Management & Ethics
Software Project Plan

CodeBot

An Educational Coding Game

COMP 4920

Joshua Lau
Josh Merideth
Mackenzie Baran
Matthieu Capuano

Product Overview

The goal of the project is to build a browser-based game which can be used to teach fundamental programming concepts. The user interface will consist of a text editor and a game board. The gameboard will be a grid which has the player's avatar placed on it, along with various other obstacles and objectives. The player will need to write code to direct the avatar around the board and solve the different challenges.

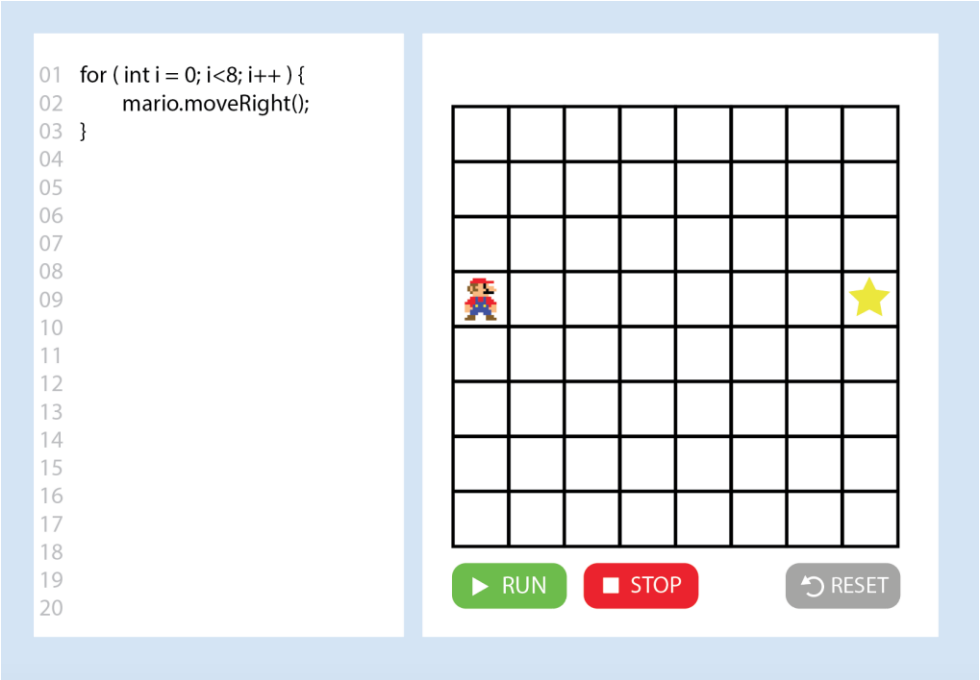

Figure 1. Basic UI mockup

Our intention is to gradually introduce more complicated programming concepts through progressively more difficult challenges. The users will need to write a program to solve each consecutive level/challenge. The idea is for the project to be used either individually, for self-taught purposes, or as part of a class. In the latter case, the teacher would be able to assign challenges to be solved.

Target Users & User Stories

Users

We anticipate the following types of users will use our product:

- **Teacher/Teaching assistant/Tutor** - In the context of a class or private tutoring session
- **Student/Player** - In the context of a class, private tutoring session or as a self-learning exercise
- **Challenge Developer -** The people writing the challenges and sample solutions

User Stories

We have compiled the following list of user stories, each from the perspective of a specific type of user listed above:

