# Codémon: A Buggy World

*A Debugging-Driven RPG for Learning Programming*

## Team Members

- Connor Allen:  Project Manager – Responsible for sprint planning and task tackling.
- Isaac Hutchison: UI / UX Designer – Responsible for visual interaction layer (SFML), system design, Regex engine logic, and core state machine.
- Lon Danna — Software Developer & Test Engineer – Responsible for game logic implementation and the automated unit/integration testing suite.

**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.

---

## 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.

---

## 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.

---

## Effects (Who Cares?)

If successful, Codémon will:

- Reduce debugging anxiety (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. The bug in the Codémon's logic is identified and corrected by the player.
2. The Codémon's behavior updates in real time to match the fix.
3. The battle concludes successfully with the player winning.

**List of steps (Success Scenario):**

1. Player encounters a wild Codémon or Trainer.
2. Battle begins, player selects an attack
3. Player observes the attack has no effect or behaves incorrectly (e.g., deals 0 damage).
4. Player opens their Codémon's source code interface for the current battle.
5. Player identifies the logic error in the provided code snippet.
6. Player modifies the code and submits the fix.
7. System validates the fix.
8. Player returns to battle and the Codémon behaves correctly.
9. Battle concludes.

**Extensions/variations:**

- **Optimization Bonus:** If the player fixes the bug using fewer lines of code (refactoring), the Codémon gains a "Critical Hit" bonus for that turn.
- **Hint System:** If the player fails to find the bug after two attempts, they can use a "Scan" item to highlight the line containing the error.

**Exceptions (Failure Scenarios):**

- **Invalid Syntax:** Player submits code that contains syntax errors. The system displays a "Compiler Error" message and the Codémon skips its turn (takes damage from the enemy).
- **Incorrect Logic:** Player submits a fix that is syntactically correct but does not solve the functional bug. The attack fails again, and the player must try a different fix next turn.

## 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 or environmental obstacle (door, bridge) that is non-functional.
**Preconditions:** Player has reached the specific map coordinates of teh obstacle.
**Postconditions:**

1. The environment bug and resolved
2. The obstacle state changes (e.g., bridge extents, door opens).
3. Visual feedback confirms the successful fix and the area is now accessible.

**List of steps (Success Scenario):**

1. Player explores the world and encounters a bridge that won't extend.
2. Player interacts with the bridge control panel.
3. A code editor window appears showing the BridgeControl class logic.
4. Player identifies a boolean error (e.g., if (isPowered == false) { extend(); }).
5. Player corrects the logic and saves.
6. System validates the fix and updates the world state.
7. The bridge visually extends across the gap.
8. Player proceeds into the newly unlocked area.

**Extensions/Variations:**

- **Secret Paths:** Some obstacles are optional and lead to rare items if the player solves a more complex "Resource Management" bug.

**Exceptions (Failure Scenarios):**

- **Logic Deadlock:** Player submits code that creates an infinite loop. The system prevents the save and warns the player that the logic is "Too Unstable" to implement.
- **Out of Scope:** Player attempts to edit code outside of the editable "comment blocks." The system prevents changes to the core engine code.

## Use Case 3: Codémon Growth & Evolution (Lon)

**Actors**: Players
**Triggers**: A Codémon reaches the level required to evolve, but evolution isn't triggered.
**Preconditions**: The Codémon's current_level variable is greater than or equal to its evolution_threshold.
**Postconditions (Successful Scenario)**:

1. The evolution logic is manually debugged and corrected.
2. The Codémon undergoes a transformation animation.
3. Stats and sprite assets are updated to the evolved form.

**List of steps**:

1. The player opens the Codémon menu and selects specific Codémon.
2. Player notices the "Evolve" button is greyed out despite meeting level requirements.
3. Player selects "Inspect Logic."
4. System displays the evolution check: if (level > 100).
5. Player identifies that 100 is an incorrect constant for a low-level evolution.
6. Player changes the value to 10 and submits.
7. The "Evolve" button becomes active; Player clicks it.
8. Evolution completes.

**Extensions/Variations**:

- **Branching Evolution:** By changing a variable type (e.g., evolution_type = "Fire" to "Electric"), the player can choose which form the Codémon takes.

**Exceptions (failure Scenarios)**:

- **Input Mismatch:** Player enters a non-numeric string (e.g., "Ten") into the numeric level field. The system displays an "Input Mismatch" error and reverts the value.
- **Value Overflow:** Player enters a value that is impossible (e.g., -5). The system validates the range and rejects the change.

## 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.

- **Language:** C++ (Strictly enforced).
- **Graphics:** SFML (Simple and Fast Multimedia Library).
- **Buildability:** Must include a CMakeLists.txt for cross-platform compilation.

