Team Nora— Carcassonne

Christina C., Nora X., Chelsea Y., Yitian Z.

Table of Contents

Program Outline

Runner
Description: The Runner class contains the main method for running the program and instantiates the frame.

Frame
Description: The Frame class instantiates the size of the board and creates JFrame.

Panel
Description: The Panel class creates JPanel for displaying our graphics. It reads in tile images, paints GUI elements (scoreboard, instructions, board, etc), and displays the end scores once the game is finished. It also implements KeyListener and relays information to the GameState class.

Player
Description: The Player class instantiates the number of meeples and assigns them by ID numbers.

GameState
Description:

Planning

Task Assignment

Prospectus

Task	Details	Assigned Teammate
GUI	Create “skeletons” of each GUI used in the game as well as examples of what in-game play would look like. Include explanations of different components of each GUI.	Nora
UML	Create a mapping of each class in the program, their attributes, and their methods. Include connections to show how each class is	Nora

	associated with another in a neat and readable format.	
Algorithm	Write detailed explanations of various methods throughout the program such as checking whether a tile is legal and merging feature systems as well as scoring algorithms for each feature.	Yitian
Program Outline	Write summaries of each class' purpose within the program as well as how each class interacts with one another.	Yitian
Test Data	Create possible test cases in order to test boundary/special cases and ensure that all possible quirks are discovered.	Christina
Timeline	Write detailed explanations of tasks assigned to each team member (how meta). Create a rough outline of dates where certain milestones must be completed.	Chelsea

Programming

Classes	**Assigned Teammate**
Runner, Frame, Panel, Player	Nora
GameState, Board	Yitian
Cropping tile images, TileData, Tile class	Christina
Features, FeatureSystems	Chelsea

Timeline

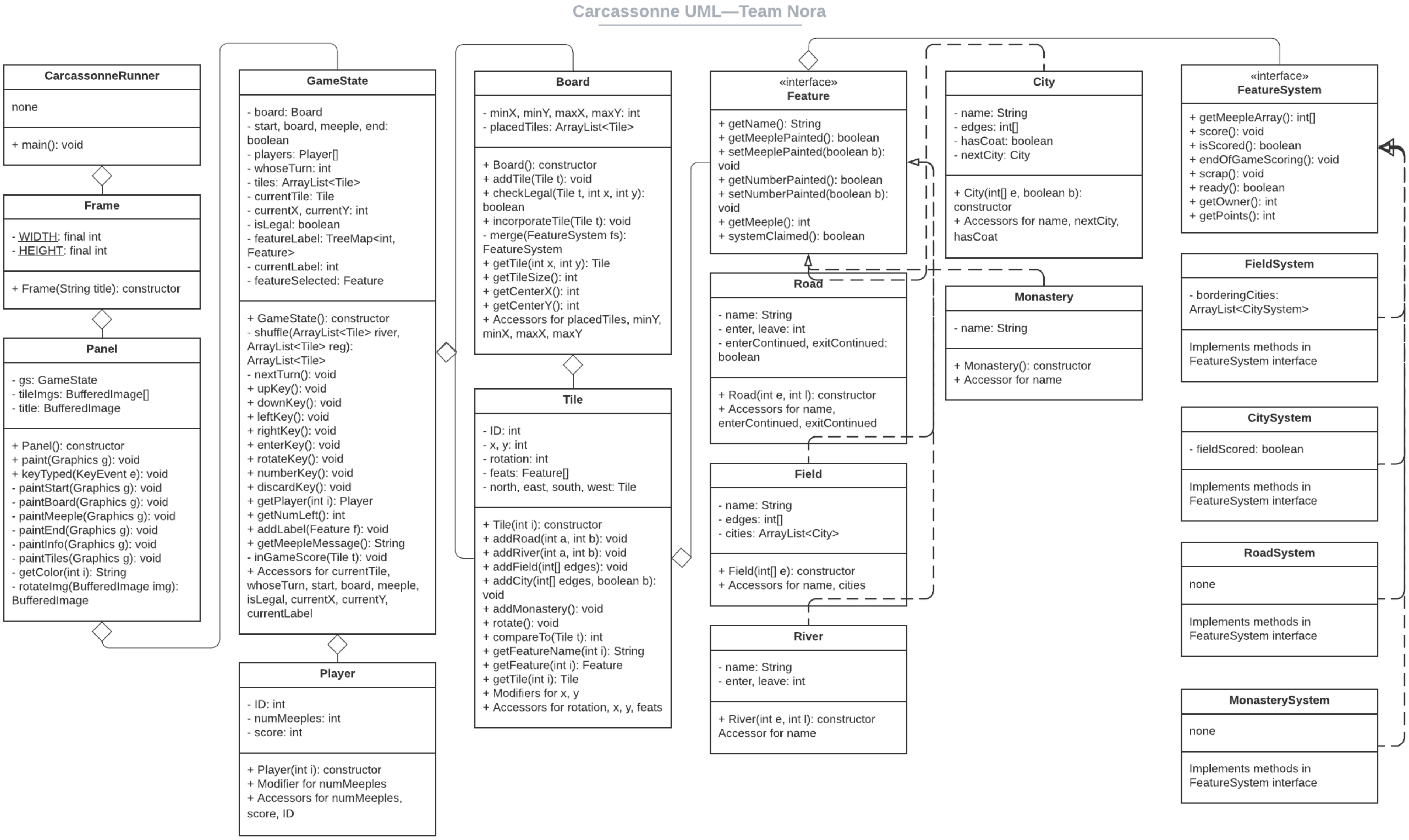
February 25	**March 1 - March 5**	**March 8 - March 12**	**March 15 - March 19**	**March 22 - March 25**
Assign roles to team members. Team discussion: basic class outlines, methods, algorithms, etc	Team discussion continues: more detailed methods and classes	Team discussion continues: going through test cases and logic issues.	Spring break. Team members create rough drafts of assigned prospectus tasks.	Team discussion continues: work through scoring and last few methods Finalize presentation and prospectus.
March 26	**March 29 - April 2**	**April 5 - April 9**	**April 12 - April 16**	**April 19 - April 23**
Submit prospectus and add final touches to slideshow presentation.	Presentations and peer evaluations.	Teammates program according to classes assigned. Program outline	Testing and debugging with focus on ensuring the GUI displays properly. Ensure that each tile image is printed properly and that each tiles	Testing and debugging with focus on scoring. Ensure that features are being scored

