1 Introduction

Settlers of Catan is a bestselling European-style board game, with more than 15 million copies sold across 30 different languages.[1]

In this problem set, you will implement ~~**Steam**~~ **Settlers of** ~~**Catan**~~ **Gates Hall**. There are few constraints on how you can implement this project, and you will have greater freedom in making design decisions. However, this does not mean you can abandon what you have learned about abstraction, style, and modularity. Rather, this is an opportunity to demonstrate all of the skills that you have learned throughout the couse.

You should begin by carefully thinking about your overarching design. You must attend a graded **design review** where you will present and discuss your ideas with a course staff member.

At the end of the semester, there will be a tournament where you may submit a bot to compete with other students' bots. There will be free food, and the winners will receive bragging rights and have their names enshrined eternally in the CS 3110 Tournament Hall of Fame.

1.1 Reading this document

The non-native types discussed in this text are defined in `definition.ml`. Constants are defined in `constant.ml`, and use the following naming convention: they begin with a lowercase c, followed by a descriptive name in all caps. It may help to have these two files open for reference as you read the writeup.

Any updates to this writeup will be marked in red.

1.2 Point breakdown

- Design review: 10 points
- Game: 65 points
  1. Part A1: 20 points
  2. Part A2: 35 points
  3. Part B: 10 points
- Bot: 25 points

[1] http://en.wikipedia.org/wiki/The_Settlers_of_Catan

Contents

1 Introduction **1**

1.1 Reading this document 1

1.2 Point breakdown 1

2 Game Rules **2**

2.1 Setup 2

2.2 Goal 3

2.3 Initial Move 3

2.4 Dice Roll 3

2.5 Robber 3

2.6 Building 4

2.7 Development Cards 4

2.8 Trading 5

2.9 Trophies 5

3 Gameplay **6**

3.1 Initial Phase 6

3.2 Main Phase 6

4 State Representation **6**

4.1 Print Display 8

5 Game Implementation **9**

5.1 Part A – Gameplay 9

5.1.1 Part A1 – Basic Functionality 9

5.1.2 Part A2 – Full Moveset 9

5.1.3 Custom game type 10

5.1.4 handle_move 10

5.1.5 presentation 11

5.2 Part B – Trophies and Win Conditions 11

5.2.1 Extending handle_move 11

6 Bot Implementation **11**

6.1 name 11

6.2 initialize 12

6.3 handle_request 12

7 Source Code **13**

8 Running the Game **13**

9 Your Tasks **14**

10 Guidelines **14**
10.1 Preliminary Guidelines 14
10.2 Design Review Guidelines 15

11 Submission **15**

2 Game Rules

We will describe the rules here succintly, but there are a lot of details and corner cases which may need clarification. See the official rules and FAQ if you find any of the rules unclear.[2]

There are three rules that we will modify in our implementation: resources are unlimited, the robber reveals the stolen resource, and players can only make a limited number of trades per turn. If there are any other discrepancies, this writeup should be considered canonical, but please ask a member of course staff if you are unsure about any of the rules.

2.1 Setup

The game is played on a hexagonal board, which features 19 tiles and 9 surrounding ports.

Each hexagonal tile has a terrain of either hill, pasture, mountain, field, forest, or desert, and a corresponding resource. Hills produce brick, pastures produce wool, mountains produce ore, fields produce grain, forests produce lumber, and deserts produce nothing. Each tile also has a number, which indicates the dice roll that generates resources for all settlements bordering the tile (the dots underneath simply indicate the approximate probability of rolling that number with two standard six-sided dice). Finally, each tile has six corners where settlements can be erected, and six edges where roads can be built.

Each port serves two adjacent points along the coastline and provides advantageous trade ratios to players with settlements on those points.

[2]Please note that some of the FAQ only apply to 5–6 player games.

2.2 Goal

The goal of the game is to earn victory points. A player earns victory points directly by building towns and cities, drawing victory point development cards, and having the longest road or largest army in the game. (We have provided the longest road algorithm for you!) Other actions, such as building roads, trading, and playing development cards, do not directly give victory points, but may help indirectly.

