implementation of the SHA1 algorithm is encapsulated in `hash.cc`, and I have designed the hashing module to be a black box that is easily replaceable, considering Git's efforts to move to SHA256.

Another important part of the model is the utility classes `TreeDiff` and `Diff`. `TreeDiff` represents a diff between two `Tree` objects, and contains a `std::map` mapping file system paths to `TreeChange` structs. In turn, `TreeChange` contain a type (add, delete, modify) and two blobs representing the old and new versions of the file. On the other hand, `Diff` represents a diff between two text files. Internally, `Diff` uses an unoptimized version of the Myers diff algorithm to identify changes, storing them in hunks, or groups of related changes. `TreeDiff` and `Diff` also support merging two diffs into one, and identifying any conflicts that may occur as a result.

`Repository` leverages the low-level operations in `RepositoryStore` as well as the operations found in model classes to support high-level operations. One key operation that it supports is a three-way merge functionality, which take two trees X and Y and a base tree B, and generates two `TreeDiff` objects by comparing X and B and Y and B respectively. Then, it merges these diffs, resets the working tree to B, and incrementally applies the merged diff. This allows the implementation of other operations like merge, cherry-pick, and rebase.

View classes (subclasses like command) leverage all of these operations to implement their respective functionalities, being responsible for parsing the command-line input and for displaying the output.

3.2 Design Challenges

3.2.1 Persistent Data and Serialization

The main challenge that I faced while designing myvc is finding a robust way to handle persistent data. Like Git, myvc makes use of a hidden `.myvc` directory in the file system to store entities like objects, branches, the index, and HEAD. During the execution of a subcommand, myvc needs to be able to fetch these entities from the file system as well as synchronize them in the case of updates.

In my original design, a stateless `RepositoryStore` class is responsible for handling reading and writing to the file system. This class knows where every type of entity can be found, and provides methods like `getBlob(Hash): Hash` and `createTree(Tree)` which read and write directly to the file system. Importantly, while `RepositoryStore` knows where to access entities, it does not know *how* to serialize them. Instead, we provide the following interface[1] implemented by all serializable entities:

[1]In my UML diagram for due date 1, there is an extra `sync()` method. However, I quickly realized that it doesn't make sense for `Synced` to have this method, so I removed it.

```cpp
class Synced {
public:
  virtual std::vector<char> serialize() const = 0;
  virtual void deserialize(const std::vector<char> &) = 0;
};
```

Then, `RepositoryStore` would simply call these methods on the entities that it is fetching or storing. The rationale behind this separation is that it makes the most sense for a class to handle its own serialization and deserialization, as these processes are tightly-coupled with its internal representation. On the other hand, a change to an entity's internals should not affect the implementation of `RepositoryStore`.

While this model works well for immutable entities like objects, it makes interacting with mutable entities like branches awkward. Consider the following interaction:

```cpp
Branch a = store.getBranch(branchName);
Branch b = store.getBranch(branchName);
b.setCommit(store.getCommit("da39a3ee5e6b4b0d3255bfef95601890afd80709"));
store.updateBranch(b);
```

Notice that the state of `a` is not updated, which leads to the possibility of having extremely subtle bugs, especially if `a` is stored later. Furthermore, it would be much nicer if we didn't have to call `updateBranch` every time the state is updated, and it is instead synchronized automatically. However, this is impossible with a stateless `RepositoryStore`.

Another issue with this implementation is that reading and writing to the file system every time we fetch or store an entity can be extremely slow, especially if we are updating the same entity multiple times or creating a temporary entity that does not need to be stored.

My solution to this problem is to allow `RepositoryStore` to have an internal state that essentially caches all of the objects already fetched or created, as follows:

```cpp
class RepositoryStore {
    mutable std::map<Hash, std::unique_ptr<Object>> objects;
    mutable std::map<std::string, Branch> branches;
    mutable std::optional<Index> index;
    mutable std::optional<Head> head;
};
```

Every time an entity is fetched, the created object is cached in one of these members. This solves the aforementioned synchronization problem because when fetching a branch, we can return a *reference* to the `Branch` object instead. Then, since there is only one `Branch` object per branch, the state will be synchronized. Furthermore, we no longer have to synchronize updated entities explicitly in the client code: `RepositoryStore` can effectively become an RAII class that writes all of its cached objects to the file system in its destructor. Lastly, we do not have to worry about unnecessary reads and writes to the file system, as the updated state is guaranteed to be written exactly once.

The new design of `RepositoryStore` provides an interface that is much nicer to work with. One last change that I made is to rename `Synced` to `Serializable` (as this is more representative of what the interface represents), and to use input and output streams instead of `vector<char>`:

```
class Serializable {
public:
    virtual void write(std::ostream &) const = 0;
    virtual void read(std::istream &) = 0;
};
```

3.2.2 Entity Representation

According to my original proposal for myvc, a commit object was to be represented as something like the following class:

```
struct Commit : public Object {
    std::vector<Commit> parents;
    Tree tree;
};
```

This seems like the most natural way to represent a commit in C++, as it conceptually has a copy of the working tree as well as parent commits. However, consider what would happen if we had a repository with thousands of commits. When `RepositoryStore` needs to fetch the commit at HEAD, it must load every commit that is an ancestor of that commit through thousands of file system accesses, even if those commits are never used in the program. That would make myvc completely unscalable!