		with attributes and methods listed is used to help the programming process.	corresponds to the correct tile data. GUI elements should be proportional and be in the correct location. Ensure that tile rotations work appropriately, tiles are shown to be illegal when in inappropriate location, meeples are painted in correct location, etc.	correctly and at the correct instances.
April 26 - April 30	**May 3 - May 6**	**May 7**		
Further testing and debugging— we will be testing very special boundary cases and attempting to create unique test cases as to discover possible fixes. If time permits, additional features such as AI will be implemented.	Final touches on the project as well as additional features, including those mentioned in the “Further Considerations” section.	Project Due!		

Milestones

Date	**Milestone**
March 26	Submit prospectus
April 5	Begin programming
April 9	Finish programming
April 12	Begin testing and debugging
April 26	Finalize program and begin implementing additional features
May 7	Submit project!

UML



Carcassonne UML—Team Nora

CarcassonneRunner

none

+ main(): void

Frame

- WIDTH: final int
- HEIGHT: final int

+ Frame(String title): constructor

Panel

- gs: GameState
- tileImgs: BufferedImage[]
- title: BufferedImage

+ Panel(): constructor
+ paint(Graphics g): void
+ keyTyped(KeyEvent e): void
- paintStart(Graphics g): void
- paintBoard(Graphics g): void
- paintMeeple(Graphics g): void
- paintEnd(Graphics g): void
- paintInfo(Graphics g): void
- paintTiles(Graphics g): void
- getColor(int i): String
- rotateImg(BufferedImage img): BufferedImage

GameState

- board: Board
- start, board, meeple, end: boolean
- players: Player[]
- whoseTurn: int
- tiles: ArrayList<Tile>
- currentTile: Tile
- currentX, currentY: int
- isLegal: boolean
- featureLabel: TreeMap<int, Feature>
- currentLabel: int
- featureSelected: Feature

+ GameState(): constructor
- shuffle(ArrayList<Tile> river, ArrayList<Tile> reg): ArrayList<Tile>
- nextTurn(): void
+ upKey(): void
+ downKey(): void
+ leftKey(): void
+ rightKey(): void
+ enterKey(): void
+ rotateKey(): void
+ numberKey(): void
+ discardKey(): void
+ getPlayer(int i): Player
+ getNumLeft(): int
+ addLabel(Feature f): void
+ getMeepleMessage(): String
- inGameScore(Tile t): void
+ Accessors for currentTile, whoseTurn, start, board, meeple, isLegal, currentX, currentY, currentLabel

Player

- ID: int
- numMeeples: int
- score: int

+ Player(int i): constructor
+ Modifier for numMeeples
+ Accessors for numMeeples, score, ID

Board

- minX, minY, maxX, maxY: int
- placedTiles: ArrayList<Tile>

+ Board(): constructor
+ addTile(Tile t): void
+ checkLegal(Tile t, int x, int y): boolean
+ incorporateTile(Tile t): void
- merge(FeatureSystem fs): FeatureSystem
+ getTile(int x, int y): Tile
+ getTileSize(): int
+ getCenterX(): int
+ getCenterY(): int
+ Accessors for placedTiles, minY, minX, maxX, maxY

Tile

- ID: int
- x, y: int
- rotation: int
- feats: Feature[]
- north, east, south, west: Tile

+ Tile(int i): constructor
+ addRoad(int a, int b): void
+ addRiver(int a, int b): void
+ addField(int[] edges): void
+ addCity(int[] edges, boolean b): void
+ addMonastery(): void
+ rotate(): void
+ compareTo(Tile t): int
+ getFeatureName(int i): String
+ getFeature(int i): Feature
+ getTile(int i): Tile
+ Modifiers for x, y
+ Accessors for rotation, x, y, feats

«interface»
Feature

+ getName(): String
+ getMeeplePainted(): boolean
+ setMeeplePainted(boolean b): void
+ getNumberPainted(): boolean
+ setNumberPainted(boolean b): void
+ getMeeple(): int
+ systemClaimed(): boolean

Road

- name: String
- enter, leave: int
- enterContinued, exitContinued: boolean

+ Road(int e, int l): constructor
+ Accessors for name, enterContinued, exitContinued

Field

- name: String
- edges: int[]
- cities: ArrayList<City>

+ Field(int[] e): constructor
+ Accessors for name, cities

River

- name: String
- enter, leave: int

+ River(int e, int l): constructor
Accessor for name

City

- name: String
- edges: int[]
- hasCoat: boolean
- nextCity: City

+ City(int[] e, boolean b): constructor
+ Accessors for name, nextCity, hasCoat

Monastery

- name: String

+ Monastery(): constructor
+ Accessor for name

«interface»
FeatureSystem

+ getMeepleArray(): int[]
+ score(): void
+ isScored(): boolean
+ endOfGameScoring(): void
+ scrap(): void
+ ready(): boolean
+ getOwner(): int
+ getPoints(): int

FieldSystem

- borderingCities: ArrayList<CitySystem>

Implements methods in FeatureSystem interface

CitySystem

- fieldScored: boolean

Implements methods in FeatureSystem interface

RoadSystem

none

Implements methods in FeatureSystem interface

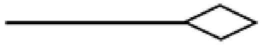
MonasterySystem

none

Implements methods in FeatureSystem interface

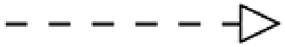
Key

Association



The class this arrow points to contains an instance of the class. For example, class GameState contains instances of the Player class Implementation