The first player to have `cWIN_CONDITION` victory points on their turn immediately wins.

2.3 Initial Move

At the beginning of the game, players take turns making a series of initial moves. An initial move consists of building one town and one road from that town to any adjacent intersection. See the building rules in subsection 2.6 .

Each player makes two initial moves per game, starting the game with two towns and two roads. Each player receives a corresponding resource for every terrain tile adjacent to their second town.

2.4 Dice Roll

One of the first actions of every turn is to roll the dice. Dice rolls simulate the sum of rolling two 6-sided dice.

If the roll **is not** `cROLL_ROBBER`, resources are generated. Every settlement that borders a tile corresponding to the dice roll will generate resources for its owner unless it is inhabited by the robber. Each surrounding town produces `cRESOURCES_GENERATED_TOWN`, and each city produces `cRESOURCES_GENERATED_CITY` of that resource.

Our game has an infinite supply of resources, and you can ignore any outside rules corresponding to a limited number of resources. **This is our first of exactly three deviations from the official rules.** We make this decision to follow the spirit of the game more closely.

If the roll **is** `cROLL_ROBBER`, any player who has more than `cMAX_HAND_SIZE` resources in their inventory must immediately discard exactly the floor of half of the number of resources they have. After all necessary discards have been made, the active player activates the robber (see next subsection).

2.5 Robber

The robber can be activated by rolling `cROLL_ROBBER` or by playing a knight.

Once activated, the active player must move the robber to a new tile and specify the color of a player with a settlement surrounding that tile other than their own, if one exists. A resource will be stolen from that player, if their inventory is not empty, and given to the active player. The robber's new tile will not generate resources until the robber is moved.

In our game, the robber reveals the stolen resource to all players. Due to this change, players' resources are perfect information at all times. **This is the second deviation**

from the official rules. This change will allow you to write smarter bots with deeper strategies and less guessing.

2.6 Building

Victory points are most commonly acquired by building. Each buildable item has a resource cost associated with building. There are limits to the number of roads, towns, and cities that a player can build on the board.

- Building a `Card` allows you to draw a random development card from the remaining deck. Your development cards are hidden from all other players (though they can see how many you have), and are the only imperfect information in our formulation of the game. You may purchase cards as long as the deck is not empty.
- Building a `Town` allows you to build a basic settlement that you can place at any valid location. **A town can only be built if it is at least two road lengths away from any existing settlement**. This includes towns built during initial moves. Towns are worth `cVP_TOWN` victory points and gain `cRESOURCES_GENERATED_TOWN` resources from resource generation.
- Building a `City` allows you to upgrade one of your existing towns into cities. Cities are better than towns because they produce more resources upon resource generation (`cRESOURCES_GENERATED_CITY`), and are worth more victory points (`cVP_CITY`). If you previously ran out of towns to build, you may build an additional town after upgrading one to a city.
- Building a `Road` allows you to build a road that connects to an existing road that belongs to the player building it. Initial moves are exempted from this rule.

2.7 Development Cards

Development cards are purchasable items, drawn from a deck at random, and hidden from other players. Each card (except `VictoryPoint`) has a powerful effect to be used when you play it. You may choose to keep development cards in your hand for as long as you want, with no penalty or danger of loss. Other players can see how many you have at all times, but not what they are. Development cards can only be played during your turn, and they are instantly used and discarded when you play them. You may only play one development card per turn. The different development cards are:

- A `Knight` card activates the robber for you. **Unlike rolling a robber, knights do not trigger discards**. Every knight you play increments your army size by 1.
- A `VictoryPoint` card gives you an automatic victory point, which will be counted in all victory point calculations. Victory point cards cannot be played, only passively held for the rest of the game. Other players will not know how many of these victory points you have until the game is over.

- A `RoadBuilding` card allows you to build either one or two roads in any valid positions of your choice.
- A `YearOfPlenty` card allows you to gain either one or two free resources of your choice.
- A `Monopoly` card allows you to specify a resource type. All other players must give you all of that resource that they currently have in their inventory.

