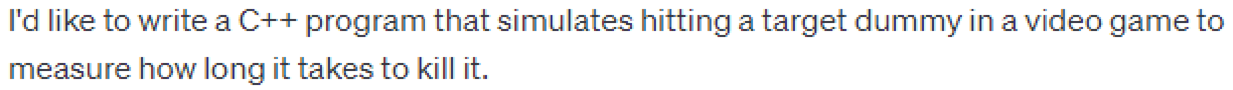
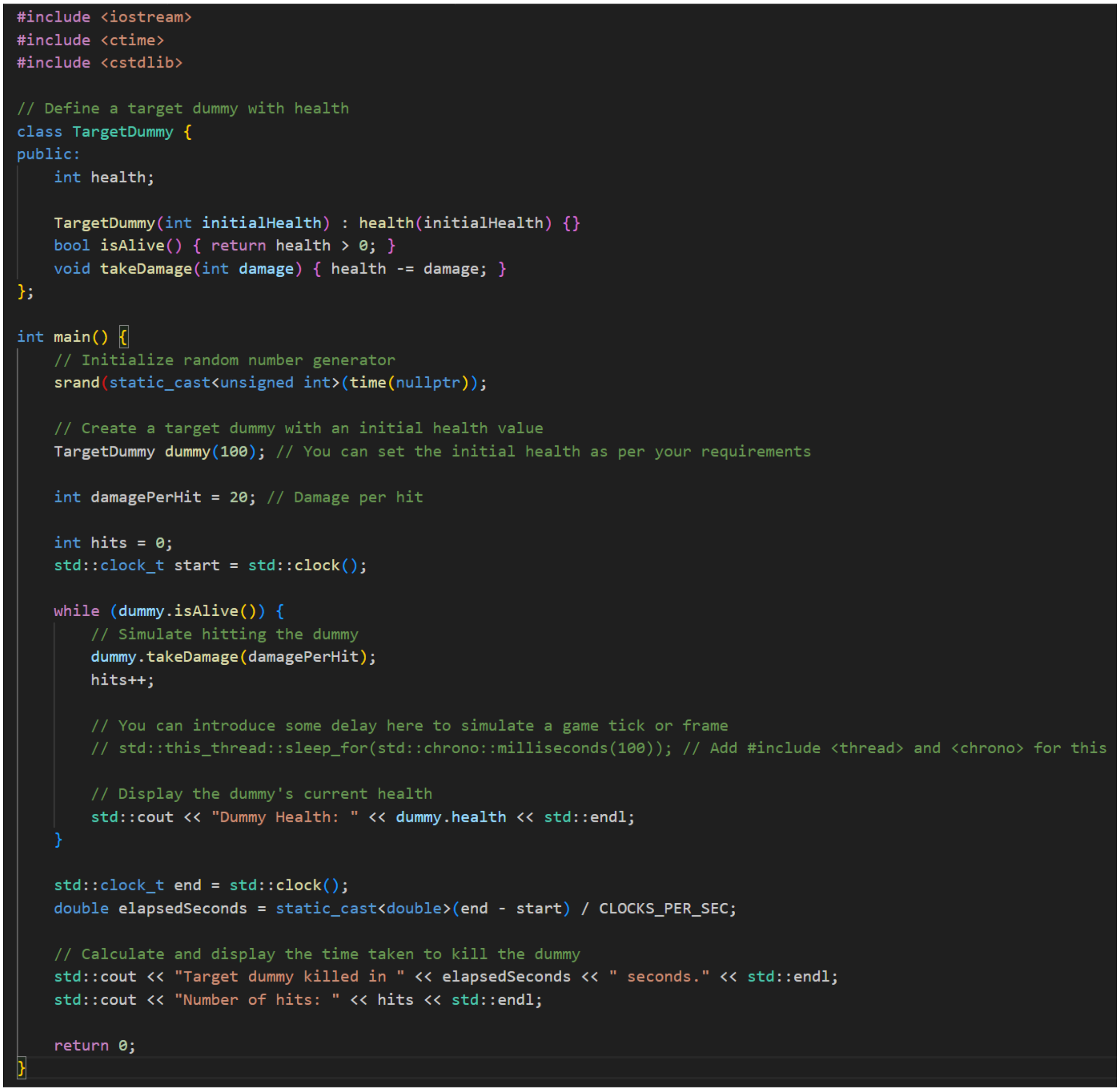
As a long time MMO player (sufferer?), one of the greatest challenges I face on a regular basis is not some difficult boss or mechanic; it's the dreaded Update Day. Because a new update means changes to gear, abilities, and more. And that means it's time to go through the process of testing all that stuff to see what performs best. I've often thought to myself, a far-off look in my eye, "Wouldn't it be nice if I could speed this up?" Well, I finally decided to start on that project - and who better to help me out than my good old friend ChatGPT?

I decided that I would try to get ChatGPT to write the code from descriptions of what I wanted, without me directly telling it what code to write (spoiler: this would not last). So I loaded it up, and gave it this prompt:



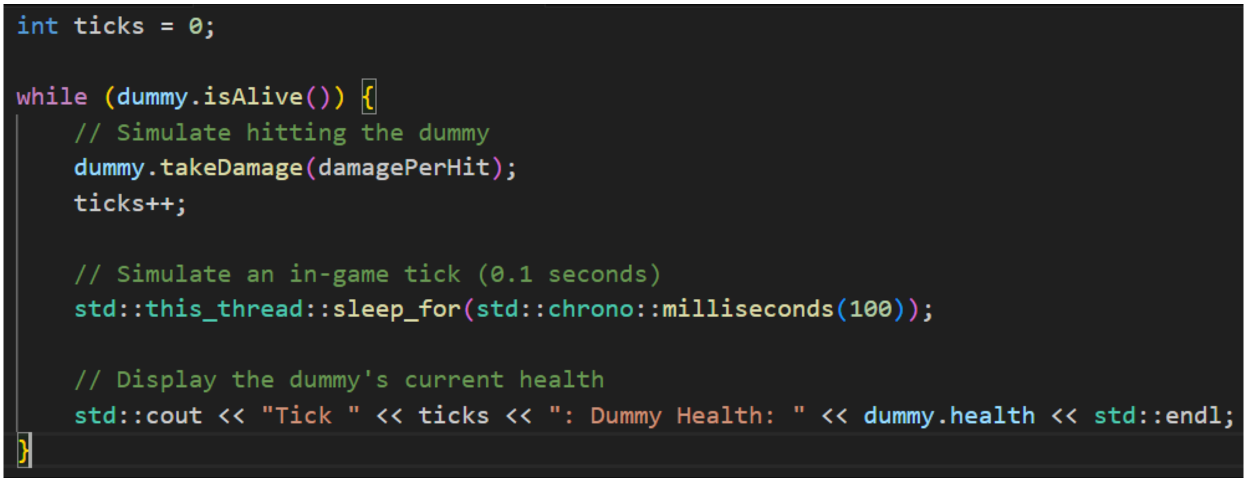
I'd like to write a C++ program that simulates hitting a target dummy in a video game to measure how long it takes to kill it.

And this is what I got:



```cpp
#include <iostream>
#include <ctime>
#include <cstdlib>

// Define a target dummy with health
class TargetDummy {
public:
    int health;

    TargetDummy(int initialHealth) : health(initialHealth) {}
    bool isAlive() { return health > 0; }
    void takeDamage(int damage) { health -= damage; }
};

int main() {
    // Initialize random number generator
    srand(static_cast<unsigned int>(time(nullptr));

    // Create a target dummy with an initial health value
    TargetDummy dummy(100); // You can set the initial health as per your requirements

    int damagePerHit = 20; // Damage per hit

    int hits = 0;
    std::clock_t start = std::clock();

    while (dummy.isAlive()) {
        // Simulate hitting the dummy
        dummy.takeDamage(damagePerHit);
        hits++;

        // You can introduce some delay here to simulate a game tick or frame
        // std::this_thread::sleep_for(std::chrono::milliseconds(100)); // Add #include <thread> and <chrono> for this

        // Display the dummy's current health
        std::cout << "Dummy Health: " << dummy.health << std::endl;
    }

    std::clock_t end = std::clock();
    double elapsedSeconds = static_cast<double>(end - start) / CLOCKS_PER_SEC;

    // Calculate and display the time taken to kill the dummy
    std::cout << "Target dummy killed in " << elapsedSeconds << " seconds." << std::endl;
    std::cout << "Number of hits: " << hits << std::endl;

    return 0;
}
```

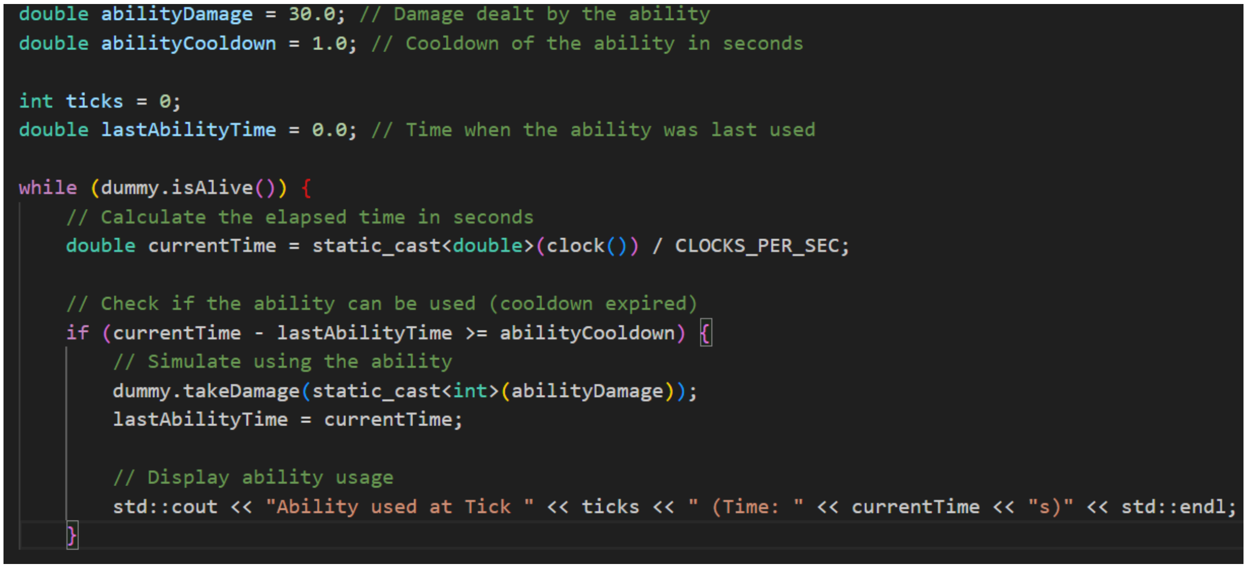
Pretty good! It creates a target dummy, hits it til it’s dead, and then reports the time it took to kill. There’s one slight problem though - it doesn’t compile. The line that initializes the random number generator has three left parentheses, and two right parentheses. ChatGPT was more than happy to correct its mistake, and then I asked it to calculate the time taken based on some arbitrary tick, instead of wall-clock time. And this is the relevant code:



```
int ticks = 0;

while (dummy.isAlive()) {
    // Simulate hitting the dummy
    dummy.takeDamage(damagePerHit);
    ticks++;

    // Simulate an in-game tick (0.1 seconds)
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    // Display the dummy's current health
    std::cout << "Tick " << ticks << ": Dummy Health: " << dummy.health << std::endl;
}
```