Implementation

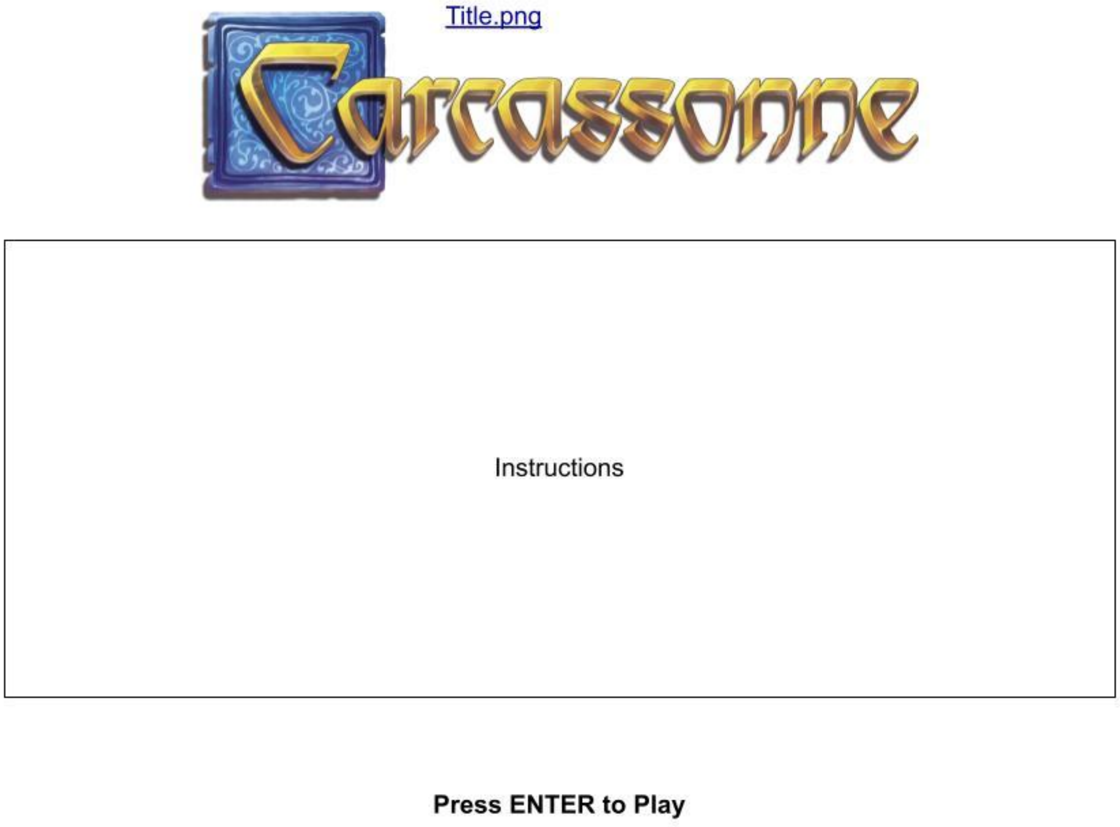


The class implements the interface it points to. For example, class Field implements interface Feature.

Graphic User Interface (GUI)

Frame aspect ratio 4:3. Uses keys ONLY—no mouse.

