Codémon: A Buggy World

A Debugging-Driven RPG for Learning Programming

Team Members

- Connor Allen — Project Planner
- Isaac Hutchison — UI / UX Design
- Lon Danna — GitHub & Infrastructure Coordinator

GitHub Repository:
https://github.com/Connor-Allen10/codemon

Communication: Discord
Version Control: GitHub (feature branches + pull requests)

Abstract (Executive Summary / TL;DR)

Codémon is a Pokémon-style role-playing game designed to teach one of the most critical yet under-practiced programming skills: debugging. Instead of writing code from scratch, players progress through the game by identifying and fixing bugs embedded directly into the world, battles, and creatures. By reframing debugging as the *core mechanic* rather than a punishment, Codémon offers an engaging, low-pressure way for students to build confidence, pattern recognition, and problem-solving skills essential to real-world software development.

Goal

The goal of Codémon is to help users practice and improve debugging skills in a fun, gamified environment. The system emphasizes understanding program behavior, spotting errors, and fixing bugs efficiently — skills that are central to professional programming but often underemphasized in traditional learning tools.

Current Practice

Most educational coding games and platforms focus on:

- Writing new code from scratch
- Solving algorithmic puzzles
- Debugging only as a failure state

In these systems, debugging is often treated as a secondary or punitive task rather than a primary learning objective. As a result, students may graduate with limited confidence in diagnosing real-world bugs.

Novelty

Codémon's novelty lies in making **debugging the main gameplay mechanic**:

- The game world itself is "buggy"
- Enemies behave incorrectly due to code errors
- Progression depends on diagnosing and fixing bugs

Rather than reinventing coding education, Codémon shifts focus to a *missing skill* by embedding debugging challenges directly into gameplay systems such as battles, exploration, and progression.

Effects (Who Cares?)

If successful, Codémon will:

- Reduce fear and frustration around debugging
- Help students develop faster bug-recognition skills
- Serve as a supplementary learning tool for CS students
- Provide a more realistic representation of professional programming work

---

1. Use Cases (Functional Requirements)

Use Case 1: Debugging Battles (Connor)

Actors: Player
Triggers: Player encounters a Codémon with broken behavior, battle is started
Preconditions: Player has a starting Codémon to battle with
Postconditions:
1) Player successfully fixes the bug and wins the battle.

2) The Codémon responds appropriately to the bug fix.
List of steps:
1) Player encounters a Codémon
2) Player goes to battle
3) Player tries to attack with their Codémon and observes its buggy behaviour
4) Player opens their Codémon's code and attempts to fix the bug.
5) Player returns to battle and the Codémon behaves correctly
6) Battle concludes

Use Case 2: World Progression Through Debugging (Isaac)

- Environmental obstacles (bridges, doors, paths) contain bugs
- Player fixes logic or syntax errors to unlock new areas
- Visual feedback confirms correct debugging

Actors: Player
Triggers: Player encounters a blocked path, door, bridge, or environmental obstacle that does not function correctly
Preconditions:
1) Player has access to the area containing the obstacle
2) The obstacle is controlled by buggy logic or syntax
Postconditions:
1) Player successfully fixes the bug controlling the obstacle
2) The obstacle functions correctly and allows access to a new area
3) Visual feedback confirms the successful fix
List of steps:
1) Player explores the world and encounters a blocked environmental obstacle
2) Player interacts with the obstacle and observes incorrect or broken behavior
3) Player opens the debugging interface associated with the obstacle
4) Player inspects the provided code snippet containing logic or syntax errors
5) Player corrects the bug and submits the fix
6) System validates the fix and updates the world state
7) The obstacle visually changes to indicate it is now functional
8) Player proceeds into the newly unlocked area

Use Case 3: Codémon Growth & Evolution (Lon)

Actors: Players
Triggers: A Codémon reaches the level required to evolve, but evolution isn't triggered.
Preconditions: Codémon is at the level required for evolving.
Postconditions: Evolution logic is corrected, Codémon transforms and updates its stats.
List of steps:

1. The player opens the Codémon menu and selects specific Codémon.
2. The player clicks the Inspect Logic button to see why evolution is failing.
3. The system displays an if statement: if current_level >= 100.