1. As a player, I would like to be able to choose a challenge to complete, so I can work on the specific problem I want to learn
2. As a player, I want to be able to write and edit my code, so that I can complete a challenge
3. As a player, I want to be able to save my progress, so I can pick up from where I left off
4. As a player, I want to be able to review my code from past challenges, so I can review how I passed it
5. As a player, I want to be able to join a class, so that I can complete the challenges my teacher assigns
6. As a teacher, I would like to choose challenges, so I can have my students complete specific ones
7. As a teacher, I would like to be able to create an account with teacher privileges, so I can direct my class
8. As a teacher, I would like to be able to create a class, so I can teach all of my students at once
9. As a teacher, I would like to be able to add students to my class, so I can create a class with all of my students
10. As a developer, I want to be able to create new challenges, so I can expand the currently available challenges for users
11. As a developer, I want to be able to access all currently available challenges, so I can see what users can work with
12. As a developer, I want to be able to modify all currently available challenges, so I can make necessary modifications in case of bugs, etc.
13. As a developer, I want to be able to block users if they somehow do something inappropriate, so I can keep the service child-friendly
14. As a player, I want to be rewarded for devising cleverer solutions so I have incentive to improve my past solutions.
15. As a player, I want to view my progress on challenges at-a-glance so I can be motivated to make more progress.
16. As a student, I want the text editor to highly syntax errors while coding, so I can fix faster

Minimum Viable Product and Features

Minimum Viable Product

The user stories above define the majority of overarching goals for the project. They can be split down into more specific subtasks. This is explained more thoroughly under the Sprint Planning section below. The MVP will have the following essential features:

- A basic User Interface with a textbox and a grid/gameboard (with avatar and goal)
- The ability to paste code into the textbox for each challenge
    - A later goal is to have a better text editor, in which you can write code, edit it, and get detection of syntax errors, unclosed parenthesis, etc. The MVP should just be able to read read and understand the code though
- The ability to run the pasted code
- The pasted code, when run, would either return an error message if the code is incorrect (not necessarily descriptive for the MVP) or run correctly and move the avatar accordingly
- The MVP should have at least one basic challenge, with a starting setting and victory condition (such as reading a target square)
    - Only needs one button to run, the “run” button

Additional Features

In addition to the MVP features defined above, the ones listed below will be added to the product over consecutive sprints. They are derived from the user stories and generally equivalent the subtasks identified for each story. The list is vaguely prioritised, with the highest priority items at the top of the list and lowest priority at the bottom.

- The ability to edit the code in the textbox
- The ability to have the code automatically step through while running, at understandable speed
    - The ability to adjust the speed of the automatic step-through
    - The ability to step through the code manually, one line at a time (e.g. using breakpoints)
- Create additional challenges (At least 5 by end of project) to teach
    - Basic syntax
    - Loops
    - Conditionals
    - Etc.
- An embellished User Interface (more than just a grid and textbox)
- The ability to step back through the code
- Syntax highlighting of errors in the code
- Students will have the ability to choose a challenge that they want to complete
- Students will have the ability to save their progress
    - This implies that users have some sort of account (may or may not be implemented depending on progress)

Sprint Plan

The plan is to undergo 5 week-long sprints, starting in week 9, and ending before the final deadline in week 13 (this includes a sprint during the break between weeks 9 and 10). They will begin and end each Tuesday - the day which coincides with our tutorial - so the first sprint will begin on Tuesday 19th of September. The details for the first sprint are below, the other sprints will be planned out during weekly meetings at the start of every sprint. These meetings will occur each week after the Seminar.

Sprint 1

The first sprint meeting occurred early, so the first sprint has already been planned out. The relevant **user stories for this sprint:** 2, 10, 11, 12. The goal is to get most of the MVP done by the end of the first sprint.

- Create the basic website/User Interface, including:
    - Textbox to code
    - Create the grid with avatar
    - Create “run” button that reads the code in, but does not necessarily adjust the UI yet
- Create the basic framework for the programming language
    - Decide on the *most basic* syntax - with the ability to augment this in a later sprint
    - Interpreter for the syntax => pretty hard to divide this up

Releases: Releases will occur every two sprints (so every two weeks) and during the final deadline. So at the end of sprints 2, 4, and 5.