A second try would be to implement commits as a plain data structure:

```
struct Commit : public Object {
    std::vector<Hash> parents;
    Hash tree;
};
```

Then, every time we fetch a commit, `RepositoryStore` only needs at most one read from the file system - the information associated with the commit is loaded as needed. This solution is far more scalable, but it means that we can no longer leverage the object-oriented paradigm of C++. For example, if we wanted to get the tree of the first parent commit of `c`, we would need to write something like `store.getTree(store.getCommit(c.parents.at(0)).tree)`. In the old implementation, we were able to just write `c.parents.at(0).tree` instead.

Another problem is that we would no longer be able to implement operations associated with commits in the same module; for example, we wouldn't be able to add a static method `Commit Commit::getLCA(Commit, Commit)` without manually passing in a `RepositoryStore` because we would have no way of fetching parents.

A third attempt is to implement a hybrid of the first two attempts:

```cpp
class Commit : public Object {
private:
    std::vector<Hash> parents;
    Hash tree;
    std::weak_ptr<RepositoryStore> store;
public:
    vector<Commit> getParents() const;
    Tree getTree() const;
};
```

Every entity associated with a repository will contain a private pointer to its `RepositoryStore`. This means that we can still load entities lazily as before, while allowing us to write `c.getParents().at(0).getTree()`. Essentially, we hide the calls to `RepositoryStore` inside the class - optically, it *looks* like `Commit` contains commits and a tree. Now, we are also able to implement methods like `Commit::getLCA` because we no longer need to directly interact with the store.

One remaining problem with this solution is that it introduces unnecessary coupling between `RepositoryStore` and `Commit`. Why should `Commit` need to depend on all of the methods of `RepositoryStore` when all it requires is `getCommit` and `getTree`? With this in mind, the final attempt is as follows:

```cpp
class Commit : public Object {
public:
    class Provider {
    public:
        virtual Commit getCommit(Hash) const = 0;
        virtual Tree getTree(Hash) const = 0;
    };
private:
    std::vector<Hash> parents;
    Hash tree;
    std::weak_ptr<Provider> provider;
public:
    vector<Commit> getParents() const;
    Tree getTree() const;
};
```

We introduce a helper interface `Commit::Provider` that is capable of allowing the commit to fetch its internal state. Then, `RepositoryStore` will simply implement `Commit::Provider` and inject itself in all of the `Commit` objects that it generates. Notice that this is an example of the interface segregation principle and the dependency inversion principle. To decouple `Commit` from `RepositoryStore`, we make `Commit` only depend on the methods that it needs from `RepositoryStore` through the introduction of an abstract interface.

An unintended benefit of this solution is that we may make providers other than `RepositoryStore`. For example, if we wanted to implement some kind of in-memory repository or network-based repository, all we would have to do is implement methods from the relevant providers.

3.2.3 Operations on Trees

The design described in 3.2.2 allows classes like `Commit` to provide static methods like `Commit::getLCA` to implement relevant operations. While this design works well for immutable operations, one case where it fails is for `Tree` objects. Recall that a tree is implemented as follows:

```
class Tree : public Object {
public:
    class Provider {
    public:
        virtual Blob getBlob(Hash) const = 0;
        virtual Tree getTree(Hash) const = 0;
    };
private:
    // a Node contains the hash of a Blob or a Tree
    std::map<std::filesystem::path, Node> nodes;
    std::weak_ptr<Provider> prov;
};
```

One important operation that must be supported is mutating trees. This is essential for performing operations like three-way merges, where we have to apply a merged diff on a tree object. Since objects are immutable in myvc, this would amount to creating a new tree object representing the updated state of the tree[2]. However, `Tree::Provider` does not require any way of creating new trees, and it seems dubious to add a `createTree` method simply for handling updates, especially since `Tree` is suppossed to be immutable.

Instead, we will implement mutable operations on trees in a new `TreeBuilder` class that is specifically responsible for building trees:

```
class TreeBuilder {
public:
    class Provider : public Tree::Provider {
    public:
        virtual void createTree(Tree) = 0;
    };
private:
    Tree tree;
    std::weak_ptr<Provider> prov;
public:
    void updateEntry(const fs::path &, Tree::Node);
    void deleteEntry(const fs::path &);
    Tree getTree() const;
};
```

[2]This is similar to how recursive data structures like linked lists are updated in functional languages like Racket.

TreeBuilder needs to be able to access trees as well as create them, so it is natural to give it a TreeBuilder::Provider that extends the functionality of Tree::Provider. Whenever we need to update trees, we create a new TreeBuilder object and use these methods to create the new tree, thus separating mutable operations on trees from the Tree class itself. This is an example of the single responsibility principle.

3.2.4 Subcommand Representation

In my original design, all of the logic for implementing myvc's subcommands was concentrated in a single CommandExecutor class that only depended on RepositoryStore. While implementing this, I quickly realized that it was infeasible, for two reasons:

1. The functionalities of the subcommands are quite segregated, so implementing everything in one class would lead to a lot of unnecessary coupling.
2. RepositoryStore alone does not provide the high-level functionalities necessary to support merge, cherry-pick, and rebase.