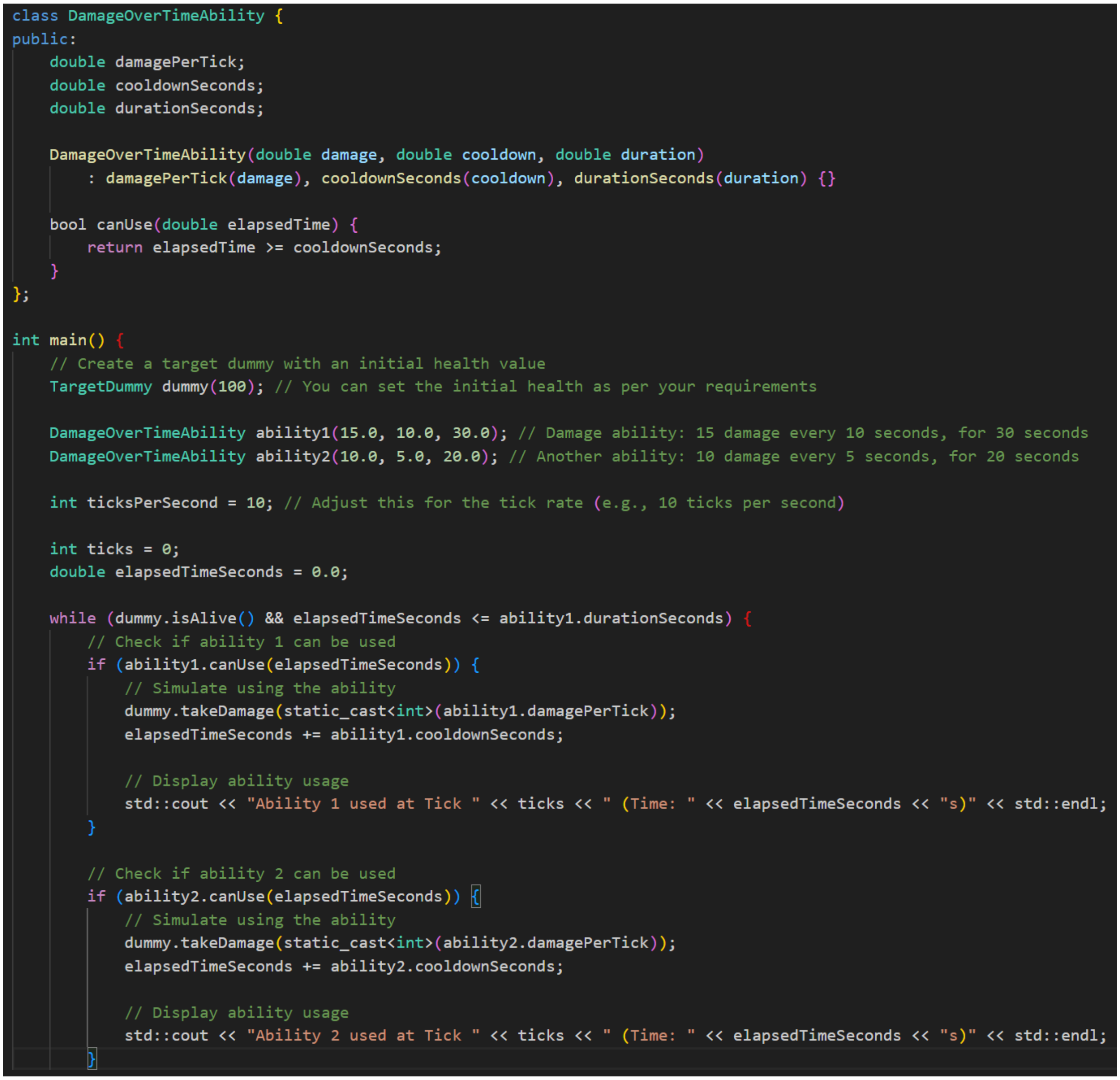
Alright, but the whole point is that I want to speed up the testing process, not wait around while it runs! So I asked it to fix that, and it did so by introducing a tickDuration variable that could be used to calculate the time taken. Good. At this point I had a barebones framework that worked fine. Surely extending more functionality would be easy as pie, right?

Next I figured I'd have to change it from taking (damagePerHit) damage every tick, to having an ability that would fire off damage, but only once per second. Relevant ChatGPT code (Mismatched brackets from here on out are my own doing when copying code):



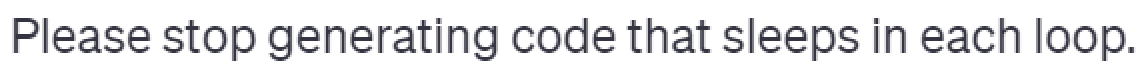
```
double abilityDamage = 30.0; // Damage dealt by the ability
double abilityCooldown = 1.0; // Cooldown of the ability in seconds

int ticks = 0;
double lastAbilityTime = 0.0; // Time when the ability was last used

while (dummy.isAlive()) {
    // Calculate the elapsed time in seconds
    double currentTime = static_cast<double>(clock()) / CLOCKS_PER_SEC;

    // Check if the ability can be used (cooldown expired)
    if (currentTime - lastAbilityTime >= abilityCooldown) {
        // Simulate using the ability
        dummy.takeDamage(static_cast<int>(abilityDamage));
        lastAbilityTime = currentTime;

        // Display ability usage
        std::cout << "Ability used at Tick " << ticks << " (Time: " << currentTime << "s)" << std::endl;
    }
```

Aaaand we're back to using wall-clock time. This, I would find out, would be a recurring theme throughout this project. Upon requesting wall-clock time not be used, ChatGPT made it calculate time based on the number of ticks…but it also left in a line that had the loop sleep on each iteration. I also noticed that it had, at some point, reintroduced the mismatched parentheses error on the random number generator line (another recurring theme). So it fixed both issues, and I requested that it make the number of ticks in a second a variable instead of a hard-coded value, so that it could be easily changed. It did so, and back came the sleeping on each iteration of the loop. Fixed that, and I figured it was time for something a little more in-depth. I asked ChatGPT to implement an ability that would, upon being used, deal (x) damage every (y) seconds, for a total duration of (z) seconds. Important code:



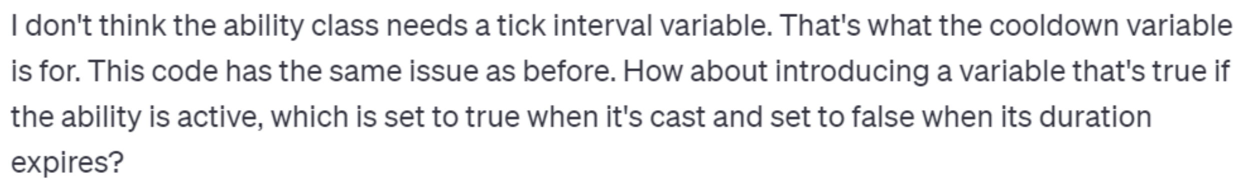
```
class DamageOverTimeAbility {
public:
    double damagePerTick;
    double cooldownSeconds;
    double durationSeconds;

    DamageOverTimeAbility(double damage, double cooldown, double duration)
        : damagePerTick(damage), cooldownSeconds(cooldown), durationSeconds(duration) {}

    bool canUse(double elapsedTime) {
        return elapsedTime >= cooldownSeconds;
    }
};

int main() {
    // Create a target dummy with an initial health value
    TargetDummy dummy(100); // You can set the initial health as per your requirements

    DamageOverTimeAbility ability1(15.0, 10.0, 30.0); // Damage ability: 15 damage every 10 seconds, for 30 seconds
    DamageOverTimeAbility ability2(10.0, 5.0, 20.0); // Another ability: 10 damage every 5 seconds, for 20 seconds

    int ticksPerSecond = 10; // Adjust this for the tick rate (e.g., 10 ticks per second)

    int ticks = 0;
    double elapsedTimeSeconds = 0.0;

    while (dummy.isAlive() && elapsedTimeSeconds <= ability1.durationSeconds) {
        // Check if ability 1 can be used
        if (ability1.canUse(elapsedTimeSeconds)) {
            // Simulate using the ability
            dummy.takeDamage(static_cast<int>(ability1.damagePerTick));
            elapsedTimeSeconds += ability1.cooldownSeconds;

            // Display ability usage
            std::cout << "Ability 1 used at Tick " << ticks << " (Time: " << elapsedTimeSeconds << "s)" << std::endl;
        }

        // Check if ability 2 can be used
        if (ability2.canUse(elapsedTimeSeconds)) {
            // Simulate using the ability
            dummy.takeDamage(static_cast<int>(ability2.damagePerTick));
            elapsedTimeSeconds += ability2.cooldownSeconds;

            // Display ability usage
            std::cout << "Ability 2 used at Tick " << ticks << " (Time: " << elapsedTimeSeconds << "s)" << std::endl;
        }
```

This wasn’t too bad, but it did have a number of issues, such as the while loop now ending prematurely when the first ability ran out, instead of only when the dummy died. It also brought back sleeping on each loop iteration, to which I responded in no uncertain terms:



