Project 3

Github Repo

CECS 326 - Spring 2022

Stephen Lyons, 025785875

Noah Daniels, 026466906

Professor Hailu Xu

California State University, Long Beach

College of Engineering

1250 Bellflower Blvd, Long Beach, CA 90840

Due 11/10/2022

Project Architecture

This project was programmed in the Rust programming language and the rust toolchain was used to compile and run the code. All classes and the main code are written in the main2.rs file. The code can be run via the executable file or by installing the rust toolchain:

To install Rust

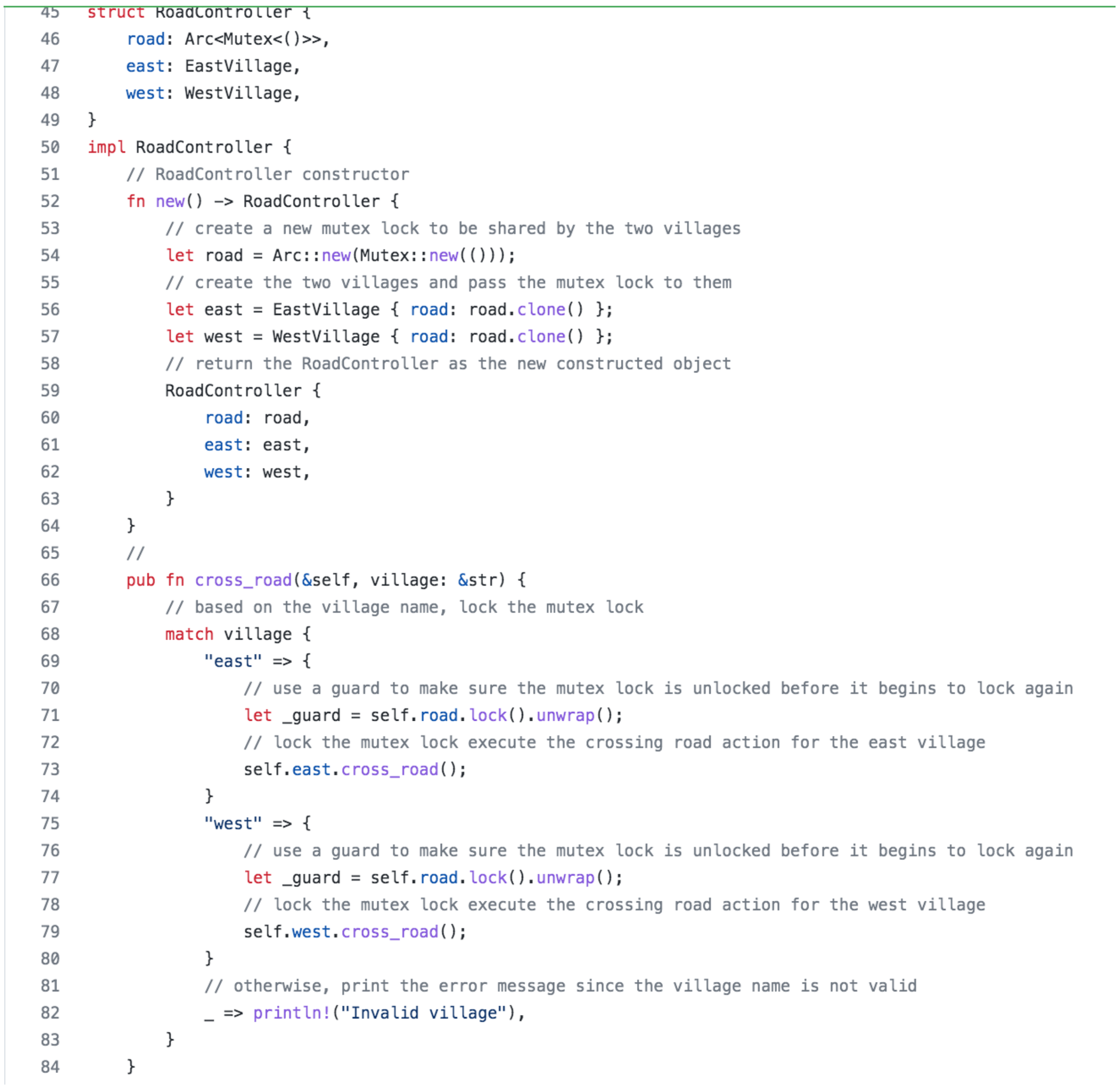
1. Install Rustup: <https://rustup.rs/>
2. Install Rust stable: `rustup install stable`