Table 1. Sprint planning start and end dates

Sprint	Start Date	End Date
1	September 19th	September 26th
2*	September 26th	October 3rd
3	October 3rd	October 10th
4*	October 10th	October 17th
5*	October 17th	Before Final Due Date

*Releases at the end of sprint

Project Management

The project management tool that will be used throughout this project is Zenhub. A screenshot of the Zenhub setup after the first Sprint meeting is included below:

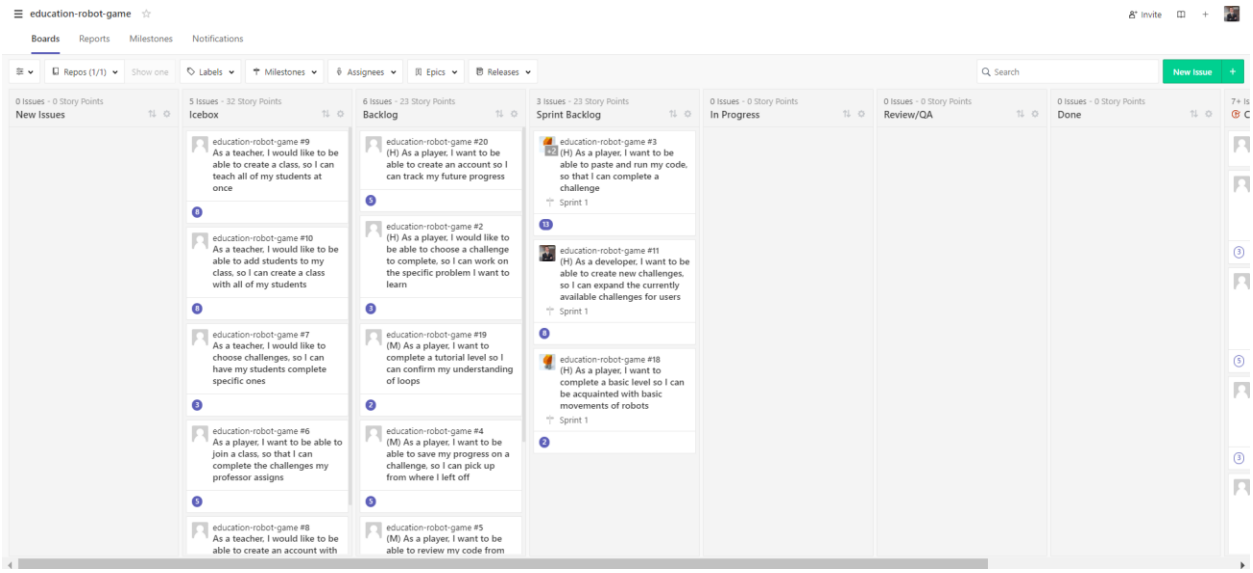

Figure 2. Zenhub Board setup after first sprint meeting

The setup is as follows. All user stories have been included in the Zenhub boards as issues, these are the boxes that can be seen in Figure 2. During sprints, additional issues can be created as the programmers realize that other features may be necessary or useful. These will remain in the “New Issues” pipe until the next sprint meeting after seminars on tuesday. During these meetings, all new issues will be assigned story points (an estimate of how much work they involve) by the group. These points can be seen on the bottom left of each box in Figure 2. On average, we estimate that each team member will complete 6 story points worth of work each week, which may vary week-to-week and individual-to-indiviudal depending on external commitments and responsibilities. The issues can then be moved to one of the following pipes depending on how important or urgent they are:

- **Icebox:** For non-urgent or critical features/stories that should be implemented at a later sprint
- **Backlog:** This is the stack of work that needs to be implemented after the current sprint is over, they are essentially prioritized during the following sprint planning. Things can move from the Icebox to the Backlog as the latter empties
- **Sprint Backlog:** These are the stories/features to be worked on and completed at the end of the current sprint. While worked on, they are moved to the “In Progress” box. That pipe should, ideally, be empty by the sprint.