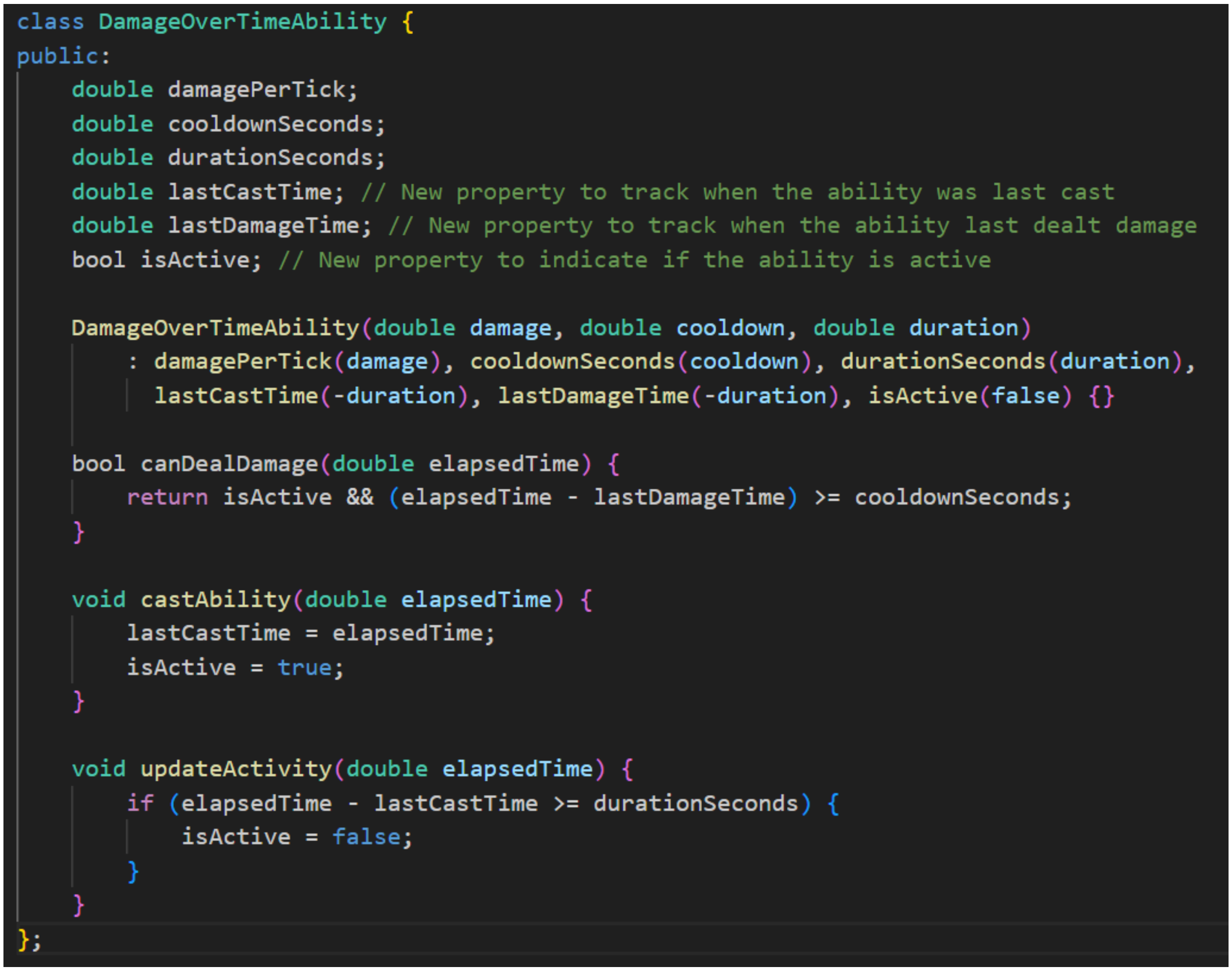
Please stop generating code that sleeps in each loop.

This actually worked, and I did not see another sleep_for call for the rest of the project. This was about the point when I really had to start going back-and-forth with ChatGPT to fix the issues in the code. I was still trying not to tell it what code to write, and after a handful of exchanges, I made my first suggestion to to the AI:



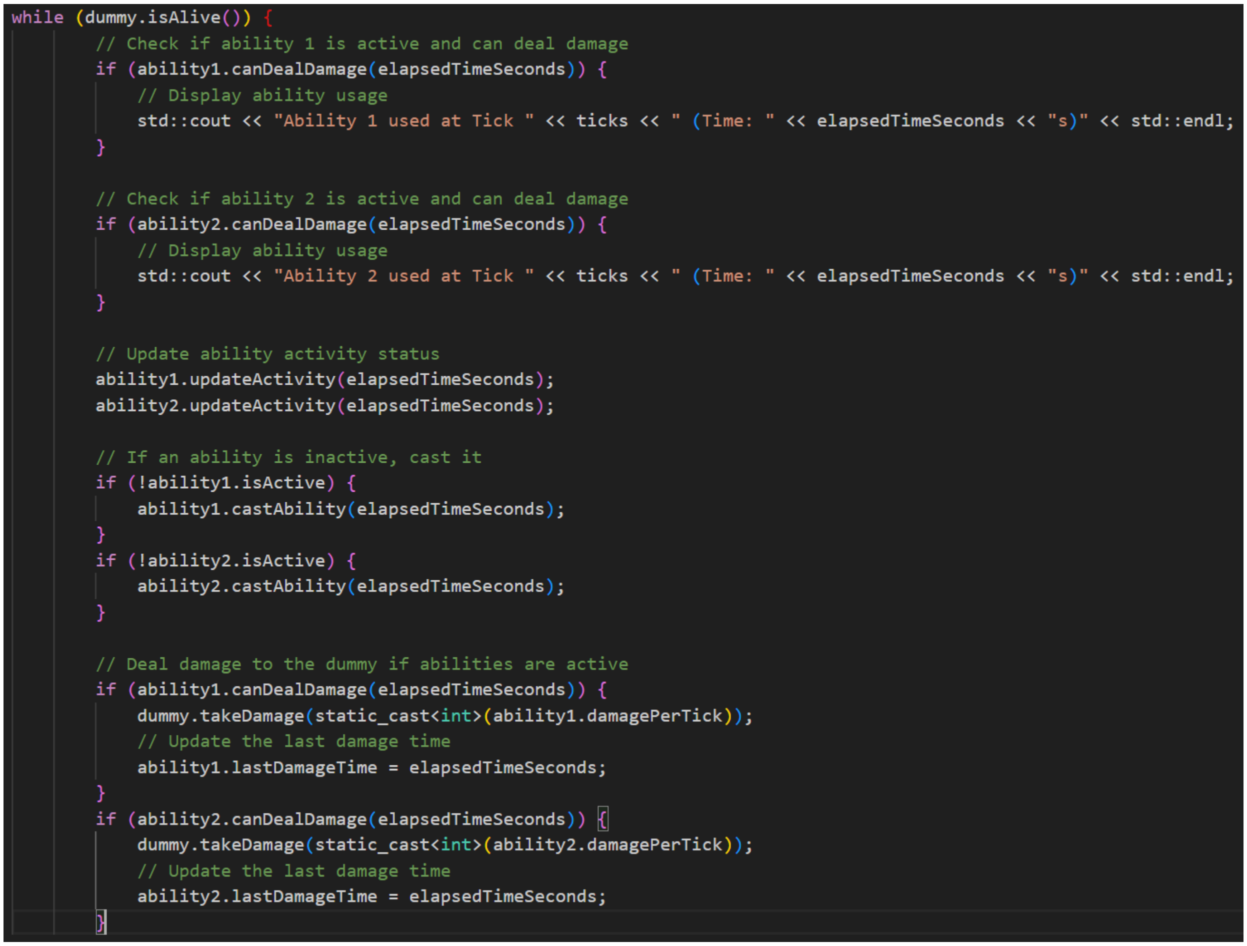
I don't think the ability class needs a tick interval variable. That's what the cooldown variable is for. This code has the same issue as before. How about introducing a variable that's true if the ability is active, which is set to true when it's cast and set to false when its duration expires?

It promptly took my suggestion, and used it to create code that still didn’t work. After several more back-and-forths (including fixing some very simple syntax errors), with me being extremely specific about what I wanted, I finally got code that almost functioned correctly. The ability class now looked like this:



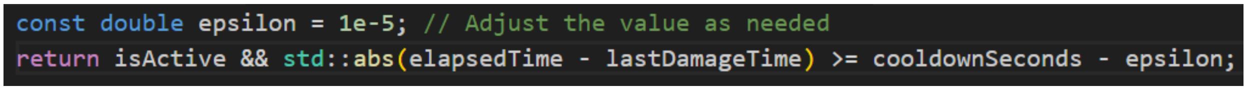
```
class DamageOverTimeAbility {
public:
    double damagePerTick;
    double cooldownSeconds;
    double durationSeconds;
    double lastCastTime; // New property to track when the ability was last cast
    double lastDamageTime; // New property to track when the ability last dealt damage
    bool isActive; // New property to indicate if the ability is active

    DamageOverTimeAbility(double damage, double cooldown, double duration)
        : damagePerTick(damage), cooldownSeconds(cooldown), durationSeconds(duration),
          lastCastTime(-duration), lastDamageTime(-duration), isActive(false) {}

    bool canDealDamage(double elapsedTime) {
        return isActive && (elapsedTime - lastDamageTime) >= cooldownSeconds;
    }

    void castAbility(double elapsedTime) {
        lastCastTime = elapsedTime;
        isActive = true;
    }

    void updateActivity(double elapsedTime) {
        if (elapsedTime - lastCastTime >= durationSeconds) {
            isActive = false;
        }
    }
};
```

And the relevant part of the loop body like this:



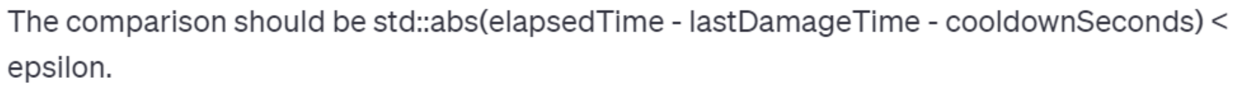
```
while (dummy.isAlive()) {
    // Check if ability 1 is active and can deal damage
    if (ability1.canDealDamage(elapsedTimeSeconds)) {
        // Display ability usage
        std::cout << "Ability 1 used at Tick " << ticks << " (Time: " << elapsedTimeSeconds << "s)" << std::endl;
    }

    // Check if ability 2 is active and can deal damage
    if (ability2.canDealDamage(elapsedTimeSeconds)) {
        // Display ability usage
        std::cout << "Ability 2 used at Tick " << ticks << " (Time: " << elapsedTimeSeconds << "s)" << std::endl;
    }

    // Update ability activity status
    ability1.updateActivity(elapsedTimeSeconds);
    ability2.updateActivity(elapsedTimeSeconds);

    // If an ability is inactive, cast it
    if (!ability1.isActive) {
        ability1.castAbility(elapsedTimeSeconds);
    }
    if (!ability2.isActive) {
        ability2.castAbility(elapsedTimeSeconds);
    }

    // Deal damage to the dummy if abilities are active
    if (ability1.canDealDamage(elapsedTimeSeconds)) {
        dummy.takeDamage(static_cast<int>(ability1.damagePerTick));
        // Update the last damage time
        ability1.lastDamageTime = elapsedTimeSeconds;
    }
    if (ability2.canDealDamage(elapsedTimeSeconds)) {
        dummy.takeDamage(static_cast<int>(ability2.damagePerTick));
        // Update the last damage time
        ability2.lastDamageTime = elapsedTimeSeconds;
    }
```

The next issue I had to solve was that comparing doubles in the canDealDamage and updateActivity functions was prone to floating point errors. So I asked ChatGPT to fix this issue by doing the comparison using an epsilon. An epsilon, for those who have never had the absolute pleasure of dealing with floating point errors, is a small number that basically amounts to a tolerance for inequality. It's a number that says, if you're not exactly equal, but within "this" close of being equal, then that's acceptable, you count as being equal. So this is what ChatGPT gave me:



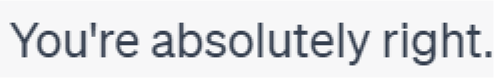
```
const double epsilon = 1e-5; // Adjust the value as needed
return isActive && std::abs(elapsedTime - lastDamageTime) >= cooldownSeconds - epsilon;
```

And this is where I made the biggest mistake of the entire project:



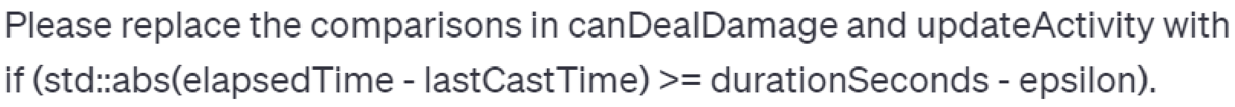
The comparison should be std::abs(elapsedTime - lastDamageTime - cooldownSeconds) < epsilon.