Start Page



Title.png

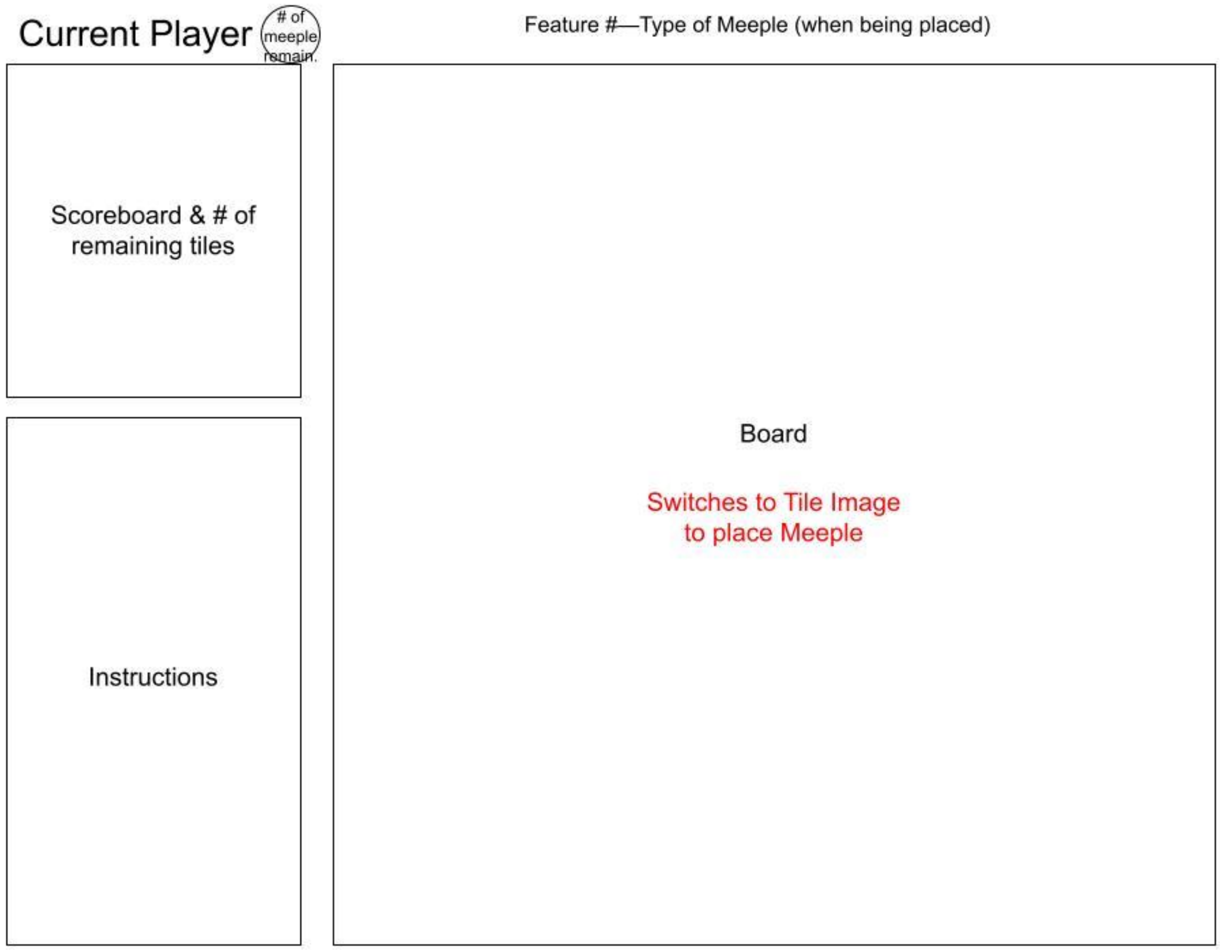
Instructions

Press ENTER to Play

Included Instructions

Circle = Meeple, Enter to place Meeple/Tile, R to rotate, arrow keys to move, backspace to discard Meeple

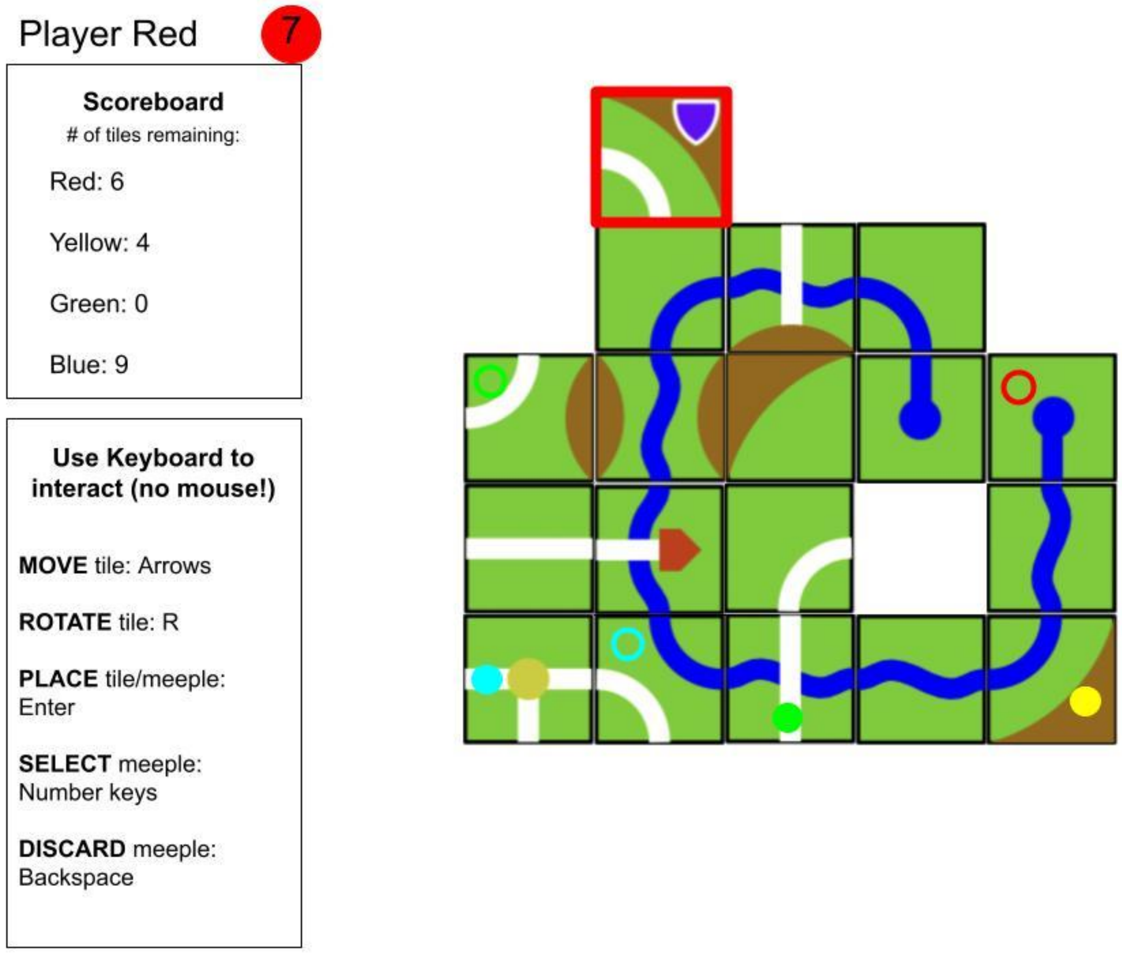
Game Page


Current Player
# of meeple remain.
Feature #—Type of Meeple (when being placed)
Scoreboard & # of remaining tiles
Instructions
Board
Switches to Tile Image to place Meeple

Blown-up tile image last-placed is shown to allow player to place Meeple each turn; otherwise, board is shown.

Keys

- Up, down, left, right arrow keys to move tile location.
- R key to rotate tile
- Enter key to place tile/Meeple
- Backspace key to discard Meeple
- 1-8 keys to place Meeple



Player Red 7

Scoreboard

# of tiles remaining:

Red: 6

Yellow: 4

Green: 0

Blue: 9

Use Keyboard to interact (no mouse!)

MOVE tile: Arrows

ROTATE tile: R

PLACE tile/meeple: Enter

SELECT meeple: Number keys

DISCARD meeple: Backspace

Other Meeples are filled in circles, field Meeples are unfilled circles.