2.8 Trading

There are two types of trades that players can make on their turn.

- A maritime trade, by default, converts several of any one resource to a single instance of another resource. If the player has a seaside settlement at a port, it will offer a more favorable trade ratio if it is compatible with that resource. A seaside settlement is not required to make a maritime trade at the default exchange rate. Players can make an unlimited number of valid maritime trades per turn.
- A domestic trade is a request sent to another player, detailing the resources that you will give up, and the resources that you want from the other player. A valid trade proposal must offer resources that you have, and request resources that the other player has. Both costs must contain at least one resource. The other player will then decide whether to accept or reject the trade. Players can make at most `cNUM_TRADES_PER_TURN` trade proposals per turn.[3] **This is the third and final deviation from the official rules.** We make this change to prevent an (effectively) infinite number of trades per turn from stalling the game. To this end, we must count trades regardless of whether they are accepted or rejected.

2.9 Trophies

There are two trophies in the game, **Longest Road** and **Largest Army**. Only one player may possess a trophy at any time. Both trophies operate under a **"first no tie"** rule of possession:

Playing a knight increases a player's army size by 1. If upon playing a knight, a player's army is the largest of all players, and is at least `cMIN_LARGEST_ARMY` knights, then they receive the **Largest Army** trophy, which counts as `cVP_LARGEST_ARMY` victory points for as long as it is possessed. If playing a knight ties two players for the most knights played, the trophy does not change hands.

If after building a road (by any means), a player has the longest road of all players, at least `cMIN_LONGEST_ROAD` lengths, then you receive the **Longest Road** trophy, which counts as `cVP_LONGEST_ROAD` victory points for as long as you possess it. If building a road ties two players for the longest road, the trophy does not change hands. (**Longest Road** has been implemented; there is a function in Utils to help you with this.)

[3] Although this could be considered a limitation in our implementation, the number of unique, valid trades is potentially enormous, and it is not practical to consider.

3 Gameplay

We separate gameplay into two phases, the initial setup phase, and the main game phase.

3.1 Initial Phase

Game play is initialized with four players. One player is randomly selected to go first. In forward order, players make a first round of initial moves. Once all four players have gone, the order of play is reversed, and each player makes a second initial move, beginning with the player who went last. On the second round, each player receives their initially generated resources. After the second round is complete, the first player's turn begins the main phase.

3.2 Main Phase

Play is structured in turns, beginning with the first player from the initial phase. Turns proceed in forward order for the rest of the game. Turns contain some finite combination of the following moves:

- Roll the dice. This prompts resource generation, forces discards, and/or activates the robber. Your turn cannot end before you roll.
- Play a development card. This is the only move you can make before rolling the dice. You can only play one card per turn, either before or after rolling, and it cannot be a card that was purchased on the same turn.
- Build an item. At any point after rolling, you may build as many items as you can afford.
- Attempt a trade. At any point after rolling, you may make as many valid maritime and/or domestic trades as you can afford.
- End the turn. At any point after rolling, you may pass play to the next player.