After having to correct ChatGPT so many times so far, I got overconfident and corrected it without testing its code first, thinking it made a mistake in the calculation. It told me:



You're absolutely right.

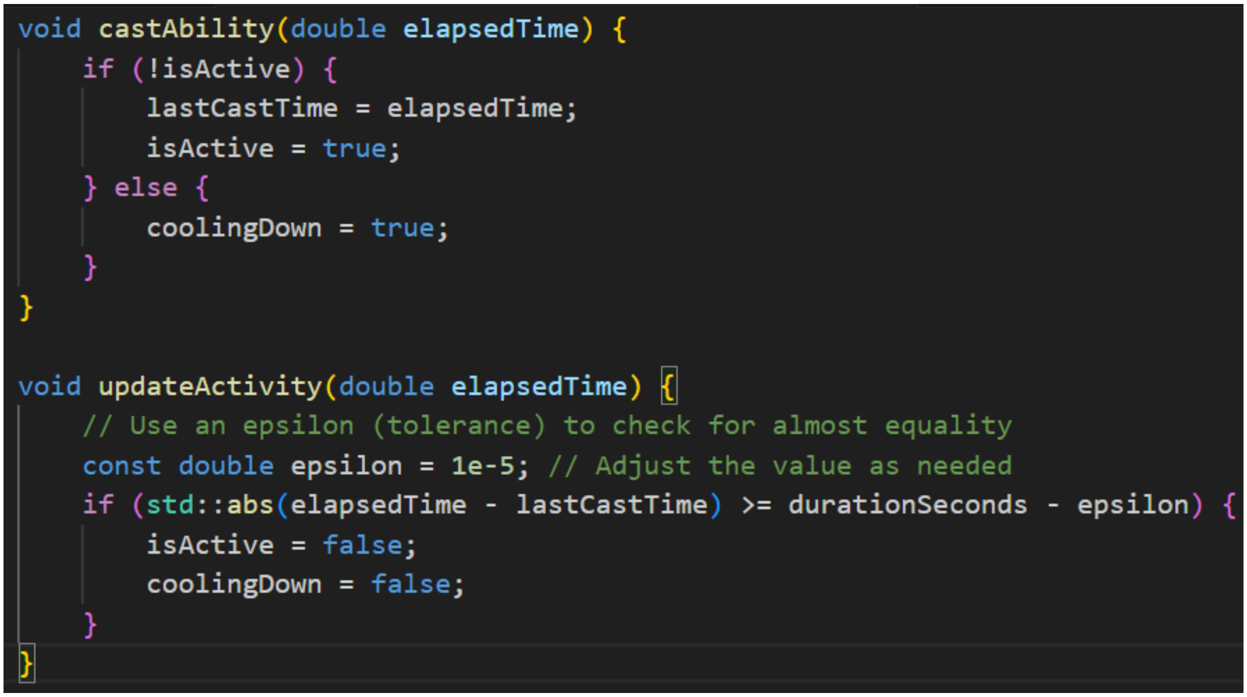
And it made the change I requested. This would be the bane of my existence for the rest of the project. I tested the new code, and promptly realized I had made a mistake, and ChatGPT's code was correct. I asked it to revert the change, and it happily agreed, and spat back out the same code with my error in it. I told it that it was right and I was wrong, and it agreed, and gave me the same incorrect code. Finally I resorted to straight-up telling it the exact code to use:



Please replace the comparisons in canDealDamage and updateActivity with if (std::abs(elapsedTime - lastCastTime) >= durationSeconds - epsilon).

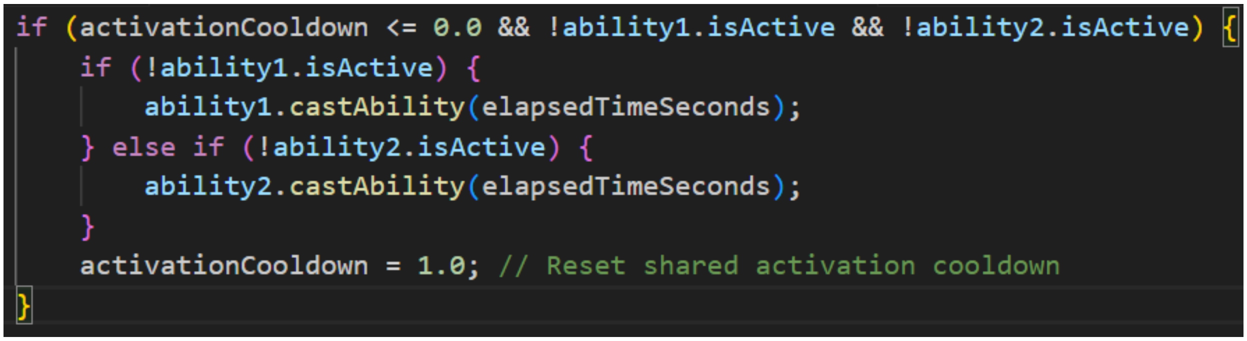
Finally, mercifully, it fixed the functions, and the code worked more or less correctly to my specifications to that point. Progress!

The next step I figured wouldn't be too bad - in a real game, you wouldn't be able to cast two abilities at the same time, so I asked ChatGPT to make it so that only one ability could be cast at a time, and after an ability was cast, the loop would have to wait for one second before casting another ability. The results were…not great. It changed the ability class, giving each ability a member variable which limited that specific ability to being cast no more than once every second - an individual cooldown:



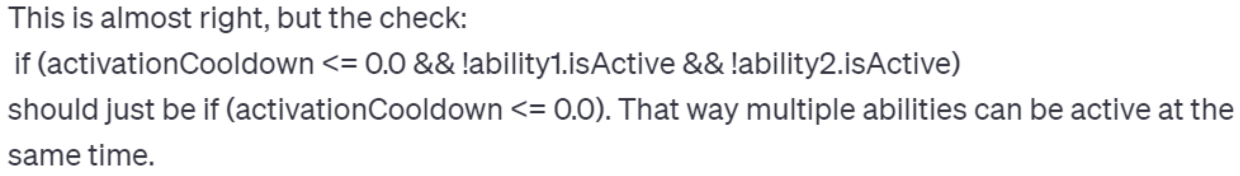
```
void castAbility(double elapsedTime) {
    if (!isActive) {
        lastCastTime = elapsedTime;
        isActive = true;
    } else {
        coolingDown = true;
    }
}

void updateActivity(double elapsedTime) {
    // Use an epsilon (tolerance) to check for almost equality
    const double epsilon = 1e-5; // Adjust the value as needed
    if (std::abs(elapsedTime - lastCastTime) >= durationSeconds - epsilon) {
        isActive = false;
        coolingDown = false;
    }
}
```

This isn't what I wanted, and also didn't work right anyway. What I was looking for was a shared cooldown between any and all abilities. After some back-and-forths, I had this:



```
if (activationCooldown <= 0.0 && !ability1.isActive && !ability2.isActive) {
    if (!ability1.isActive) {
        ability1.castAbility(elapsedTimeSeconds);
    } else if (!ability2.isActive) {
        ability2.castAbility(elapsedTimeSeconds);
    }
    activationCooldown = 1.0; // Reset shared activation cooldown
}
```

And I made a simple request:

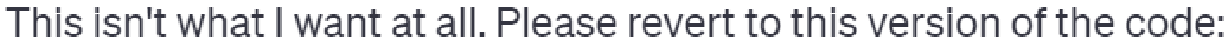


This is almost right, but the check:

if (activationCooldown <= 0.0 && !ability1.isActive && !ability2.isActive)

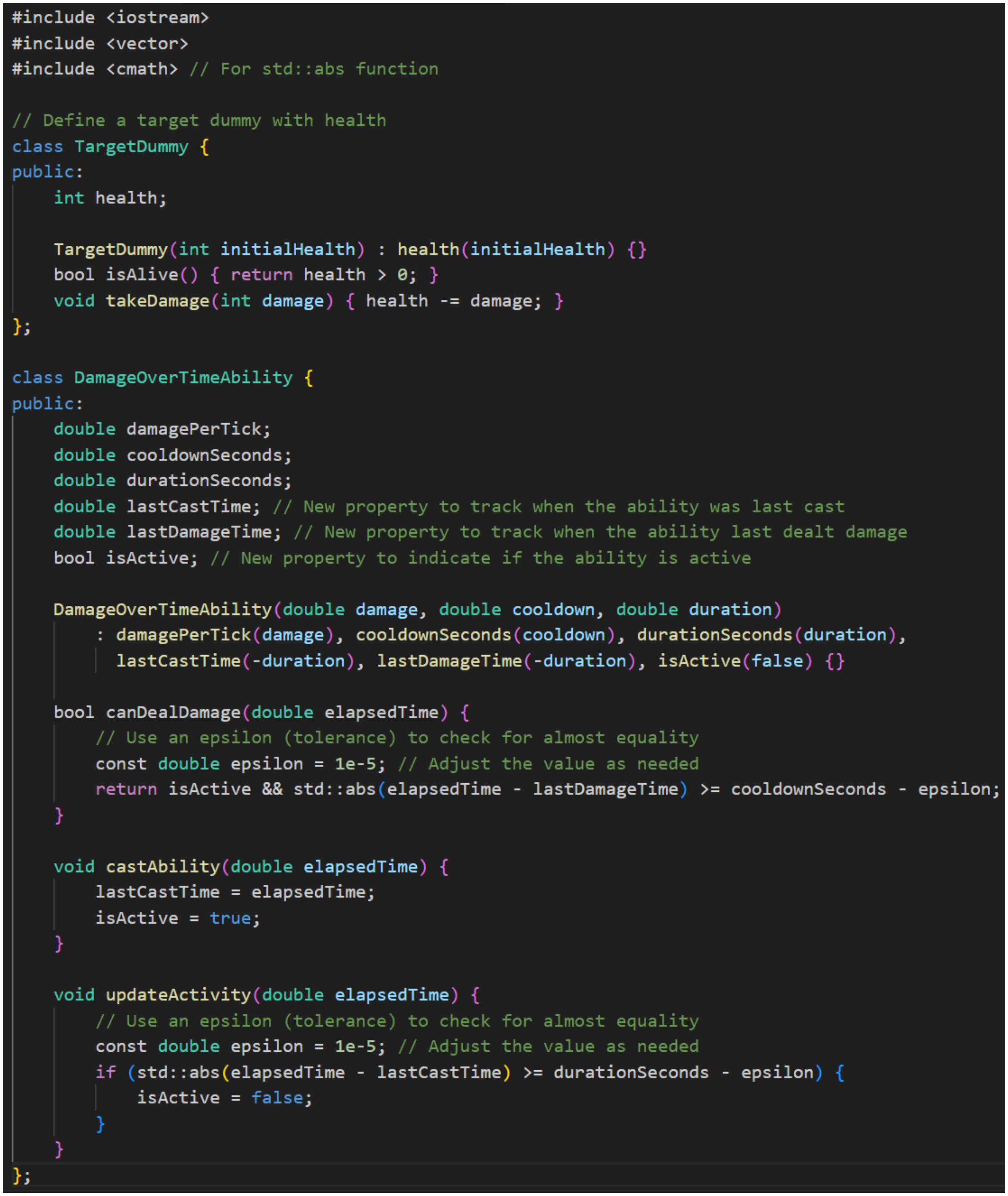
should just be if (activationCooldown <= 0.0). That way multiple abilities can be active at the same time.