●	○
Other	Farmer

Players

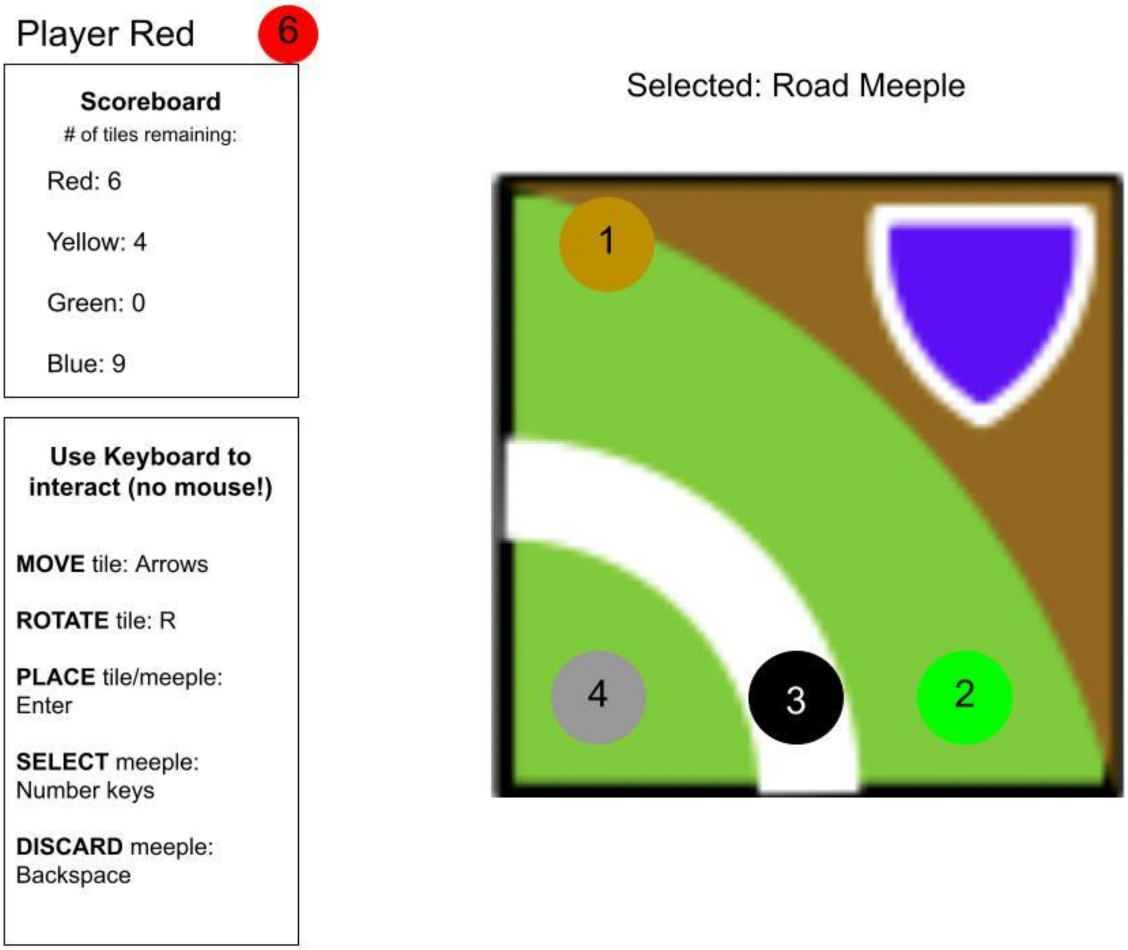
- Always referred to by **color** not ID/number.
- Player 0 = red, 1 = yellow, 2 = green, 3 = blue.

Board

- Tile sizes are incremented as width of board grows.
- Board shifts to stay centered.

Placing tiles

- Tile is automatically placed at (0,0) and can then be moved around.
- Tile has **green** border when in legal location/rotation and **red** border when not.
- Tiles are rotated 90 degrees *counterclockwise* each time.



Player Red 6

Scoreboard
# of tiles remaining:

Red: 6

Yellow: 4

Green: 0

Blue: 9

Use Keyboard to interact (no mouse!)