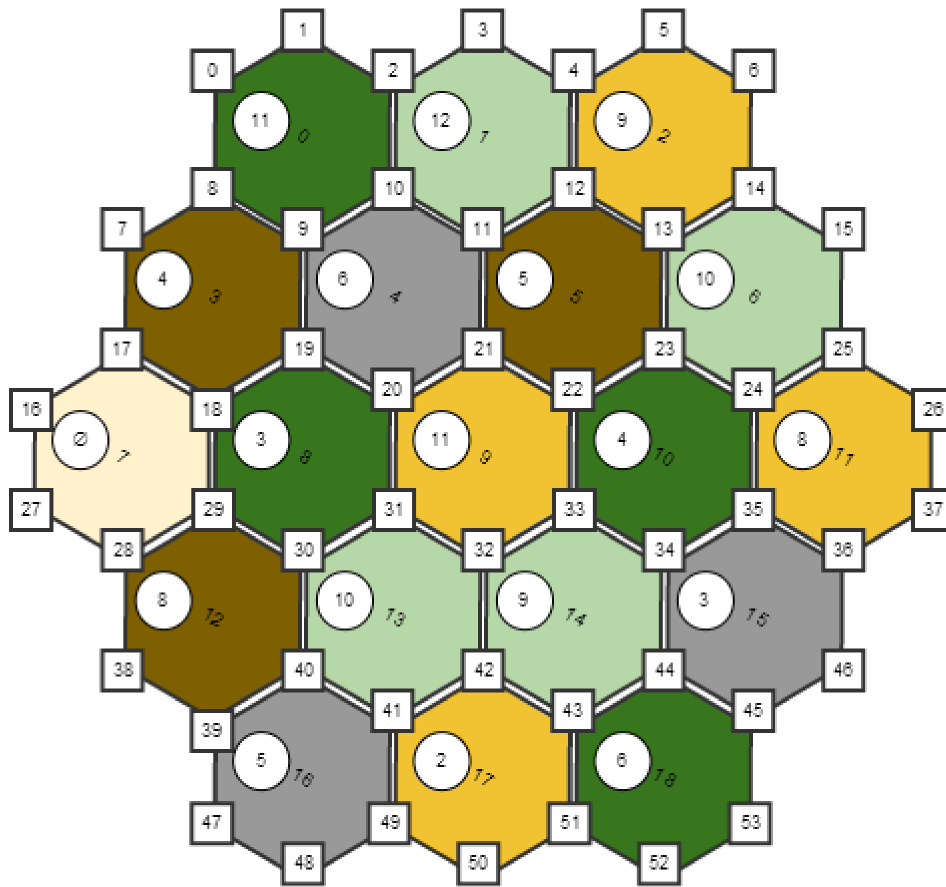
You may make as many valid moves as you wish during your turn. If at any point during the turn, your victory points exceed `cWIN_CONDITION`, you instantly win the game.

You might notice that with this formulation, it is possible for a game to go on for an arbitrarily long length of time. Because it is not part of the game rules, a turn-limit based trade condition is implemented in `play.ml` for practicality.

4 State Representation

Please see `definition.ml` for detailed type definitions.

For our implementation, we refer to each hex tile and intersection point by the numbers indicated below.

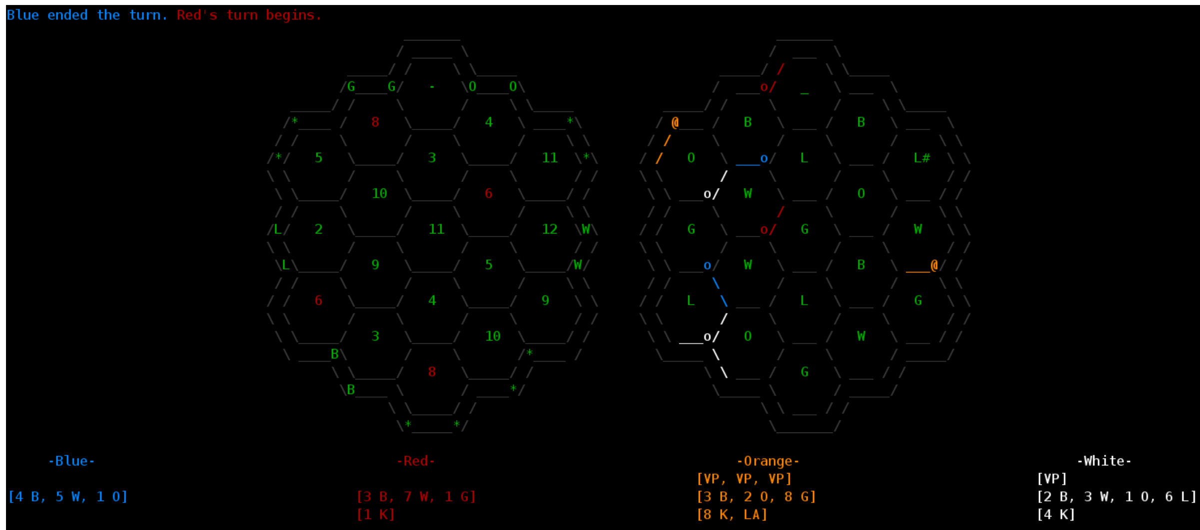


The distribution of hexes and roll markers here is the **Beginner's Board** featured in the official rules. This is the default board that you will play on.