Very easy. I told it exactly what to do. (At this point I had long given up on never telling it what code to write.) What it gave me was bizarre. It completely removed the if statement entirely, moving all of the cooldown related code back into the ability class, and basically going back to the code above. I told it that what it gave me wasn't what I asked for, and it responded by apologizing, and nuking the cooldown code entirely so that there was nothing left of it, as well as removing good sections of code, like the updateActivity function. At this point, I could see the entire project crumbling before my eyes, and so I made the most drastic request of the project:

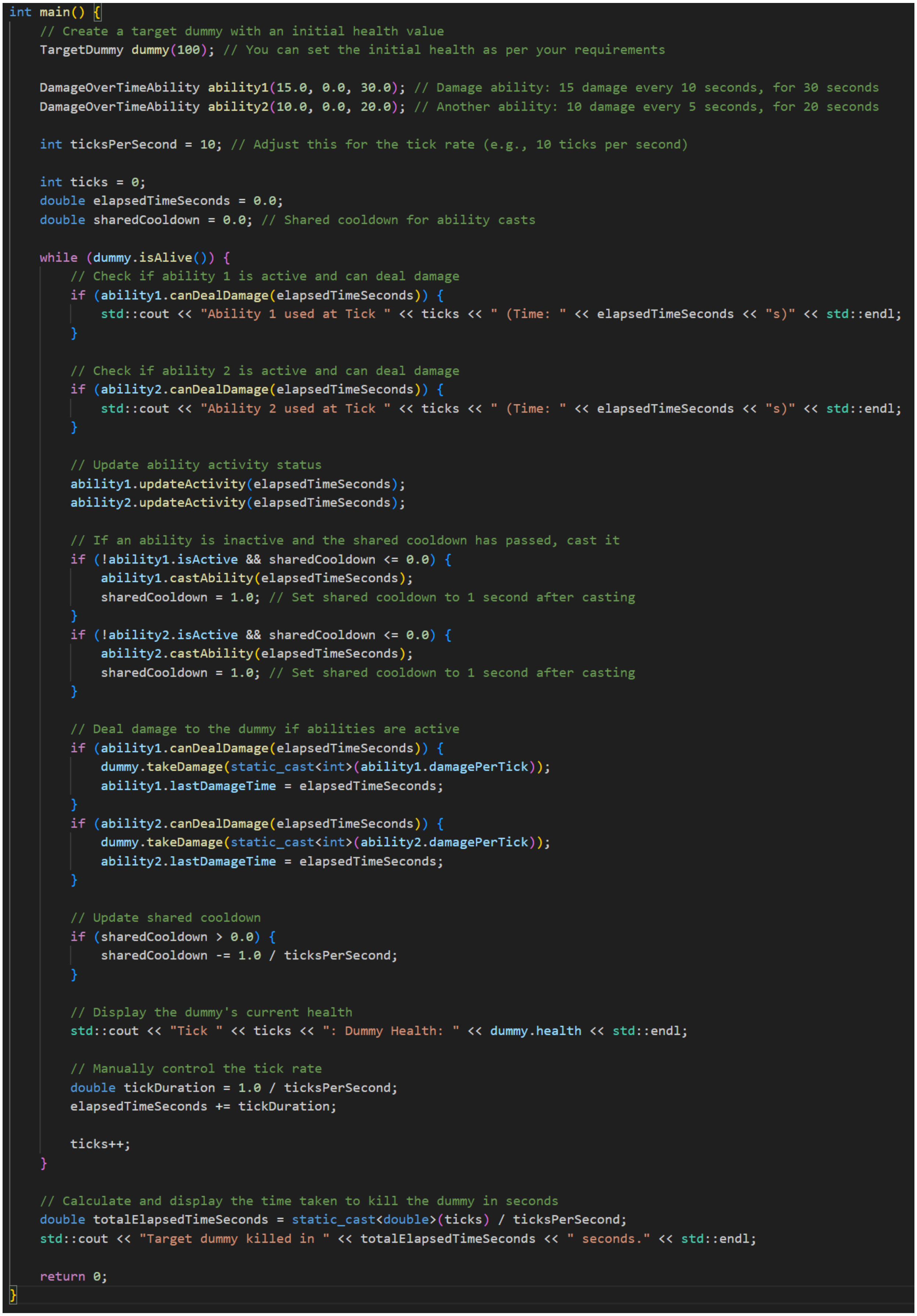


This isn't what I want at all. Please revert to this version of the code:

And I provided ChatGPT with the full text of the code that it had generated before I asked for a shared cooldown. This worked, and I got back to a functioning version of the code. Seeing that I could resort to this if I needed to, I decided to try again. I asked it to give the abilities a shared cooldown…and it did so almost flawlessly. It gave me just about exactly what I wanted. This is the full text of what I had at this point, for anyone interested (there won't be a quiz):



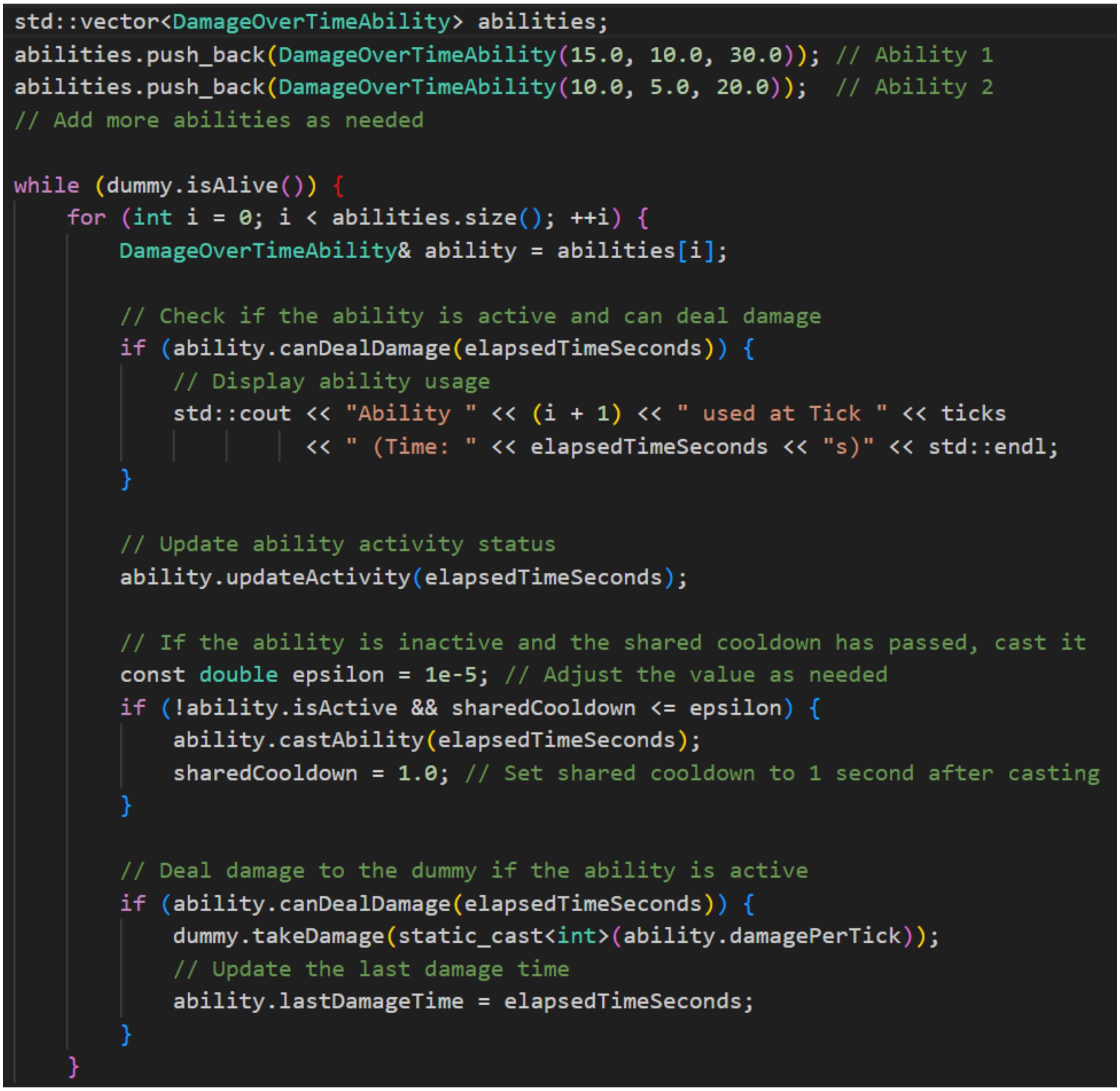
```cpp
#include <iostream>
#include <vector>
#include <cmath> // For std::abs function

// Define a target dummy with health
class TargetDummy {
public:
    int health;

    TargetDummy(int initialHealth) : health(initialHealth) {}
    bool isAlive() { return health > 0; }
    void takeDamage(int damage) { health -= damage; }
};

class DamageOverTimeAbility {
public:
    double damagePerTick;
    double cooldownSeconds;
    double durationSeconds;
    double lastCastTime; // New property to track when the ability was last cast
    double lastDamageTime; // New property to track when the ability last dealt damage
    bool isActive; // New property to indicate if the ability is active

    DamageOverTimeAbility(double damage, double cooldown, double duration)
        : damagePerTick(damage), cooldownSeconds(cooldown), durationSeconds(duration),
          lastCastTime(-duration), lastDamageTime(-duration), isActive(false) {}

    bool canDealDamage(double elapsedTime) {
        // Use an epsilon (tolerance) to check for almost equality
        const double epsilon = 1e-5; // Adjust the value as needed
        return isActive && std::abs(elapsedTime - lastDamageTime) >= cooldownSeconds - epsilon;
    }

    void castAbility(double elapsedTime) {
        lastCastTime = elapsedTime;
        isActive = true;
    }

    void updateActivity(double elapsedTime) {
        // Use an epsilon (tolerance) to check for almost equality
        const double epsilon = 1e-5; // Adjust the value as needed
        if (std::abs(elapsedTime - lastCastTime) >= durationSeconds - epsilon) {
            isActive = false;
        }
    }
};
```



```cpp
int main() {
    // Create a target dummy with an initial health value
    TargetDummy dummy(100); // You can set the initial health as per your requirements

    DamageOverTimeAbility ability1(15.0, 0.0, 30.0); // Damage ability: 15 damage every 10 seconds, for 30 seconds
    DamageOverTimeAbility ability2(10.0, 0.0, 20.0); // Another ability: 10 damage every 5 seconds, for 20 seconds

    int ticksPerSecond = 10; // Adjust this for the tick rate (e.g., 10 ticks per second)

    int ticks = 0;
    double elapsedTimeSeconds = 0.0;
    double sharedCooldown = 0.0; // Shared cooldown for ability casts

    while (dummy.isAlive()) {
        // Check if ability 1 is active and can deal damage
        if (ability1.canDealDamage(elapsedTimeSeconds)) {
            std::cout << "Ability 1 used at Tick " << ticks << " (Time: " << elapsedTimeSeconds << "s)" << std::endl;
        }

        // Check if ability 2 is active and can deal damage
        if (ability2.canDealDamage(elapsedTimeSeconds)) {
            std::cout << "Ability 2 used at Tick " << ticks << " (Time: " << elapsedTimeSeconds << "s)" << std::endl;
        }

        // Update ability activity status
        ability1.updateActivity(elapsedTimeSeconds);
        ability2.updateActivity(elapsedTimeSeconds);

        // If an ability is inactive and the shared cooldown has passed, cast it
        if (!ability1.isActive && sharedCooldown <= 0.0) {
            ability1.castAbility(elapsedTimeSeconds);
            sharedCooldown = 1.0; // Set shared cooldown to 1 second after casting
        }
        if (!ability2.isActive && sharedCooldown <= 0.0) {
            ability2.castAbility(elapsedTimeSeconds);
            sharedCooldown = 1.0; // Set shared cooldown to 1 second after casting
        }

        // Deal damage to the dummy if abilities are active
        if (ability1.canDealDamage(elapsedTimeSeconds)) {
            dummy.takeDamage(static_cast<int>(ability1.damagePerTick));
            ability1.lastDamageTime = elapsedTimeSeconds;
        }
        if (ability2.canDealDamage(elapsedTimeSeconds)) {
            dummy.takeDamage(static_cast<int>(ability2.damagePerTick));
            ability2.lastDamageTime = elapsedTimeSeconds;
        }

        // Update shared cooldown
        if (sharedCooldown > 0.0) {
            sharedCooldown -= 1.0 / ticksPerSecond;
        }

        // Display the dummy's current health
        std::cout << "Tick " << ticks << ": Dummy Health: " << dummy.health << std::endl;

        // Manually control the tick rate
        double tickDuration = 1.0 / ticksPerSecond;
        elapsedTimeSeconds += tickDuration;

        ticks++;
    }

    // Calculate and display the time taken to kill the dummy in seconds
    double totalElapsedTimeSeconds = static_cast<double>(ticks) / ticksPerSecond;
    std::cout << "Target dummy killed in " << totalElapsedTimeSeconds << " seconds." << std::endl;

    return 0;
}
```