MOVE tile: Arrows

ROTATE tile: R

PLACE tile/meeple: Enter

SELECT meeple: Number keys

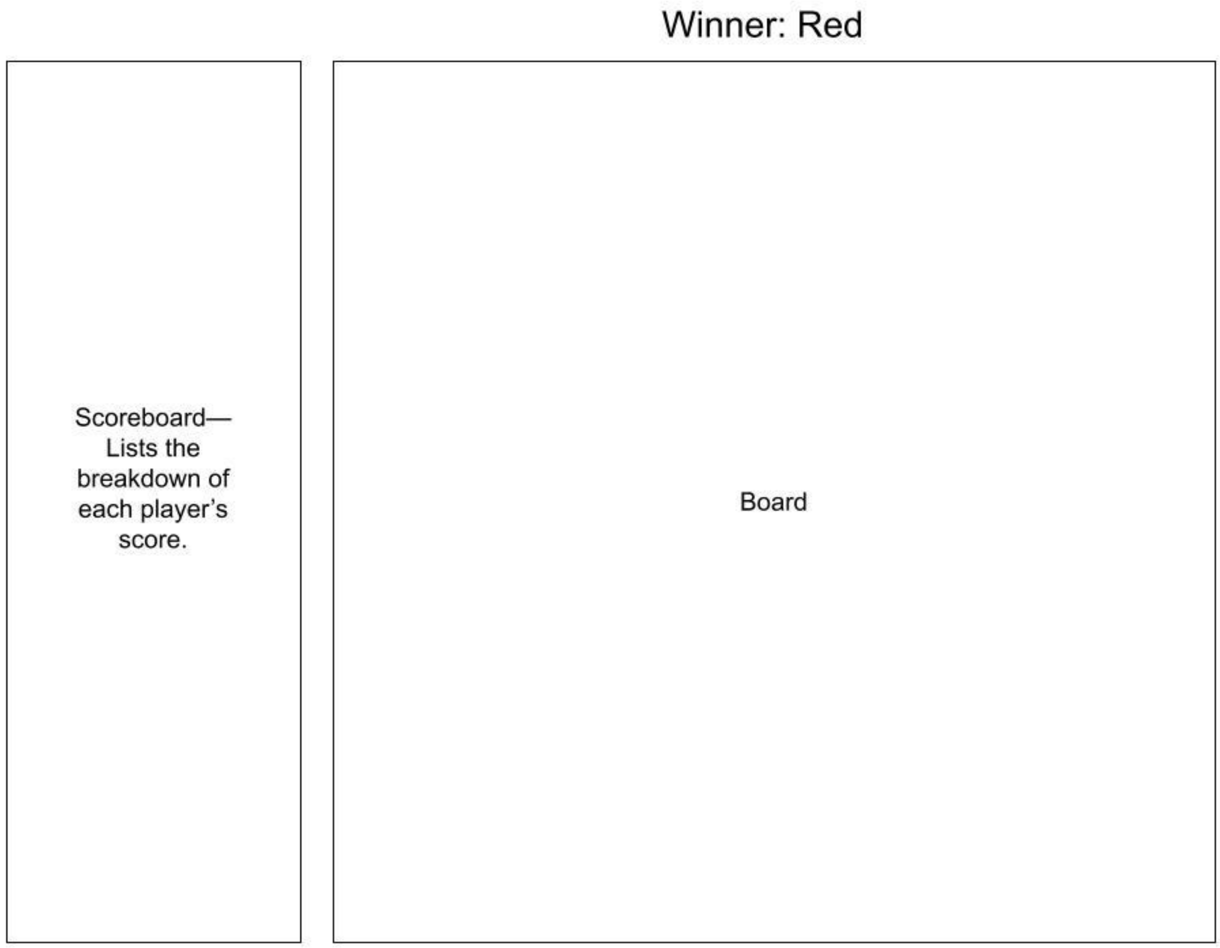
DISCARD meeple: Backspace

Selected: Road Meeple

Placing meeples

- Features are numbered and player types number in to place meeple
  - If a feature is already claimed, the circle the number is in is grayed out. If not, circle is colored according to the feature it represents (city: brown, field: green, monastery: red, road: white)
  - When a feature is selected, the circle is colored black.
- Screen is automatically shown after tile is placed.
- Based on what number key the player presses, type of meeple is displayed for validation.
- Number keys to select location -> Check type of meeple is correct -> Enter key
- If no number is selected and enter key is pressed, then a meeple is not placed

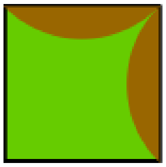
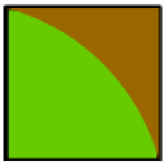
End Page


Winner: Red
Scoreboard—Lists the breakdown of each player's score.
Board

Test Data

Possible Test Cases

- feature where meeple already placed can't have another meeple on it: make boolean checking if that feature already is claimed
    - Check scoring at end for meeples that share the same feature (for meeples that were placed before features were fully complete/ connected)
- feature that checks if a tile fits in a specific location (with different types of tiles such as roads in mid, tiles w/o roads, dif rotations)
- test case where multiple tiles are placed onto the board and we run a "test game" and record what feature systems were completed, rotations, how each algorithm fits in, etc (example)
    - 1. The starting tile will appear in the middle of the screen and the program will randomly choose
    - another river tile for the next player.
    - 2. The player can choose rotate the tile by clicking "R" and can use arrow keys to choose the
    - location. If the edges match the tile will be placed when clicking the Enter key.
    - 3. When this tile placed the river is completed and the program should have a river score for this
    - player. And the meeple should be returned back to this player.
    - 4. When this tile placed the road is completed and the program should have a road score for this
    - player. Also the meeple should be returned back to this player.
    - 5. After this tile placed the city is completed and program should have a city score this this player
    - and return the meeple back to this player.
- checking city connection between tiles
- checking number of cities on a tile:

 can be part of 2 or 1 cities  part of 1 city

- using references to check whether city is finished

Testing the logic of the game will consist of:

- Testing all methods in each class
- Testing how all the methods will work together
- Test the classes working together in the methods
- Play the game not as intended to try to find bugs and issues

Examples of Logic Tests and Scenarios we must test include:

- Ensuring the directions of the game are understandable to the player and run smoothly
    - Everyone knows how to play the game and how it is supposed to be played
- Which tile can make connections with other tiles and with meeples

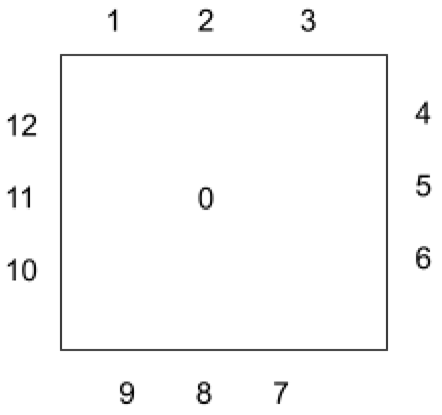
- Testing the scoring system of counting the complete and incomplete types and the last field.
- Play a full round with all numbers of players to ensure that all scenarios work
    - 2, 3, 4, 5 players total

Testing Graphics

- Testing the graphics of the game will consist of:
- Playing the game not as intended to try to find bugs and issues
    - Clicking randomly and finding ways the game could softlock or break apart
- Getting others to test the game to see their reactions and how they played to determine
- whether the setup is comfortable and for other graphic related issues
- Checking that all graphics are where they should be and function with the logic section to
- move the images and text in accordance with the rest of the game
- Ensuring the setup piece is in the right place and ready to be played
    - Each of the players and their tiles
    - Assignment of colors and player turns
    - tile used during the game and if no use discard it

- Making sure that the tile can be attached to the set up tile
    - For players to connect segment in an attempt to gain more points
- Checking that the discarding of tiles switches the tiles and players afterward
- Ensuring that adding the tiles to the table correctly relate to the player and points
    - Clicking and Receiving where the piece is intended to be played
- Make sure the scoring points and how to end the game are shown correctly
    - End scoreboard and end of the game.