During the sprint planning, items moved to the sprint backlog are assigned to one or more team members and then broken down into specific subtasks. One can tell that the stories above were assigned to group members by looking at the pictures on the top left of each box. Note also that these stories have been assigned to specific sprints, in this case all stories in the sprint backlog are part of sprint 1. This can be seen at the bottom of each box, above the story points. When

one of these detailed boxes is opened, the specific subtasks, assignees, and other details can be seen. An example is shown in Figure 3.

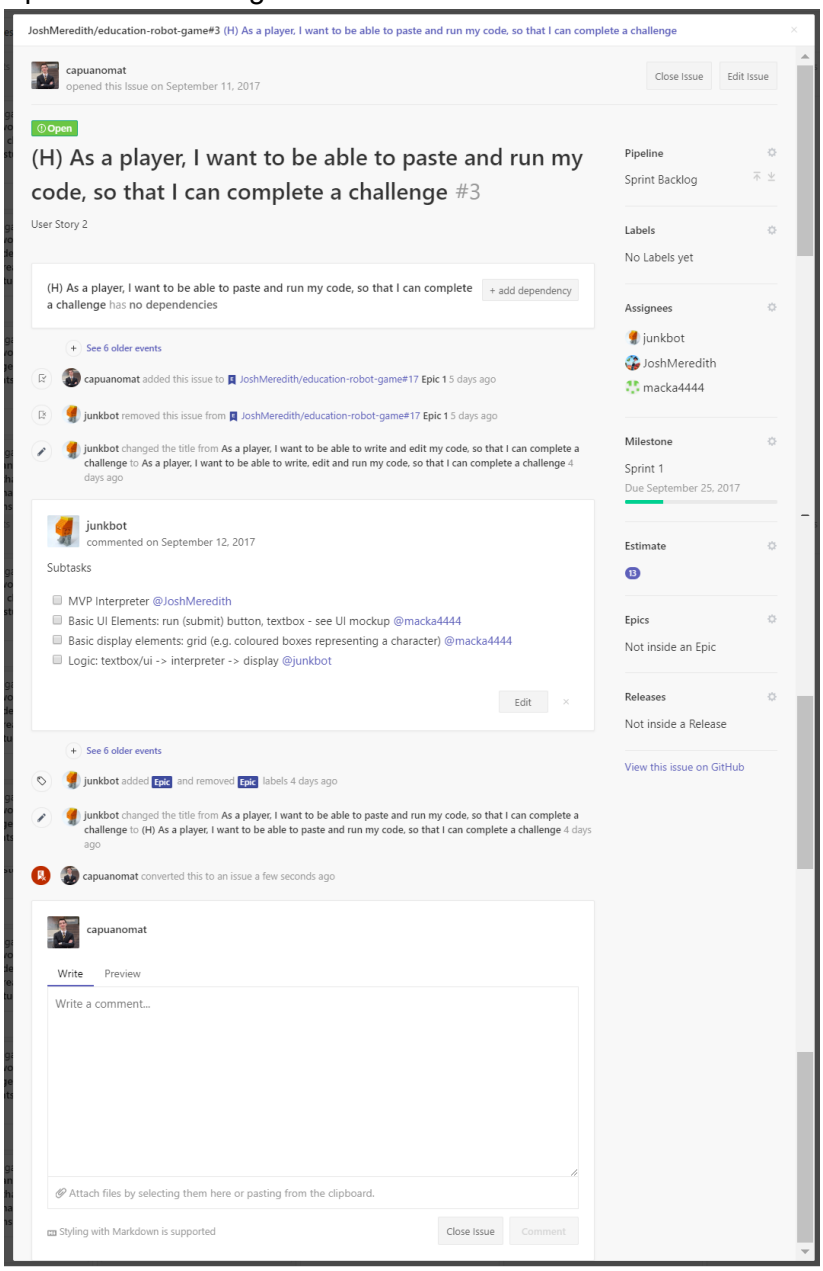

Figure 3. Example of a detailed user story currently being worked on

As an example, the story above was assigned to “junkbot” (Joshua Lau), “JoshMeredith” (Josh Meredith), and “macka4444” (Mackenzie Baran). Three assignees were chosen because this is a large task - with 13 story points. The subtasks can be seen under the “subtasks” box and are also each assigned to an assignee. In this case, there are four subtasks.