We represent the tiles as a `hex list` of length `cNUM_PIECES`. Similarly, we represent the intersections as an `intersection list` of length `cNUM_POINTS` ordered by index.

4.1 Print Display

Here is an example of a random game in progress:



User Interface: Note that the user interface prints the board rotated clockwise 90 degrees. Because ASCII.

The most recent move is displayed in the top left corner.

The left board contains static information about the roll numbers for each hex (red numbers are high probability rolls) and the locations and resources of ports.

The right board shows the resource generated in each hex along with player roads and settlements. Towns are represented as **o**, while cities are represented as @. The robber is denoted by # in the hex that it is currently on.

Player information is split into three lines, the in-hand development cards, the in-hand resources, and the player trophies.

In the example above, Orange is clearly in the lead, with two cities on the board, three victory points in hand, and 8 knights already played, which also donates the Largest Army trophy.

5 Game Implementation

A large part of your implementation will be the functions in `game.ml`. These functions are currently stubs. You should think carefully about your design decisions here and consider any addition modules you might want before you start coding. There are also many useful functions in `shared/util.ml` that may save you time.

Because this is a substantial project, we split the game implementation into two parts. You are free to work in any way you like, but it may be helpful to implement and test these parts separately.

5.1 Part A – Gameplay

Part A involves implementing all of the game logic excluding trophies. This means that you may ignore the `trophies` associated with each player and do not have to account for largest army and longest road. You may also assume that the game is never won and return `None` in every `game outcome`. Note that your bot will still be graded under the full game, even if you only implement Part A.

5.1.1 Part A1 – Basic Functionality

20 points will be allotted for the following functionality:

- Implement your game state and conversion functions
- Handle initial moves
- Handle rolling dice and resource generation (do nothing on rolling `cROLL_ROBBER`)
- Handle ending the turn, which passes play on to the next player.

5.1.2 Part A2 – Full Moveset

35 points will be allotted for completing the functionality described in part A. Prompt for, handle, and override invalid moves for:

- Discard and robber moves due to `cROLL_ROBBER`
- Trade responses
- Maritime and domestic trades
- Buying builds
- Playing cards

5.1.3 Custom game type

You may wish to define your own type for representing the game state. You should make sure that **at least** all of the information represented in `state` is either directly represented or indirectly inferable from your `game` type. Your type may contain additional information that makes your calculations easier, but know that it will need to be derivable when converting from `state`.

You will also need to implement `game_of_state` and `state_of_game` to convert to and from your custom type and the `state` type in `definition.ml`.

You should be **VERY SURE** that your game type is comprehensive and that your conversion functions are correct. They will be called many times, and may manifest confusing, subtle, and gamebreaking bugs if they are incorrect.

> **Warning:** Do not use references or other side effects (besides printing) in any part of your `game` implementation or any modules that it depends on. Your game will not be graded correctly and it will be much harder for you to ensure correctness. However, you may find side effects useful in implementing your bot, and you may use them there.

5.1.4 handle_move

This function takes a state and a move and returns the updated state after that move has been completed. This is the primary function in the game implementation.

The `Play` module advances the game by repeatedly looking at the current state, finding the `next` color and request, and calling `handle_request` from the proper bot module for the next move. The `Game.handle_move` function will then be called to advance the game for every move.

Because the `Play` module makes no effort to check bot output, `handle_move` must check whether the move is valid, and override invalid moves. This is a crucial part of the game implementation. Invalid moves are either moves that do not satisfy the required request, are out of sequence or otherwise unexpected, or simply fail to satisfy relevant game rules. In the case of an invalid move, the game should substitute a "minimum viable move." If a specific request (`InitialRequest`, `RobberRequest`, `DiscardRequest`, `TradeRequest`) must be served, a random valid move is made instead. If an `Action` is required, choose the valid option between `RollDice` and `EndTurn`.