---

# 4. Team Process Description

## Technical Approach

Our approach centers on a Curated Debugging Dataset and a Modular Validation Engine.

- **Bug Tiering:** Bugs are categorized into **Syntax** (missing symbols), **Logical** (infinite loops/wrong operators), and **Resource** (memory leaks).
- **The Validation Logic:** We utilize a **Pattern-Matching Engine** that validates user input against expected logic transformations rather than raw execution to ensure security and performance.
- **Interaction:** Users use a "Terminal Overlay" with **Lifelines**. If a player fails twice, the system highlights the erroneous line (simulating compiler feedback).
- **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 problems:** 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 problems**: 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 problems:** 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.

SInce our initial Requirements document, we have moved Schedule Slippage to a higher priority (High Likelihood/High Impact). This change is based on our recent change to build a custom C++ Regex/Pattern Matching engine, which we have identified as more technically demanding and time intensive than originally estimated.

---

## 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.

---

## Test Plan

**Specific Integration test Scenario: "The First Battle"**
This test ensures the communication between the **Core Logic**, the **Debugging Engine**, and the **Visual Layer** is functional.

1. **Trigger:** `WorldState` detects a collision with a "Buggy Codémon" and cal `GameStateManager`.
2. **Action:** `GameStateManage` pushes `BattleState` onto the stack. The `RenderEngine` loads the background..
3. **Interaction:** Player selects "Attack." Logic Layer detects a `DebugChallenge` is required and triggers the `EditorUI` overlay.
4. **Validation:** Player input is passed to the `SyntaxValidator`.

    a. If Pass: `ValidationResult` returns true; `BattleState` updates Codemon animation to “Attack” and applies damage.
    b. If Fail: `ValidationResult` returns false;  `BattleState` triggers an “Error” sound and enemy takes a turn.
5. **Conclusion:** Once HP reaches 0, `BattleState` is popped off the stack, and `WorldState` resumes.

## Bugs

Bug Tracking Strategy: We utilize GitHub Issues as our primary bug tracker, each issue is labeled by severity (critical, minor) and category (ui, logic, data). No feature is considered complete until its corresponding GitHub Issue is closed by a team member other than the one who wrote the code.

## Schedule/Timeline

| Week | Focus | Tasks |
|---|---|---|
| 1 | Requirements, core design, repo setup | - Project Milestone 1<br>- GitHub Setup |
| 2 | Requirements Elicitation, project brainstorming | - Project Milestone 2<br>- 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<br>- Project Milestone 3 |
| 6 | Program is executable, begin UI, user manual, | - Mid-Term presentation<br>- Coding<br>- Project Milestone 4 |
| 7 | Finish basic functionality, enhance UI | - Project Milestone 5<br>- Write remaining test cases |
| 8 | Bug fixing & refinement, evaluate features | - Test program<br>- Project Milestone 6<br>- Begin final presentation |
| 9 | Final polish & presentation | - Final rigorous testing<br>- Project Milestone 7<br>- 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 utilizing 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 Interface:** The `JSONLoader` provides a `std::map<int, SpeciesData>` that the `BattleSystem` queries whenever a new Codémon is encountered.
- **Visual Interface:** the Logic layer sends a list of `SpriteRequest` objects to the `RenderEngine` ever frame to determine what to draw.

**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: JSON Files vs. SQLite Database**

- **Alternative:** Using a SQLite Database.
- **Pros of Alternative:** Better data relationships and faster searching for huge datasets.
- **Cons of Alternative (Why we didn't pick it):** It adds complex dependency (SQL) and makes it harder for the team to manually type in new bug challenges. JSON is more readable.

**Alternative Decision 2: Centralized State Machine vs. Independent Objects**

- **Alternative:** Decentralized objects where each entity handles its own state.
- **Pros of Alternative:** More flexible and allows features to be developed in total isolation.
- **Cons of Alternative (Why we didn't pick it):** It creates spaghetti code in small projects. A Central Brain is easier to debug and ensures only one game state (like a menu) is active at a time.

---

## Software Design

1) **Battles (Debugging Engine):**
    a) `EditorUI`: Handles text input, cursor position, and the "Submit" button.
    b) `SyntaxValidation`: Compares the player's edited string against the required solution.
2) **Game Core Logic (game brain):**
    a) `GameStateManager`: Switches between "Battle" "Explore" & "Evolution" modes.
3) **Map exploration (Visual Layer):**
    a) `RenderEngine`: Draws UI text to the screen using SFML.
    b) `AssetCache`: Manages memory by ensuring textures are only loaded once.
4) **Codémon evolution (Data Manager):**
    a) `JSONLoader`: 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.
Graphics & Input: SFML (Simple and Fast Multimedia Library)

Enforcement Plan: Automated Formatting – We will use a `.clang-format` file in the repository to ensure code style is consistent across different editors.

---