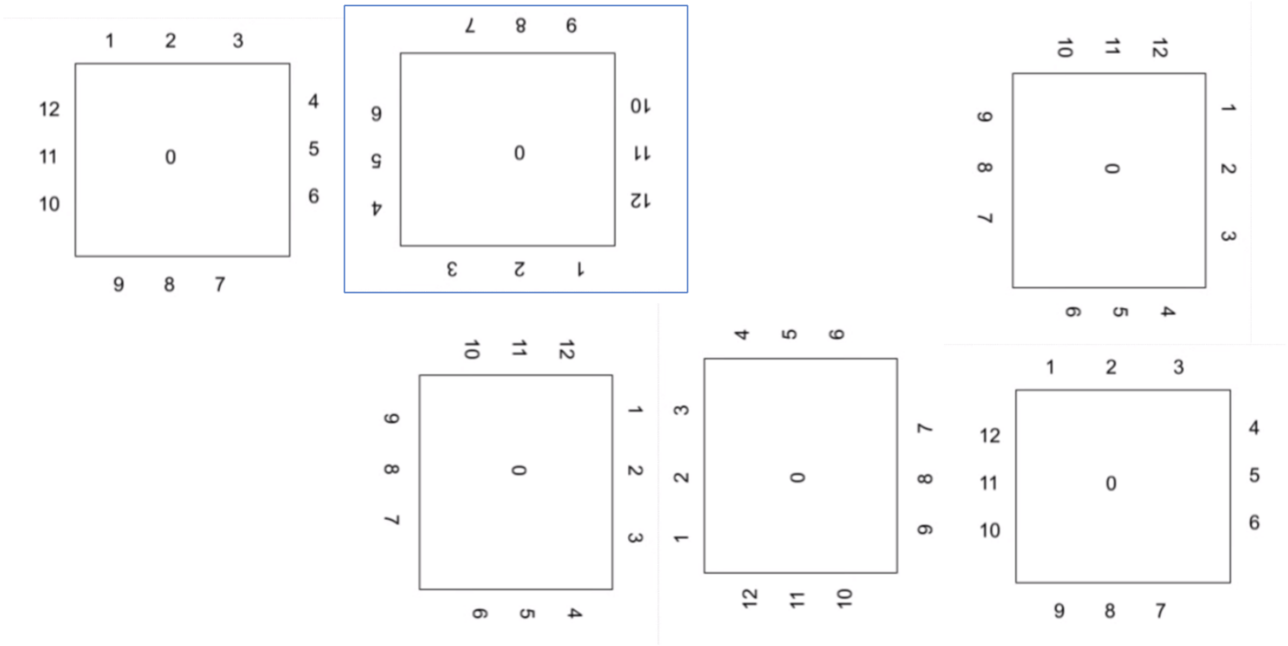
Algorithm



For the purposes of this program, each tile will be divided into 13 sections as shown above. This makes tasks such as comparing tiles and identifying feature locations easier.

Whether a tile fits on the board: checkLegal(Tile t, int x, int y) method within Board

- Note that smallest always corresponds to largest of the other tile
- Largest to smallest of the other tile
- Medium to medium
- Mathematical formula looking at the rotation to determine which sides and which numbers we are looking at



- **Step 1:**
  - Return false in case currentTile is null

  - Return false if a tile already exists in the desired spot. Check this by doing getTile(x,y). If a tile already exists we will return false
- **Step 2:**
  - Find the four bordering tiles. Temporarily place in BorderingTiles array of size 4.
  - North: getTile(x, y+1), East: getTile(x+1, y), South: getTile(x, y-1), getTile(x-1,y),
- **Step 3:** for loop going from 0 to 3. checkSide(#, currentTile, borderingTiles[#]), where # is the number in the loop. If something comes up “false”, return false immediately
- **Step 4:** by the time you get to this step, just return true.

getTile method: private helper method in Board

- So far, will go through placed tiles list and find the correct tile that contains the x and y coordinates.
- Planning to have tiles sorted by x and y coordinates to make our search log(n) instead of n

checkSide(int a, Tile currentTile, Tile borderingTile) method: private helper method in Board. checkSide returns a boolean, where true if the side matches, false if at least one thing is off.

- If the BorderingTile is null, return true.
- Int a is which side we are checking *on currentTile*. 0=north, 1=east, 2=south, 3=west
- We will get the numbers for the current tiles by doing this:
  - aSmall = ((a+currentTile.rotation)%4)*3 + 1 for smallest value
  - aMed = ((a+currentTile.rotation)%4)*3 + 2 for middle value
  - aLarge = ((a+currentTile.rotation)%4)*3 + 3 for largest value
- Int b is which side we are checking *against on borderingTile.* B is (a+2)%4
- We will get the numbers for the bordering tile by doing this:
  - bSmall = ((b+borderingTile.rotation)%4)*3 + 1 for smallest value
  - bMed = ((b+borderingTile.rotation)%4)*3 + 2 for medium value
  - bLarge = ((b+borderingTile.rotation)%4)*3 + 3 for largest value
- Then, we check matches.

  First check:
  - String str1 = currentTile.getFeature(aSmall).getName();
  - String str2 = borderingTile.getFeature(bLarge).getName()

  If str1 does not equal str2, return false immediately

  Second check:
  - Str1 = currentTile.getFeature(aMed).getName();
  - Str2 = borderingTile.getFeature(bMed).getName();

  If str1 does not equal str2, return false immediately

  Third check:
  - Str1 = currentTile.getFeature(aMed).getName();
  - Str2 = borderingTile.getFeature(bMed).getName();

  If str1 does not equal str2, return false immediately
  Finally return true.