Finally, `print_update` is called to print an updated UI with the **updated state after the move is made**, **the move that was actually made**, and the color of the player who made the move.

> **Testing:** You may want to implement `print_test` in `Print.ml` to display additional stateful information for troubleshooting.

5.1.5 presentation

This function is called whenever the state is presented to a player bot. Since there are certain elements of imperfect information, we must hide anything that the recipient should not be able to see. We facilitate this functionality with the `cards` type. All instances of type `cards` in the state that the next player **does not** have access to should be converted to a `Hidden of int`, where `int` is the number of cards.

If you are using your custom `game` type, you must figure out how to handle this case and be able to convert between it and the `cards` in `state`. It is considered incorrect if something that should be hidden is revealed (or vice-versa) when converting between `state` and `game`.

Since this is only called from `Play`, no information is lost when it is hidden in `presentation`.

5.2 Part B – Trophies and Win Conditions

Part B includes all of the game logic required to implement the largest army and longest road trophies, as well as victory point calculations. If you have made reasonable design decisions, this should be a relatively modular extension to your existing code. Part B accounts for the remaining 10 of the 65 points allotted for Game.

5.2.1 Extending handle_move

Your `handle_move` should now update players' trophies for moves that can affect their possession. If a knight is played, a player's army count is incremented. If the player makes a move that earns them a trophy, it is transfered from the previous possessor (if one exists) to that player. Finding the largest army is straightforward, but you may use the `longest_road` function in `util.ml` to help you find the player with the longest road. Please see subsection 2.9 for more specific game rules regarding trophies.

Your `handle_move` should now output the correct outcome if a player has won the game. This should occur upon the move that grants the player enough victory points to satisfy the win condition.

6 Bot Implementation

The second part of your implementation is your player bot. Your bot should make decisions based on the game state and any other information you choose to keep track of. Each instance of a bot will play for a specific color, and its `handle_request` function will be called to request a move whenever the `next` request is for that color. You should think about various strategies and choose one that is logical and well-rounded.

6.1 name

Set `name` to the name of the file, without caps and without `.ml`. For example, the simple bot we have provided you with, `babybot.ml`, contains a declaration `let name = "babybot"`. This

allows us to reference bots by name.

6.2 initialize

If your bot uses any side effects, for example, to remember additional information not contained in `state`, then `initialize` resets those side effects for a new game. If you do not use side effects, then `initialize` does nothing.

6.3 handle_request

This function should look at `next` in state, which indicates the bot's color and the type of move requested and return the next move that the bot decides to make. This is the bot's primary function. Use any information and side effects you desire to make this decision.

Please keep in mind that constants in the game formulation, especially the board layout, are subject to change. Your bot should not rely primarily on hard-coded values to make decisions.

You may wish to generate random initial states to see how well your bot performs under different circumstances. To do this, simply replace `gen_initial_state` with `gen_random_initial_state` in `init_game` in `game.ml`.

> **Failing:** Your bot should not contain any failwith statements or raise exceptions. You should always be able to return a move. This ensures that the game will never fail because of your bot.

7 Source Code

The source code in the release includes:

shared/definition.ml	Game type definitions
shared/registry.ml	Keeps track of bots
shared/constant.ml	Game constants
shared/util.ml	Useful helper functions
shared/print.ml	Print suite for gui and debugging
bots.ml	List of available bots
babybot.ml	Very basic sample bot
game.ml	Stub file for handling moves
game.mli	Interface for game
play.ml	Framework for playing game(s)
main.ml	Main function which interprets command line input

You are free (and encouraged) to add new modules to organize your `game` implementation. You should also implement your bot in the same folder. Add any bots you write to `bots.ml`. **Do not modify or add files to the `shared` folder**. Do not modify `game.mli`.