4. The player identifies that the constant 100 is a logic error for low-level evolution.
5. The player changes value to 10 and saves changes.
6. The Evolve button becomes active, the player clicks it to trigger the evolution.

Extensions: Correcting the code with cleaner logic (refactoring) grants a small evolve bonus.
Exceptions: If a player enters a non-numeric value, the system displays “Input Mismatch” and prevents the save.

---

2. Non-Functional Requirements

1. **Usability:**
    - Debugging interfaces must be intuitive for beginners
    - Clear visual feedback for correct/incorrect fixes
2. **Performance:**
    - Game must run smoothly on standard student laptops
    - Low latency for input and rendering
3. **Maintainability:**
    - Modular game and debugging systems
    - Clear separation between gameplay and bug logic

---

3. External Requirements

- **Error handling**: The product will include robust input-validation. The game should not crash if a player types something unexpected. For example, if the game asks for a number and the player enters a word or gibberish, the game should simply show an error message and let them try again rather than crashing.
- **Installability**: Anyone should be able to play the game without having to be a programmer. If it’s a computer program, we will provide a ready to use file. You shouldn’t have to download a bunch of tools to get the game to open.
- **Buildability**: We will provide the source code and clear instructions. The repository will include a set up guide to help install the C++ compiler and SFML headers.
- **Scope**: The project is scaled for 3 members, focusing on one polished zone and robust battle system rather than an over extended open world.

- Must use GitHub for version control and collaboration
- Must be implemented using C++
- Must comply with CS 362 project scope and timeline
- Must be publicly accessible for evaluation

---

4. Team Process Description

Software Toolset / Technical Approach

- **Language:** C++
- **Graphics & Input:** SFML (Simple and Fast Multimedia Library)
- **GitHub**: Version control and feature branches and pull requests
- **Architecture:**
    - Core game loop
    - Debugging challenge engine
    - Modular enemy and world systems

Bug challenges will be represented as controlled code snippets with predefined failure modes and validation logic.

Risk Assessment

Risk #1: Scope creep due to creative ambition

1) Likelihood of occurring: 5/10 (medium)
2) Impact if occurs: 4/10 (medium)
3) Evidence: Educated guess based on previous projects
4) Steps to reduce likelihood:
    a) Define a simple core gameplay loop early
    b) Implement minimum viable features first
    c) Treat advanced mechanics as stretch goals
5) Plan for detecting problem: Set concrete requirements with numeric limits. E.g: 3 types of Codémon, max map size 100x100, etc.
6) Mitigation plan: Pause implementation of newest, most ambitious version and roll back to the version that stays within requirement limits. Once requirements are polished and tested, resume implementation of stretch goals.

Risk #2: Technical issues, feature difficult to implement

1) Likelihood of occurring: 3/10 (low)
2) Impact if occurs: 7/10 (high)
3) Evidence: Educated guess based on previous projects
4) Steps to reduce likelihood:
    a) Plan each core feature as soon as possible so that this risk may only apply to optional features/stretch goals
5) Plan for detecting problem: Feature is taking longer than a few days to get working on a basic level

6) Mitigation plan: Modify features to make technical implementation easier.