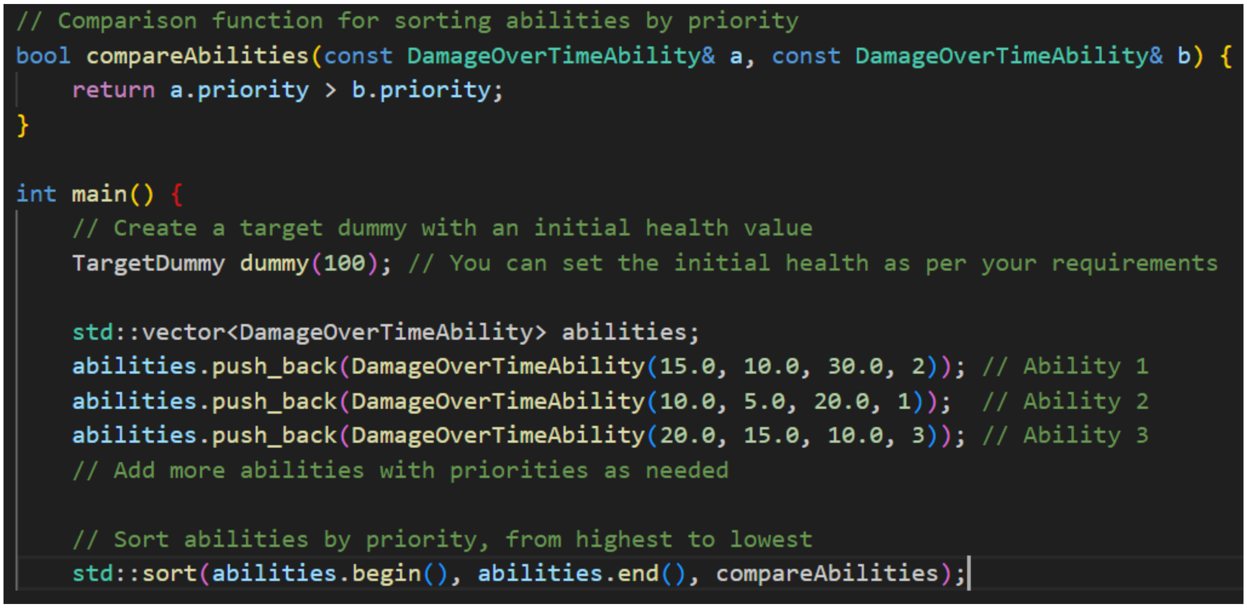
The only issue that I could see was that the comparison sharedCooldown <= 0.0 could run into floating point errors. And this is where my fatal mistake came back to haunt me. I asked it to fix the issue using an epsilon, which it did…and it simultaneously changed the comparisons in the canDealDamage and updateActivity functions back to the erroneous ones I had given it earlier. I asked it to fix those…and it broke the sharedCooldown comparison. I had to go back-and-forth with ChatGPT about a half-dozen times just to get these comparisons correct. And I would continue to battle with it over these comparisons for the rest of the project.

At this point, I was mostly happy with the functionality, but if I wanted this project to be even remotely useful, it would need to be able to handle more than two abilities at once. So I asked it to give me a vector of abilities instead of hard-coding in the functionality for two. I received this chunk of code in my main function (irrelevant parts omitted once again):



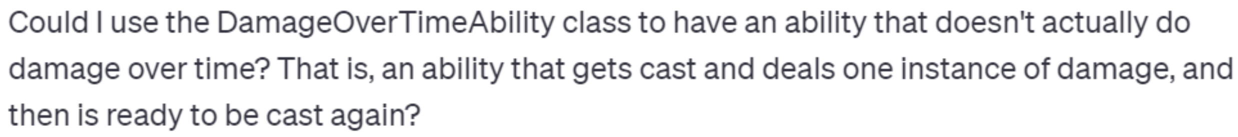
```
std::vector<DamageOverTimeAbility> abilities;
abilities.push_back(DamageOverTimeAbility(15.0, 10.0, 30.0)); // Ability 1
abilities.push_back(DamageOverTimeAbility(10.0, 5.0, 20.0));  // Ability 2
// Add more abilities as needed

while (dummy.isAlive()) {
    for (int i = 0; i < abilities.size(); ++i) {
        DamageOverTimeAbility& ability = abilities[i];

        // Check if the ability is active and can deal damage
        if (ability.canDealDamage(elapsedTimeSeconds)) {
            // Display ability usage
            std::cout << "Ability " << (i + 1) << " used at Tick " << ticks
                      << " (Time: " << elapsedTimeSeconds << "s)" << std::endl;
        }

        // Update ability activity status
        ability.updateActivity(elapsedTimeSeconds);

        // If the ability is inactive and the shared cooldown has passed, cast it
        const double epsilon = 1e-5; // Adjust the value as needed
        if (!ability.isActive && sharedCooldown <= epsilon) {
            ability.castAbility(elapsedTimeSeconds);
            sharedCooldown = 1.0; // Set shared cooldown to 1 second after casting
        }

        // Deal damage to the dummy if the ability is active
        if (ability.canDealDamage(elapsedTimeSeconds)) {
            dummy.takeDamage(static_cast<int>(ability.damagePerTick));
            // Update the last damage time
            ability.lastDamageTime = elapsedTimeSeconds;
        }
    }
```

I was quite happy with what ChatGPT gave me here; it worked flawlessly…other than breaking the epsilon comparisons again. I had it fix those again, and then I requested that each ability be given a priority, and for it to iterate over the abilities based on highest priority. Pertinent code:



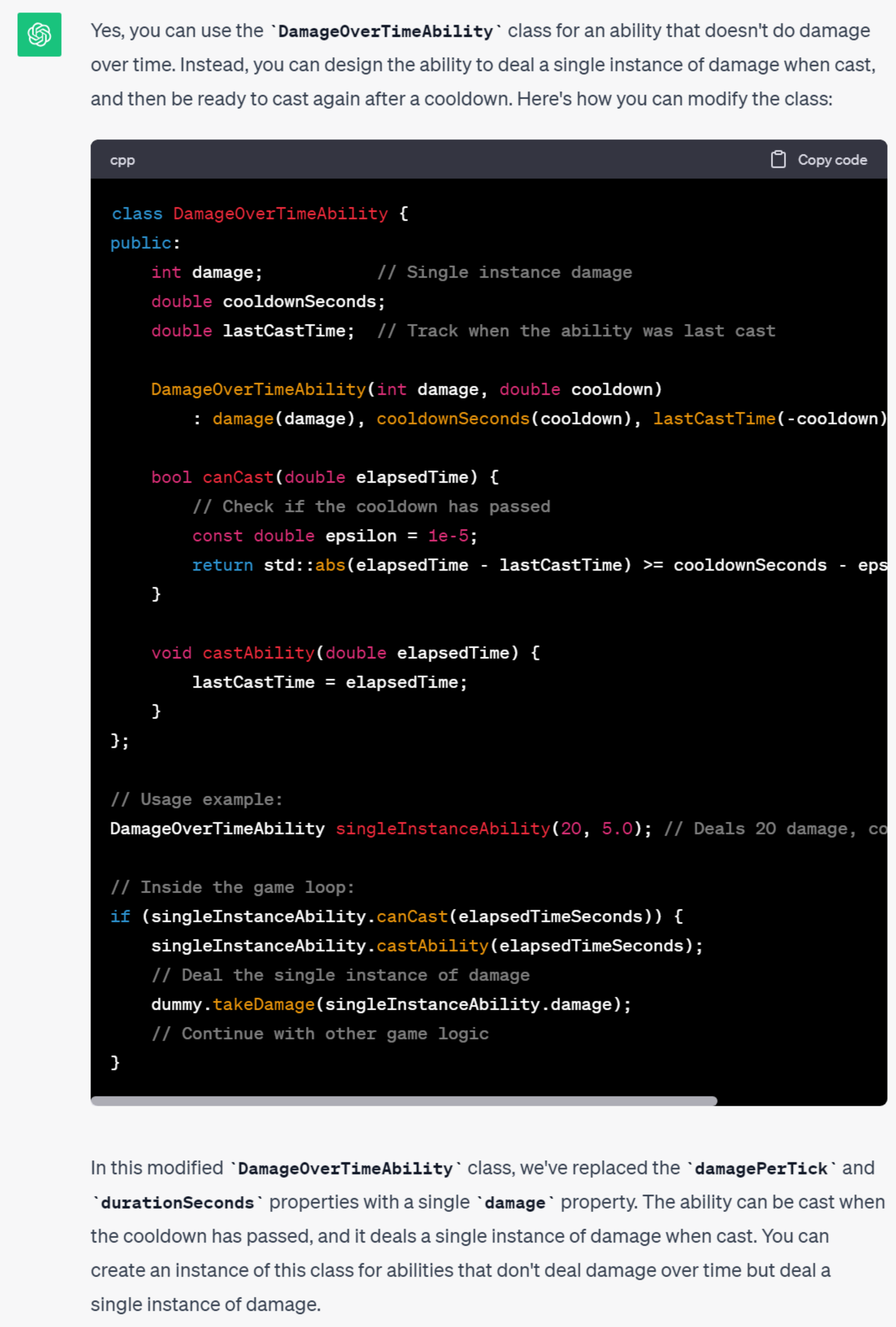
```cpp
// Comparison function for sorting abilities by priority
bool compareAbilities(const DamageOverTimeAbility& a, const DamageOverTimeAbility& b) {
    return a.priority > b.priority;
}

int main() {
    // Create a target dummy with an initial health value
    TargetDummy dummy(100); // You can set the initial health as per your requirements

    std::vector<DamageOverTimeAbility> abilities;
    abilities.push_back(DamageOverTimeAbility(15.0, 10.0, 30.0, 2)); // Ability 1
    abilities.push_back(DamageOverTimeAbility(10.0, 5.0, 20.0, 1));  // Ability 2
    abilities.push_back(DamageOverTimeAbility(20.0, 15.0, 10.0, 3)); // Ability 3
    // Add more abilities with priorities as needed

    // Sort abilities by priority, from highest to lowest
    std::sort(abilities.begin(), abilities.end(), compareAbilities);
```

It gave me a nice custom sort and sorted the abilities for me - great! All I had to do was have it fix the epsilon comparisons again, fix an error, fix the epsilon comparisons again, and fix some edge-case bugs, and I had a properly functioning program. Lovely. At this point I decided to test ChatGPT's knowledge of the program we had created. I asked it:



Could I use the DamageOverTimeAbility class to have an ability that doesn't actually do damage over time? That is, an ability that gets cast and deals one instance of damage, and then is ready to be cast again?