8 Running the Game

To run the game, use the 3110 tools to compile and run `main.ml`. Main requires the names of the four bots that will compete.

```
cs3110 compile main.ml
cs3110 run main.ml bot1 bot2 bot3 bot4
```

Now you're playing. You can add optional parameters for additional functionality.

```
cs3110 run main.ml bot1 bot2 bot3 bot4 [n] [disp]
```

Where `n` is the number of games you want to play, and `disp = sleep | nosleep | step`. In `sleep` mode, the game will automatically tick forward every `cSTEP_TIME` seconds. This is the default display option. In `nosleep` mode, the print module will be suppressed, and only game results will be shown. This is useful when you want to play through many games very quickly and aggregate the results. In `step` mode, press **Enter** to step through the game, enter **s** to print out a complete code representation of the state, and enter **q** to quit the game.

You may find it particularly useful to play with `n = 1` and `disp = step` when you're debugging your game. You can also use the keyboard shortcuts **Shift+PageUp** and **Shift+PageDown** to step backwards and forwards through the portion of the game already played.

We recommend that you increase the font size in your terminal to make the display clearer. We **highly recommend** that you vastly increase the **scrollback lines** parameter in your terminal, so that you can step further back in time.

9 Your Tasks

Here is a summary of what this project entails:

- **Design Review:** Present your plans and design decisions. Sign up for a review slot on CMS
- **Game:** Implement the `game` type, `game_of_state`, `state_of_game`, `handle_move`, and `presentation` in `game.ml`
- **Bot:** Write a bot to play the game intelligently. We will grade your bot based on its performance against staff bots.
- **Tournament:** Optionally, Submit your champion bot to the CS3110 tournament. This slot will remain open on CMS past the project due date.

10 Guidelines

This is a large and open-ended project. Make sure you spend time thinking about and designing each part before writing code. Please manage your time well.

Here are some things that you should think about while working on this project.

10.1 Preliminary Guidelines

Your design is important. This project is quite large and complex. If you do not have a solid design, you will quickly get bogged down in details when you begin implementing. Before writing any code, you should have a very clear idea of **all** of the following.

- How you will represent your custom game type, and all of the information that needs to be stored to fully represent the game state.
- How to store and access that information efficiently.
- What the interfaces of your modules will be.
- What invariants each of your modules will preserve.

Think about how to organize your program into loosely coupled modules. It will be difficult to debug your project unless you can develop modules that encapsulate important aspects of the game. Design your modules carefully so that you can work effectively with your partner and do unit testing of each module.

Implement and test gradually and systematically. Start with simpler functionality, and once you are sure those are correct, move on to the more complex ones.

10.2 Design Review Guidelines

Design reviews will take place during the week of April 28th. You will be responsible for a 5–7 minute presentation on your design and implementation so far. As reviews will take place quite late in the project timeframe, you will be expected to have a substantial understanding of the project and to have made **some** noticeable progress. We will not be as lenient as we have been with design reviews in the past. Do not expect to simply read the writeup and earn full credit. Please be prepared with some or all of the following:

- A basic understanding of how the game works, the structure of the provided files, and what is expected of you.
- A brief design document that explains your design decisions, modules, and plan of attack.
- An umprompted verbal presentation of your thought process, design decisions, and progress so far.
- The ability to justify your choices and answer questions.
- Some progress in the form of written code.
- Questions about problems that you're currently facing.

We expect you to think about your presentation, but don't worry too much about being rehearsed and polished. We want to see that you have put thought into the project and that you have made quantifiable progress.

11 Submission

You will submit:

- A `zip` of all files in your `ps6` directory, including those that you did not edit. Make sure to preserve the directory structure of the original release. We should be able to unzip your submission and build the game **without errors or warnings**. **Submissions that do not meet this criterion will be penalized.** Your final submission should include only one bot.
- Your design document in `pdf` format.

There will be a second assignment open on CMS for bot submissions. This slot will be open until the day of the tournament. Your tournament bot should be self-contained. It should not depend on any other modules besides those in `shared/`. You should add any necessary dependencies to the body of your tournament bot module.

Good luck, and **start early**!