Risk #3: Code is overwritten due to faulty version control

1) Likelihood of occurring: 2/10 (low)
2) Impact if occurs: 2/10 (low)
3) Evidence: Low risk because of GitHub automatically preventing pushing changes when you don't have the most updated version. Low impact because you can always roll back to a previous working version, however you can still potentially lose some work but that is also low risk
4) Steps to reduce likelihood:
    a) Communication – Ensure code is up to date before pushing to repo
    b) Keep a backup of the latest working version
5) Plan for detecting problem: Compare latest version on GitHub with your working version.
6) Mitigation plan: Restore previous version

Risk #4: Falling behind schedule due to other commitments

1) Likelihood of occurring: 7/10 (high)
2) Impact if occurs: 5/10 (medium)
3) Evidence: As the term progresses team members likely become busier with other classes, leading to less time available for working on the project. Also because there are no concrete due dates (not set by us) for the coding portion, it may be considered lower priority. The impact is medium because we can still deliver if we are behind, but the quality of the product may be more rushed.
4) Steps to reduce likelihood:
    a) Follow schedule
    b) Dedicate a number of hours to work every week
5) Plan for detecting problem: During weekly progress updates, assess where code is vs. where it should be
6) Mitigation plan: In-person team meeting to code together and catch up

Risk #5: Faulty test cases

1) Likelihood of occurring: 3/10 (low)
2) Impact if occurs: 5/10 (medium)
3) Evidence: We will most likely be able to develop tests that cover all of the practical cases with 80% code coverage. However there is a low chance that we miss a niche bug, this chance increases with how complicated our program becomes. If our test cases are not thorough enough, or if there is a test that presents a false positive, this could cause a problem.
4) Steps to reduce likelihood: Write tests before coding, do black box + white box testing, review tests to make sure all types of inputs are covered, as well as making sure each test makes logical sense before testing.

5) Plan for detecting problem: This is the kind of problem that will only be uncovered with more and more testing, so we must do lots of rigorous testing, especially at the end of the development process.
6) Mitigation plan: If there is a faulty test discovered, we will meet to rewrite the test and determine everything is actually working the way it is supposed to.

How our risk assessment has changed since the Requirements Elicitation: Since the last iteration of this document, our risk assessment become more focused on what can go wrong in the planning phase of the project, thinking ahead to ensure the project stays within scope, and how we can prevent becoming disorganized by planning things and starting early before jumping in.

Team Roles & Responsibilities

- **Connor Allen:** Project planning, gameplay logic, requirements tracking
    - These roles ensure the scope of the project stays in check, managing this is very important.
- **Isaac Hutchison:** UI/UX design, player interaction, visual feedback
    - The UI/UX designer role requires an experienced UI designer.
- **Lon Danna:** GitHub management, CI setup, documentation, integration
    - Justification: GitHub version control is an immensely important part of developing software in a team, so having one person be in charge of making sure everything is clean and organized is important.

Schedule/Timeline

Week	Focus	Tasks
1	Requirements, core design, repo setup	- Project Milestone 1 - GitHub Setup
2	Requirements Elicitation, project brainstorming	- Project Milestone 2 - Enhance living doc with use cases
3	Begin coding framework, plan system architecture	- Presentation
4	Define classes and core game mechanics, begin writing test cases	- Begin Project Milestone 3

5	Finalized project plan, begin coding base mechanics	- Begin mid-term presentation - Project Milestone 3
6	Program is executable, begin UI, user manual,	- Mid-Term presentation - Coding - Project Milestone 4
7	Finish basic functionality, enhance UI	- Project Milestone 5 - Write remaining test cases
8	Bug fixing & refinement, evaluate features	- Test program - Project Milestone 6 - Begin final presentation
9	Final polish & presentation	- Final rigorous testing - Project Milestone 7 - Final presentation

External feedback will be most useful around week 6 or 7 of our timeline, where the program has the functional requirements implemented, and we can iterate and improve based on test users' feedback.