Now I already knew that the answer was yes. The original specification for the damage over time ability was (x) damage, every (y) seconds, for a total of (z) seconds. The code we had created worked such that when an ability was cast it immediately dealt damage, so it would be a simple matter of having an ability where y and z were set to 0. But what would ChatGPT's answer be? Well:

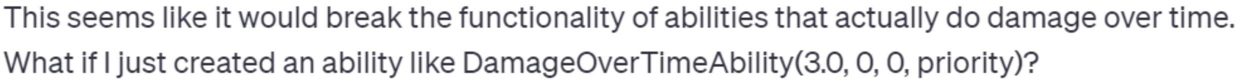


Yes, you can use the `DamageOverTimeAbility` class for an ability that doesn't do damage over time. Instead, you can design the ability to deal a single instance of damage when cast, and then be ready to cast again after a cooldown. Here's how you can modify the class:

```cpp
class DamageOverTimeAbility {
public:
    int damage;            // Single instance damage
    double cooldownSeconds;
    double lastCastTime;  // Track when the ability was last cast

    DamageOverTimeAbility(int damage, double cooldown)
        : damage(damage), cooldownSeconds(cooldown), lastCastTime(-cooldown)

    bool canCast(double elapsedTime) {
        // Check if the cooldown has passed
        const double epsilon = 1e-5;
        return std::abs(elapsedTime - lastCastTime) >= cooldownSeconds - eps
    }

    void castAbility(double elapsedTime) {
        lastCastTime = elapsedTime;
    }
};

// Usage example:
DamageOverTimeAbility singleInstanceAbility(20, 5.0); // Deals 20 damage, co

// Inside the game loop:
if (singleInstanceAbility.canCast(elapsedTimeSeconds)) {
    singleInstanceAbility.castAbility(elapsedTimeSeconds);
    // Deal the single instance of damage
    dummy.takeDamage(singleInstanceAbility.damage);
    // Continue with other game logic
}
```

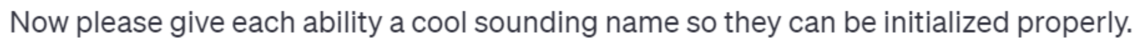
In this modified `DamageOverTimeAbility` class, we've replaced the `damagePerTick` and `durationSeconds` properties with a single `damage` property. The ability can be cast when the cooldown has passed, and it deals a single instance of damage when cast. You can create an instance of this class for abilities that don't deal damage over time but deal a single instance of damage.

…not exactly ideal. I pointed out:

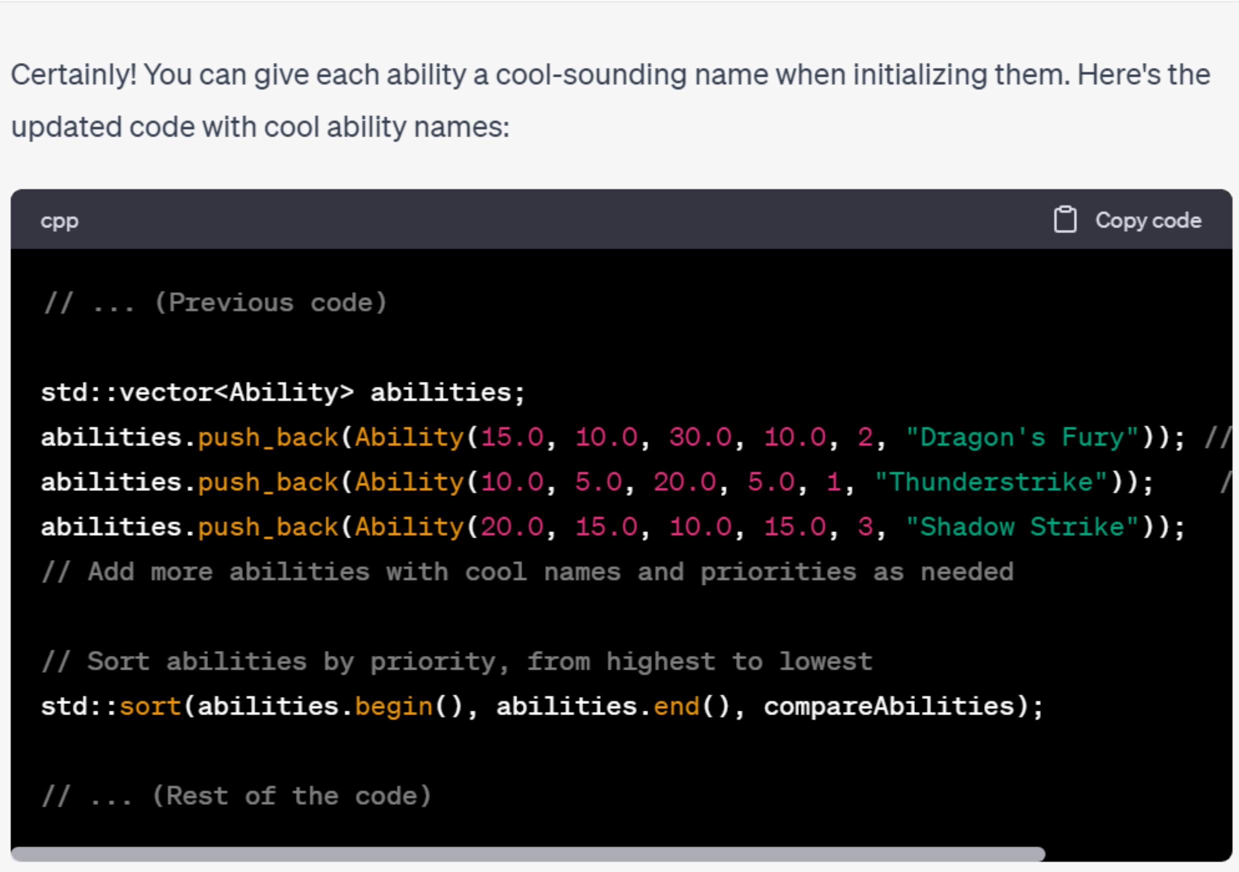


This seems like it would break the functionality of abilities that actually do damage over time. What if I just created an ability like DamageOverTimeAbility(3.0, 0, 0, priority)?

And it agreed that my way was better. I had it print out the entire body of code and everything looked good. I debated ending here, but decided there was one more bit of functionality I had to include. Remember the issue of individual vs. shared cooldowns? What I wanted back then was a shared cooldown, but it would be good to have both of them implemented. So I asked the AI, and it gave me what I can only describe as garbage. It broke the working code, and when I told it what was wrong, it broke the code even further. I ended up going back to the last resort method of telling it the entire body of code that it had given me before. After a second attempt I got it working right and I was ready to wrap the whole thing up. There were just a handful of smaller issues with code structure and quality that I wanted to fix. During these smaller fixes, ChatGPT often responded to me with small snippets of code instead of the entirety of the code, like so:



Now please give each ability a cool sounding name so they can be initialized properly.



Certainly! You can give each ability a cool-sounding name when initializing them. Here's the updated code with cool ability names:

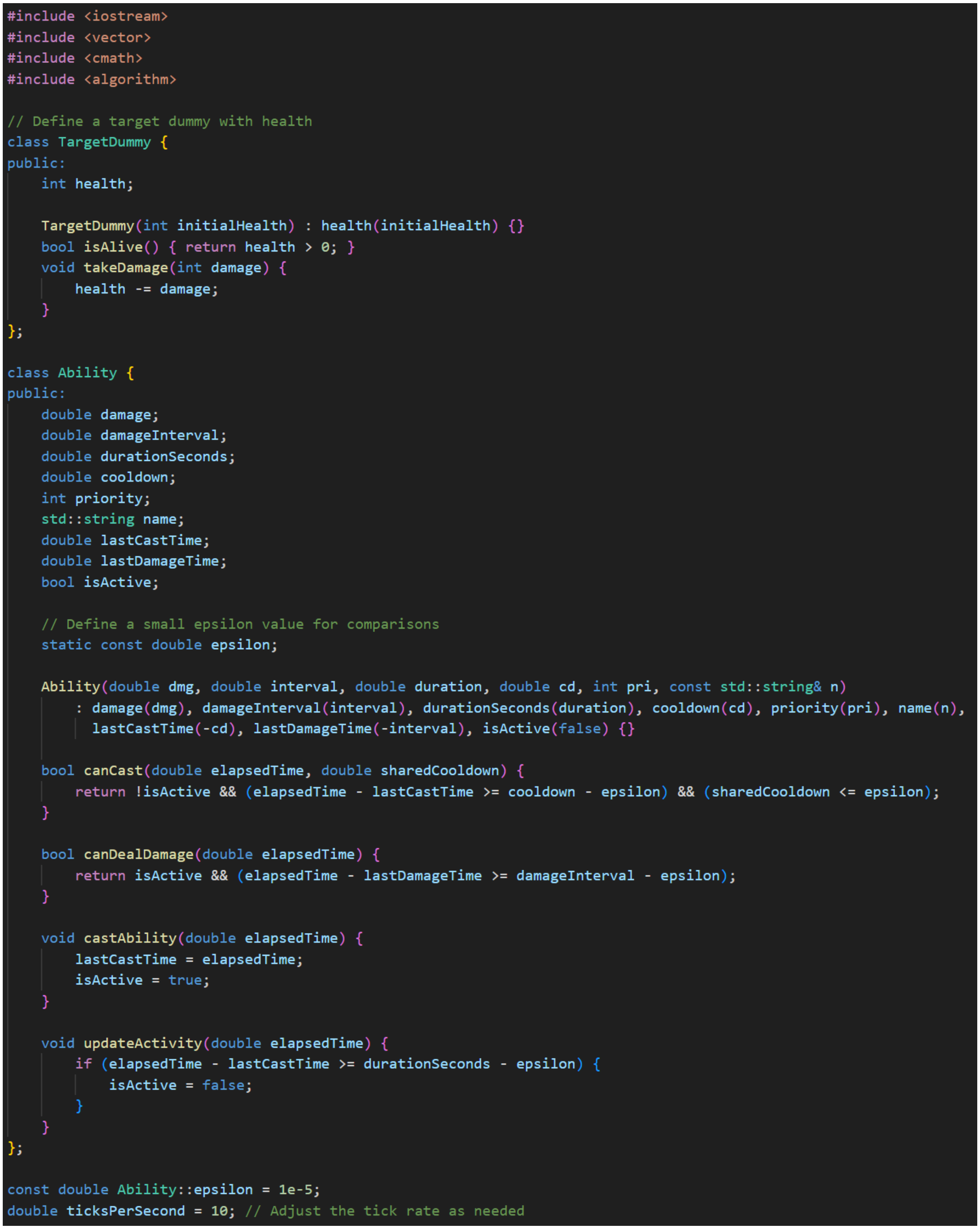
```cpp
// ... (Previous code)

std::vector<Ability> abilities;
abilities.push_back(Ability(15.0, 10.0, 30.0, 10.0, 2, "Dragon's Fury")); //
abilities.push_back(Ability(10.0, 5.0, 20.0, 5.0, 1, "Thunderstrike"));    /
abilities.push_back(Ability(20.0, 15.0, 10.0, 15.0, 3, "Shadow Strike"));
// Add more abilities with cool names and priorities as needed

// Sort abilities by priority, from highest to lowest
std::sort(abilities.begin(), abilities.end(), compareAbilities);

// ... (Rest of the code)
```