To solve the first problem, I overhauled the design of how subcommands will be implemented. Instead of a CommandExecutor class, we now have a Command abstract class supporting common operations like parsing command-line arguments and resolving a hash-like object (ex. 0adf^^2) to an actual object hash. The Command class is layed out as follows (irrelevant methods not shown):

```
class Command {
protected:
    RepositoryStore store;
    std::vector<std::string> args;

    virtual void printHelpMessage() const = 0;
    virtual void createRules(); // add rules for processing flag arguments
    virtual void process() = 0;
public:
    void execute() {
        createRules();
        // ... parse arguments
        if(hasFlag("-h") || hasFlag("--help")) {
            printHelpMessage();
        } else {
          // ... initialize repository store
          process();
        }
    }
};
```

Since we want all commands to guarantee certain common functionality like printing a help message when -h is passed, it makes sense to leverage the template method design pattern and force subclasses to implement only the extensible functionality in the process method.

It may be concerning that the result of having one subclass per subcommand is 14 separate subclasses of `Command`. Howver, I believe this is necessary because most subcommands represent complex, segregated operations that are not easily tied together. For example, subcommands like `diff` and `status` have almost nothing in common apart from needing to resolve partial hashes to commits. While this solution introduces a certain amount of bloat in the amount of subclasses needed, it serves to increase cohesion and decrease coupling.

To address the second issue, I introduced a new class `Repository` which implements common, high-level operations on repositories. The `Repository` class is implemented as follows:

```
class Repository : public RepositoryStore {
public:
    void addToIndex(const std::vector<fs::path> &);
    void commitIndex(std::string, std::set<Hash> parents);
    // ... etc
};
```

The important point is that `Repository` is implemented as a *subclass* of `RepositoryStore`. It is possible to instead implement `Repository` as containing a `RepositoryStore` as a member, but this would be inconvenient because the low-level operations provided by `RepositoryStore` should still be accessible from a `Repository` instance. Conceptually, a `Repository` should be thought of as a `RepositoryStore` *extended* with high-level functionality. With this change, the `Command` class will also initialize a `Repository` instead of a `RepositoryStore`.

4 Extra Credit Features

I completed the entire project without explicitly managing memory, and without memory leaks (as far as I am aware).

The following minor features were implemented but not specifically required in the marking scheme:

- myvc works normally when it is called while the current working directory is a subdirectory of a repository. For example, if a repository is located at `~/repo`, we may run `myvc add file.txt` while we are in `~/repo/dir/`.
- Implemented an extra subcommand myvc print that can inspect entities.
- Implemented an extra program `myvc-convert` that leverages the libgit2 library to convert a Git repository to a myvc repository. Since libgit2 is not pre-installed on the CS student environment, my submission will include a pre-compiled statically-linked version of it. This program may fail on repositories with complex components, for example repositories with submodules, but has been confirmed to work on myvc's own repository as well as the `1249` repository for CS 246E. The program may only be run after running `myvc init`.
- Implemented `--abort` for `merge` and `rebase` to cancel a merge/rebase.

5 Deviations from Plan

My actual timeline deviated from the timeline described in the due date 1 plan.

Due to having started early, I initially finished implementing objects, branches, the index, the repository store, and serializing by November 28, ahead of the estimated completion date of December 3. I then implemented hashing and diff in the planned time frame, finishing December 1. Implementing all subcommands took me until December 5, which was also the planned time frame.

However, at this point, I realized I was very unhappy with the structure of the project. From December 6 to December 10, I overhauled the project's design and the way it manages persistent state[3]. I am much happier with the resulting design.

Finally, from December 11 to December 13, my partner (Peter Ye) and I tested myvc against a series of repositories, discovering a large amount of bugs in the process. During this time, I also worked on the design document.

For a more detailed log of the project timeline, feel free to check the commit history of the Git repository that I submitted to Marmoset.

6 Final Question

What would I have done differently if I had the change to start over? A lot of things:

- I wouldn't start the implementation until I have a design that I'm completely happy about. This would avoid having to rewrite large parts of my code.
- I would modify my proposal to include less subcommands and instead focus on more useful functionalities. For example, it would have been nice to support a garbage collector (`git gc`), a reflog, or a stash. Currently, I have already done a lot of the background work necessary to implement these features. However, I did not have time to fully implement them.
- I would try to have better time management by prioritizing the functionalities that matter the most.
- I would have abandoned C++ modules from the start, which would have saved me a few hours of debugging.

7 Conclusion

Although I have spent an ungodly number of hours planning, implementing, testing, and debugging myvc, I have also thoroughly enjoyed working on this project. Not only did it give me an opportunity to learn a lot about how version control software like Git work under the hood, but I was also exposed to how the design patterns we discussed in class come up naturally in real-world software engineering. Most importantly, I have come out of this project with a newfound appreciation for good object-oriented design.

[3]This overhaul was motivated by the design challenges described in 3.2.