PlaceTile(Tile t) //this method will exist in the Board

-
    - North = (getTile(x, y+1)). current.setNorth(North), North.setSouth(current)
    - South = (getTile(x, y-1)). current.setSouth(south), South.setNorth(current)
    - East = (getTile(x+1, y)). current.setEast(east), East.setWest(current)
    - t.setWest(getTile(x-1, y))
- Attach each Feature to a respective FeatueSystem
- Road - junction holds TRUE on both sides of a road connection

incorporateTile method in Board: Merges FeatureSystems appropriately according to circumstance.

public void incorporateTile(Tile t)

Case 1:

- FeatureSystem exists next to Feature.
- Feature is added to array of Features that FeatureSystem contains.
- The feature's mySystem attribute points to the FeatureSystem it's now a part of.

Case 2:

- No Tile adjacent to the side of t.
- Feature's mySystem attribute points to newly instantiated FeatureSystem, which contains only the Feature

Case 3:

- Feature is adjacent to multiple, distinct FeatureSystems
- Feature is added to array of Features that the FeatureSystem returned from calling merge() contains.
- The feature's mySystem attribute points to the FeatureSystem it's now a part of.

private FeatureSystem merge(FeatureSystem[] fs)

- Keeps first FeatureSystem in array, combines rest of the FeatureSystems into this first FeatureSystem.
- Traverses through every ArrayList in the "rest" of the FeatureSystems (all except the System located @ index 0), sets each Feature to point to the FeatureSystem at index 0. Adds all Features in every ArrayList to the ArrayList in the FeatureSystem @ index 0.
- Returns fs[0]

Scoring Algorithms

Fields

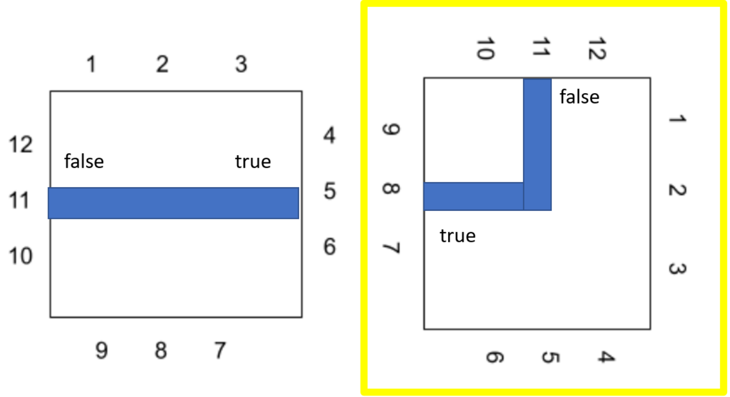
- Traverses through all fields contained, checks each field for the Cities that it contains. If the CitySystem that each City references ready() is true and fieldScored is false, fieldScored is set to true and adds the CitySystem to **borderingCities**.
- Points are awarded according to the size of **borderingCities**.
- After points are awarded to Player(s), **borderingCities** is traversed and fieldScored is set to false for each CitySystem.

Cities

- Every time a tile with city(s) on it is placed, each city Feature(s) will point to the city Feature directly adjacent to it on the tile next to it. If there is not a tile placed next to the city, the Feature points to null.
- A citySystem is finished when every city it contains does *not* point to a null.

Roads

- Each end of the road contains a boolean, true if there is a road adjacent to it or it stops, false if no tile is adjacent. So, each Road contains two booleans.
- A RoadSystem if finished if, for every Road, both booleans are true.

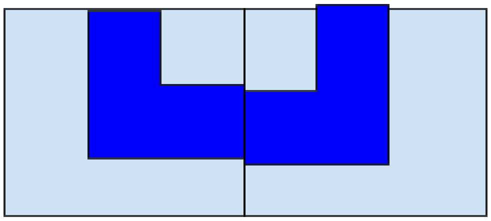


Monasteries

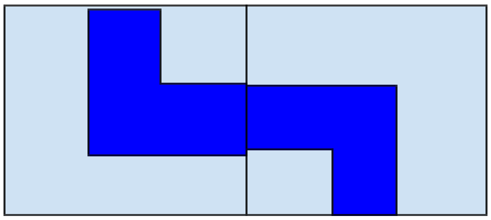
- Check north, south, east west. If they are all not null, continue, if one of them is null, return false immediately
- Check east of north, west of north, east of south, west of south. If one is null, return false immediately.
- Return true.

Rivers

Not allowed (case X)



Allowed (case y)

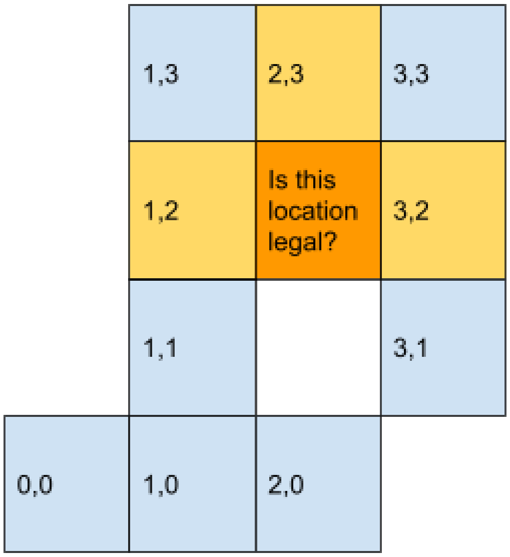


Assume the left tile was placed before the right tile.
If the river is a turn tile, the RiverSystem class will store the direction where the turn came from. In these two examples, this would be the north. If the next tile that occurs is also a turn tile, the tile must not be placed to point the same direction as where the previous one came from. For example, in the images above, the second tile cannot have the river exit north. It must exit south.

Further Considerations

Traversing through the board

As of now, using an ArrayList of tiles placed on the board and traversing through the entire List in order to find the Tile needed will suffice. However, a consideration to improve upon the O(N) time is too sort the tiles in the List according to their x and y coordinates.



	1,3	2,3	3,3
	1,2	Is this location legal?	3,2
	1,1		3,1
0,0	1,0	2,0	