Test Plan

Unit Testing

- Each class will be tested to verify that it works correctly by itself. Test with edge/boundary case inputs and different types, making sure that errors are properly handled.
- These tests will be automated and performed early, to make sure each member function does what it's supposed to.
- Classes:
    - Battle
    - Move
    - Party
    - Stats
    - DebugChallenge
    - Effects
    - ValidationResult
    - Validators
    - Context

    - Game
    - State
    - StateStack
    - BattleState
    - DebugState
    - MainMenuState
    - WorldState
    - Entity
    - Interactable
    - Map
    - Obstacle
    - Player
    - World

System (Integration) Testing

- .Once the unit tests pass, we must test how everything works together. Here is where we will test the main functionality of the program.
- These tests will be done manually, making sure that everything is performed as it would in a real execution environment.
- Modules:
    - Battle System
        - Battle
        - Move
        - Party
        - Stats
        - DebugChallenge
        - Effects
        - BattleState
        - DebugState
        - Game
    - World
        - World
        - WorldState
        - Entity
        - Interactable
        - Map
        - Obstacle
        - Player

Usability Testing

- .After the integration testing has been sufficiently completed, we will test the software with real users.
- We will test with users of varying levels of experience.
    - People who have played RPG games before

    - Average computer-proficient people
    - Less computer-proficient people
- We will provide the user with a task, and measure the amount of time taken to complete the task, noting points of hesitation. After the tests, we will interview the user to receive feedback and integrate it into the program.

Documentation Plan

User Guide

- The user guide will provide instructions on how to install and run the program, basic instruction on how to play the game, and links to coding help resources for beginner programmers who might not be familiar with C++.

Software Architecture

We are using a Layered Architecture Design. This means we keep the game visuals (the user interface) separate from the game rules (logic) and the save data, allowing us to develop the debugging engine independently from the world map. This will make it easier to fix one part without breaking others.

Major Software Components + Functionality

- Battle system
    - The Debugger: This is the special text box where the player types their fixes. It checks if the player's fix is correct.
- Map exploration
    - Buggy Environment (Visual Layer, SFML): This is what puts images on the screen and listens for your keyboard presses.
- Codemon evolution
    - Data Manager: This loads the information about Codemon and levels from files on the computer.
- Core Game Logic (The brain of the game)
    - Logic layer: this is the brains of the game. It decides if you hit a wall or if the battle should start.

Interfaces (How parts talk)

- The **Logic Layer** sends a "Request to Check Code" to the **Debugger**.
- The **Debugger** sends back a "Success" or "Failure" signal to the **Logic Layer**.

Data Storage

- We store all the game info in a JSON file. These are simple text files that look like lists. It's easy for us to add new Codemon just by typing in these text files.
    - Location: `/assets/data/species.json` and `/assets/data/map_events.json`.
    - Schema (Conceptual):
        - `ID`: Unique integers.
        - `Name`: String.
        - `BuggyCode`: The string containing the error.
        - `SolutionKey`: The specific substring or logic required to pass.

Architecture Assumptions

- Local Only: We assume all game assets and saved files are stored locally on the player's machine.

Alternative Decision 1

- Using text files (JSON) vs. a database. We chose text files because they are easier for a small team to edit quickly. A database would be overkill for a single player game.

Alternative Decision 2

- Central brain vs independent pieces. We chose a central brain (state machine) to control the game. While independent pieces are more flexible, a central brain is much easier for us to test and debug as a team of three.

Software Design

1) Battles (Debugging Engine)
    - `EditorUI`: Handles text input, cursor position, and the "Submit" button.
    - `SyntaxValidation`: Compares the player's edited string against the required solution.
2) Game Core Logic (game brain)
    - `GameStateManager`: Switches between "Battle" "Explore" & "Evolution" modes.
3) Map exploration (Visual Layer)
    - `RenderEngine`: Draws UI text to the screen using SFML.
    - `AssetCache`: Manages memory by ensuring textures are only loaded once.
4) Codémon evolution (Data Manager)
    - `JS0NLoader`: Parses text files into C++ objects that the game can read.

Coding Guidelines

C++ Style Guide
https://google.github.io/styleguide/cppguide.html
Why: It provides clear rules on memory management (smart pointers), which is where most C++ bugs occur.