As members begin work on their issues, they move their box to the “In Progress” pipe. When they are done, they move it to the “Review Q/A” section for review by other members during the

next sprint planning. Finally, if all members agree that the story is complete, it is moved to the “Done” pipe, and then to the final “Closed” pipe (not visible in Figure 2) after the next release.

Note that the team chose not to use epics, as they were not deemed to add much usefulness to the current planning method. Instead, sprints and releases will be emphasized.

Team Organisation

Group Member Roles

The scrum roles will be arranged as follows:

- **Scrum Master**: Mackenzie
- **Product Owner**: Matthieu
- **Scrum Team**: Everyone

Communication

Group members will communicate primarily through a Facebook group chat for fastest response rate. We will also communicate through this google doc prior to the end of week 8 (when this will be finalized) and through the github for technical notes.

Meeting Schedule

Each Tuesday we will begin a new sprint. As such we will meet in person each Tuesday to have our sprint review and sprint retrospective for the past week, and sprint planning meeting for the coming week.

We will strive to conduct our daily standups in person when possible. Despite this, the reality of being busy uni students with conflicting schedules is such that it is unlikely that we will always be able to do this. When it is not possible to meet up in person (which we expect to be most days) we will conduct our daily standups online through our designated communication channels. We will conduct our standups at 10am each day. The major points of discussion for each meeting will be recorded in a shared Google Docs document.

Extreme Programming Practices

As a team we will engage in the following extreme programming practices:

- **Pair programming**: two teams of two will most likely be created to handle the front-end and back-end separately. These teams may often be reflected in the assigning of stories to group members during sprint planning.
- **Continuous integration:** through frequent pushes and pulls to/from the centralized github repository
- **Small releases:** Releases will occur relatively frequently (every two weeks)
- **Refactoring:** Done through frequent review of the source code design and programming

- **Collective code ownership:** Everyone will be responsible for the entirety of the code and be authorized to edit any part of it. Collective ownership will be encouraged by rotating the pairs programming each parts of the code

System Architecture

Our heavily algorithmic code will be in Purescript due to the convenience of parser combinators libraries, as well as the ability of ADTs to represent ASTs for the interpreter. Additionally, its javascript interop and ability to represent streaming computations make it suitable over Elm:

- Implement a parser combinator based parser for our language in Purescript using a library such as `purescript-parsing`
    - Expose a function `parseAST :: String -> AST` to the rest of the frontend (e.g. for passing to the interpreter and pretty printing/formatting)
- Implement an interpreter in Purescript which streams successive world states given the initial world state and AST using a library such as `purescript-run-streaming`
    - Expose functions to retrieve these successive world states
    - With these functions, implement a javascript iterator using `yield`.

We have chosen Typescript for the frontend to conveniently use javascript libraries while not sacrificing type safety:

- In Typescript, define an immutable world type with:
    - A method to step the world one game tick forward, taking the player move as an enumerated argument, returning the new world state, nullable for invalid moves
    - Methods to inspect the world state
    - Provide these methods to the interpreter using Purescript's FFI
- In the UI code:
    - Start with a base world state given by the backend based on level
    - Invoke the parser to obtain an AST when the user saves, etc
    - Run the interpreter with these and obtain values from the iterator either on a time delay or when the user presses the step button, depending on the run mode they select.

Our selections for the backend and third party services are:

- Deployment server: Nginx - for its simplicity
- Web framework: Python/Flask - since we have prior experience within our group
- Database: Sqlite - for its compatibility with Flask
- Authentication: OpenID - for its wide availability
- Continuous integration: Travis CI - for its GitHub integration.

Testing:

- Quickcheck on functions in the parser and interpreter
- Unit tests on the parser, interpreter, and world object
- A form of webdriver such as Selenium or Watir for UI and integration testing.