“No problem,” I thought. “This works well, and it doesn’t print out a hundred lines of code for one small change.” If only I had known what was coming next…because after making all the small changes I wanted, I had it print out the whole thing all together to make sure everything was right…and it had forgotten a bunch of the code. It took a good twenty or so requests to get it back to something close to what it should have been. But finally, I had something that resembled a proper simulation. You can change the list of abilities, and it will simulate hitting a target dummy with them based on the priorities you set. Other things can be changed as well, such as the tick rate and the shared cooldown. In an accurate simulation, of course, there's a lot more that goes into the damage of an ability than just a flat number. Buffs, debuffs, critical hits - I have a lot more to improve on before I can use it to do my testing for me - but that's a project for another day.

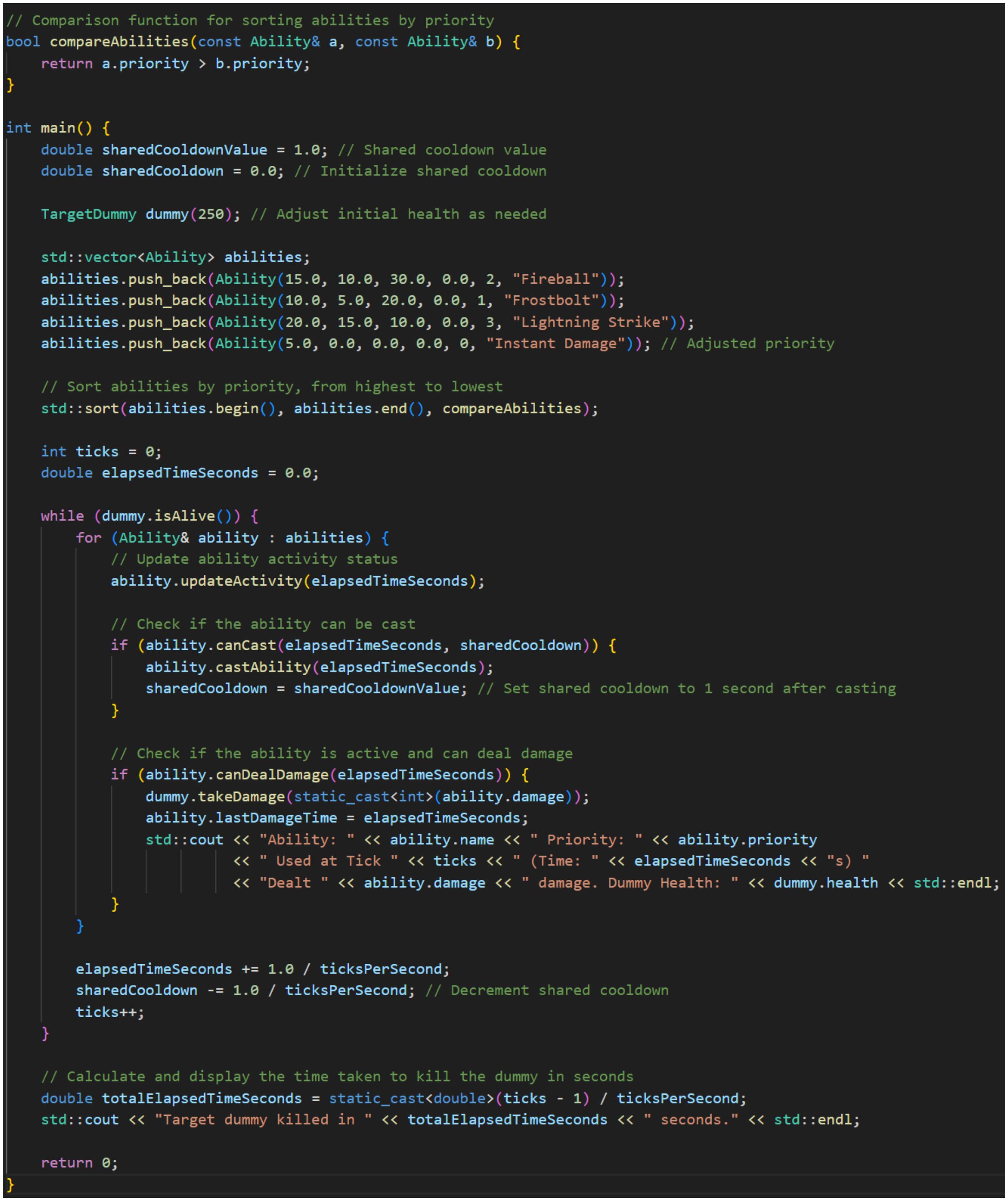
What did I learn, though, about using ChatGPT for projects like this? In no particular order:

- ChatGPT does great with smaller projects and individual snippets, but struggles to keep larger projects cohesive
- It does not tell you you're wrong - if you tell it to do something a certain way, it will absolutely do it
- Being very specific with what you want is more likely to get you what you want
- Asking it to print out the entire body of code instead of snippets seems to help it remember the whole thing
- Asking it to stop generating code using sleep_for worked, asking it to always use a certain epsilon comparison did not work
- Even if you can get it to generate code that works, it can't always give you good answers about how or why it works
- The style and readability are generally quite good; it loves to put helpful comments in the code
- On the other hand, it seems to like hard-coded values, so if you want those values to be easily changed, watch out for that

Full body of code:



```cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>

// Define a target dummy with health
class TargetDummy {
public:
    int health;

    TargetDummy(int initialHealth) : health(initialHealth) {}
    bool isAlive() { return health > 0; }
    void takeDamage(int damage) {
        health -= damage;
    }
};

class Ability {
public:
    double damage;
    double damageInterval;
    double durationSeconds;
    double cooldown;
    int priority;
    std::string name;
    double lastCastTime;
    double lastDamageTime;
    bool isActive;

    // Define a small epsilon value for comparisons
    static const double epsilon;

    Ability(double dmg, double interval, double duration, double cd, int pri, const std::string& n)
        : damage(dmg), damageInterval(interval), durationSeconds(duration), cooldown(cd), priority(pri), name(n),
          lastCastTime(-cd), lastDamageTime(-interval), isActive(false) {}

    bool canCast(double elapsedTime, double sharedCooldown) {
        return !isActive && (elapsedTime - lastCastTime >= cooldown - epsilon) && (sharedCooldown <= epsilon);
    }

    bool canDealDamage(double elapsedTime) {
        return isActive && (elapsedTime - lastDamageTime >= damageInterval - epsilon);
    }

    void castAbility(double elapsedTime) {
        lastCastTime = elapsedTime;
        isActive = true;
    }

    void updateActivity(double elapsedTime) {
        if (elapsedTime - lastCastTime >= durationSeconds - epsilon) {
            isActive = false;
        }
    }
};

const double Ability::epsilon = 1e-5;
double ticksPerSecond = 10; // Adjust the tick rate as needed
```



```cpp
// Comparison function for sorting abilities by priority
bool compareAbilities(const Ability& a, const Ability& b) {
    return a.priority > b.priority;
}

int main() {
    double sharedCooldownValue = 1.0; // Shared cooldown value
    double sharedCooldown = 0.0; // Initialize shared cooldown

    TargetDummy dummy(250); // Adjust initial health as needed

    std::vector<Ability> abilities;
    abilities.push_back(Ability(15.0, 10.0, 30.0, 0.0, 2, "Fireball"));
    abilities.push_back(Ability(10.0, 5.0, 20.0, 0.0, 1, "Frostbolt"));
    abilities.push_back(Ability(20.0, 15.0, 10.0, 0.0, 3, "Lightning Strike"));
    abilities.push_back(Ability(5.0, 0.0, 0.0, 0.0, 0, "Instant Damage")); // Adjusted priority

    // Sort abilities by priority, from highest to lowest
    std::sort(abilities.begin(), abilities.end(), compareAbilities);

    int ticks = 0;
    double elapsedTimeSeconds = 0.0;

    while (dummy.isAlive()) {
        for (Ability& ability : abilities) {
            // Update ability activity status
            ability.updateActivity(elapsedTimeSeconds);

            // Check if the ability can be cast
            if (ability.canCast(elapsedTimeSeconds, sharedCooldown)) {
                ability.castAbility(elapsedTimeSeconds);
                sharedCooldown = sharedCooldownValue; // Set shared cooldown to 1 second after casting
            }

            // Check if the ability is active and can deal damage
            if (ability.canDealDamage(elapsedTimeSeconds)) {
                dummy.takeDamage(static_cast<int>(ability.damage));
                ability.lastDamageTime = elapsedTimeSeconds;
                std::cout << "Ability: " << ability.name << " Priority: " << ability.priority
                          << " Used at Tick " << ticks << " (Time: " << elapsedTimeSeconds << "s) "
                          << "Dealt " << ability.damage << " damage. Dummy Health: " << dummy.health << std::endl;
            }
        }

        elapsedTimeSeconds += 1.0 / ticksPerSecond;
        sharedCooldown -= 1.0 / ticksPerSecond; // Decrement shared cooldown
        ticks++;
    }

    // Calculate and display the time taken to kill the dummy in seconds
    double totalElapsedTimeSeconds = static_cast<double>(ticks - 1) / ticksPerSecond;
    std::cout << "Target dummy killed in " << totalElapsedTimeSeconds << " seconds." << std::endl;

    return 0;
}
```

ChatGPT conversation log:
https://chat.openai.com/share/61b56d0c-6599-4926-8e01-4a4d26a7eb27