To run the project

1. `cd` into the project directory `project3rust`
2. `cargo run`

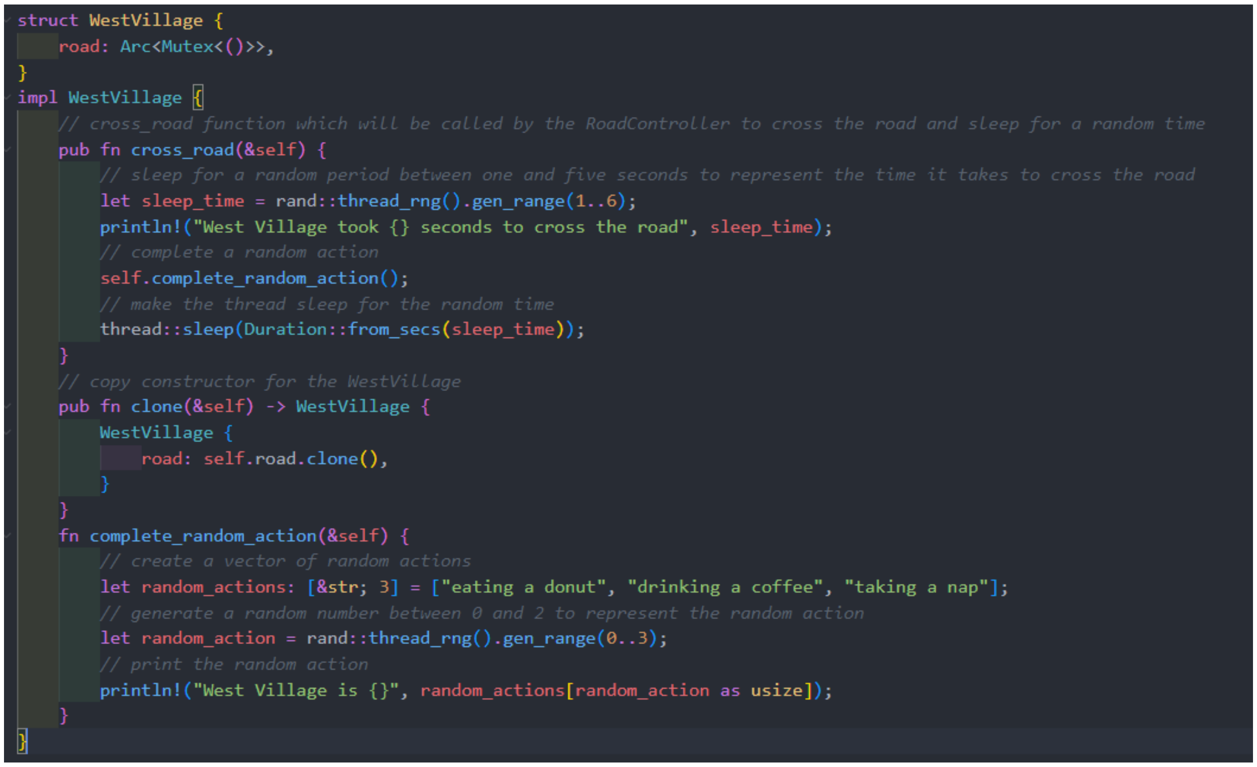
or

3. Run the executable in the target directory (project3rust\target\release\project3rust.exe)



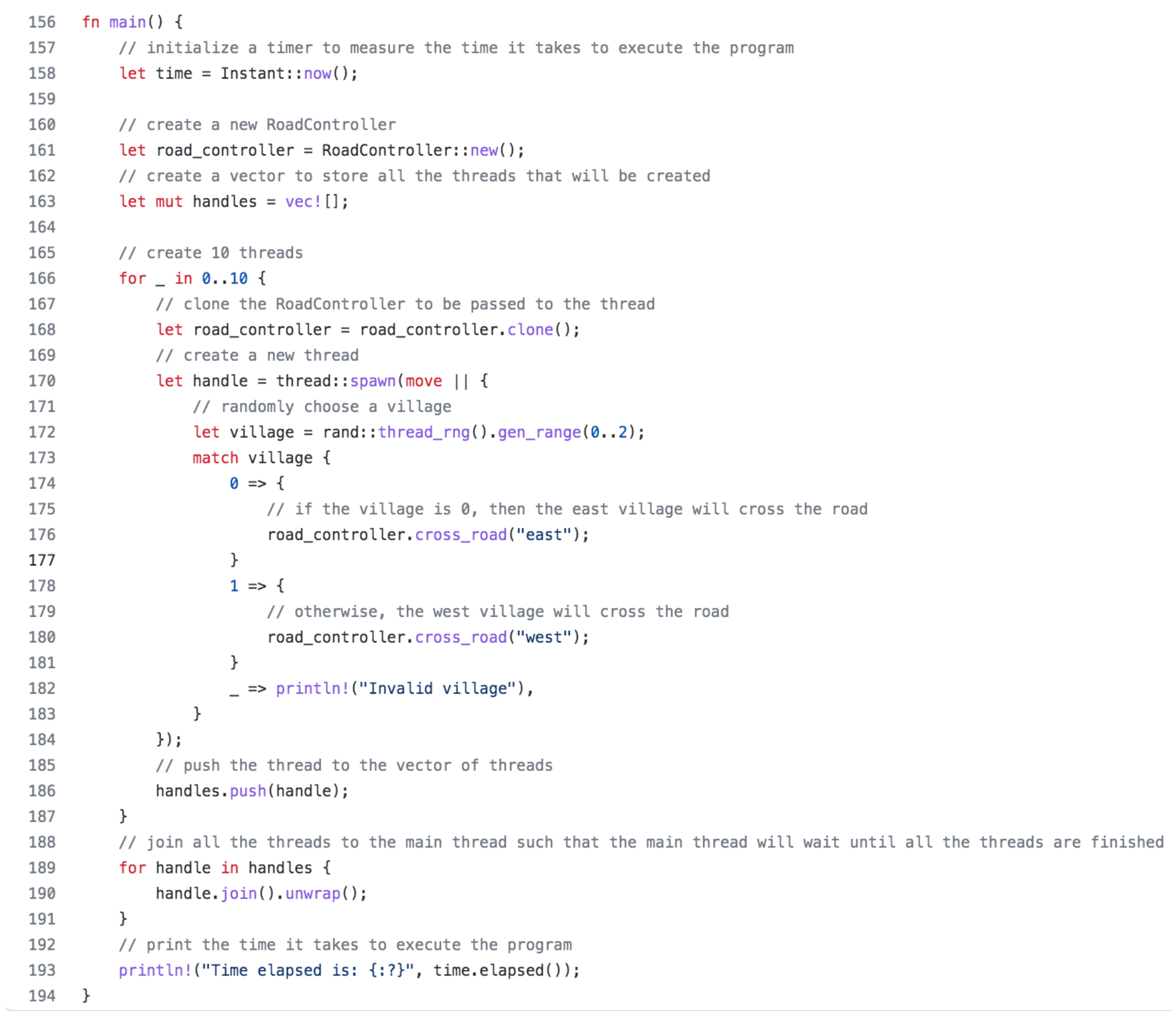
```rust
struct RoadController {
    road: Arc<Mutex<()>>,
    east: EastVillage,
    west: WestVillage,
}
impl RoadController {
    // RoadController constructor
    fn new() -> RoadController {
        // create a new mutex lock to be shared by the two villages
        let road = Arc::new(Mutex::new(()));
        // create the two villages and pass the mutex lock to them
        let east = EastVillage { road: road.clone() };
        let west = WestVillage { road: road.clone() };
        // return the RoadController as the new constructed object
        RoadController {
            road: road,
            east: east,
            west: west,
        }
    }
    //
    pub fn cross_road(&self, village: &str) {
        // based on the village name, lock the mutex lock
        match village {
            "east" => {
                // use a guard to make sure the mutex lock is unlocked before it begins to lock again
                let _guard = self.road.lock().unwrap();
                // lock the mutex lock execute the crossing road action for the east village
                self.east.cross_road();
            }
            "west" => {
                // use a guard to make sure the mutex lock is unlocked before it begins to lock again
                let _guard = self.road.lock().unwrap();
                // lock the mutex lock execute the crossing road action for the west village
                self.west.cross_road();
            }
            // otherwise, print the error message since the village name is not valid
            _ => println!("Invalid village"),
        }
    }
```

The struct class Roadcontroller contains the mutex lock and the two villages in which people travel back and forth on. The cross_road method allows us to use a “guard” or mutex lock to unlock and then lock the road again once the person is through. The clone method for all of these classes just makes a copy of the class to be edited in a safe manner. After traveling we have the two village structs, east and west, which make the traveler sleep or drink coffee for a random period of time after the journey as well as keeping track of their travel time. We create ten instances of travel (threads) in the main and push them to villages (vectors of threads) where they are satisfying the rules of the scenario.



```rust
struct WestVillage {
    road: Arc<Mutex<()>>,
}
impl WestVillage {
    // cross_road function which will be called by the RoadController to cross the road and sleep for a random time
    pub fn cross_road(&self) {
        // sleep for a random period between one and five seconds to represent the time it takes to cross the road
        let sleep_time = rand::thread_rng().gen_range(1..6);
        println!("West Village took {} seconds to cross the road", sleep_time);
        // complete a random action
        self.complete_random_action();
        // make the thread sleep for the random time
        thread::sleep(Duration::from_secs(sleep_time));
    }
    // copy constructor for the WestVillage
    pub fn clone(&self) -> WestVillage {
        WestVillage {
            road: self.road.clone(),
        }
    }
    fn complete_random_action(&self) {
        // create a vector of random actions
        let random_actions: [&str; 3] = ["eating a donut", "drinking a coffee", "taking a nap"];
        // generate a random number between 0 and 2 to represent the random action
        let random_action = rand::thread_rng().gen_range(0..3);
        // print the random action
        println!("West Village is {}", random_actions[random_action as usize]);
    }
}
```

The village classes were both constructed using the same mutex, which prevents the resource from being accessed at the same time. Both the EastVillage and WestVillage classes have a cross_road method which is called by the RoadController to cross the road and sleep for a random time. The cross_road method sleeps for a random period between one and five seconds to represent the time it takes to cross the road. The complete_random_action method is called after the thread sleeps to complete a random action. The clone method is used to create a copy of the village class to be edited in a safe manner. The EastVillage and WestVillage classes are both implemented in the main.rs file.



```rust
fn main() {
    // initialize a timer to measure the time it takes to execute the program
    let time = Instant::now();

    // create a new RoadController
    let road_controller = RoadController::new();
    // create a vector to store all the threads that will be created
    let mut handles = vec![];

    // create 10 threads
    for _ in 0..10 {
        // clone the RoadController to be passed to the thread
        let road_controller = road_controller.clone();
        // create a new thread
        let handle = thread::spawn(move || {
            // randomly choose a village
            let village = rand::thread_rng().gen_range(0..2);
            match village {
                0 => {
                    // if the village is 0, then the east village will cross the road
                    road_controller.cross_road("east");
                }
                1 => {
                    // otherwise, the west village will cross the road
                    road_controller.cross_road("west");
                }
                _ => println!("Invalid village"),
            }
        });
        // push the thread to the vector of threads
        handles.push(handle);
    }
    // join all the threads to the main thread such that the main thread will wait until all the threads are finished
    for handle in handles {
        handle.join().unwrap();
    }
    // print the time it takes to execute the program
    println!("Time elapsed is: {:?}", time.elapsed());
}
```

The main function creates a new RoadController and a vector to store all the threads that will be created. The main function then creates ten threads and pushes them to the vector of threads. The main function then joins all the threads to the main thread such that the main thread will wait until all the threads are finished. The main function then prints the time it takes to execute the program so that we can verify that the program is running correctly. If an indefinite runtime is preferred, the main function can be modified to indefinitely create threads with a while loop but we chose to use a for loop so that we could verify that the program was running correctly. The main function is also implemented in the main.rs file along with all the classes.

##Contributions

Same as in the last report, Stephen wrote the code for the project in Rust and Noah contributed some of the supporting comments and portions of the main function. The report was written by both of us and the code execution video was recorded by Noah.