Scroll Down For Report

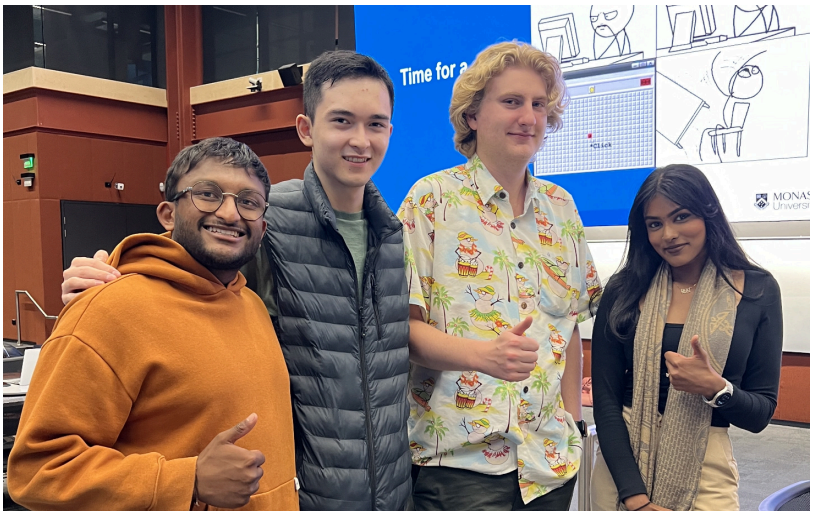
MONASH University

FIT3077 Sprint 2

By Team NOSS(115)
Nick Ballard (33861536)
Omila Herath (31471439)
Sam Morgan (32525648)
Sona Hariharan (33221898)

1. Team Introduction:

Team NOSS:



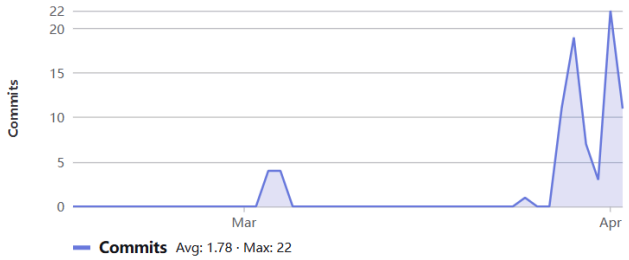
From left to right: Omila Herath, Nick Ballard, Sam Morgan, Sona Hariharan

Team Member	Technical and Professional Strengths	Fun Fact
Nick Ballard	- Proficient in python and java - Experience developing desktop, web and mobile applications - Previously used the Agile development methodology to manage projects	I am a dual citizen (AUS/USA)
Omila Herath	- Proficient in python, java and some pyTorch - Extensive experience working in Uni based team tasks and Projects - Personal Experience in creating Reinforcement Learning based projects and interested in AI	I'm training for a Half Marathon
Samuel Morgan (Lead)	- Experienced programmer, with a depth of experience working in OOP. - Completed various placements working amongst different professional teams, as such has experience in cooperation, communication and planning. - Personal experience working in Game Development, and the multidisciplinary concepts that require	I own a pet turtle named Squirt
Sona Hariharan	- Programming experience in python and java - Experience in code testing (unittest and CI/CD) - Experience developing an application while following Agile workflow as a team	I have a sister ~16 years younger than me!

2. Contributor Analytics:

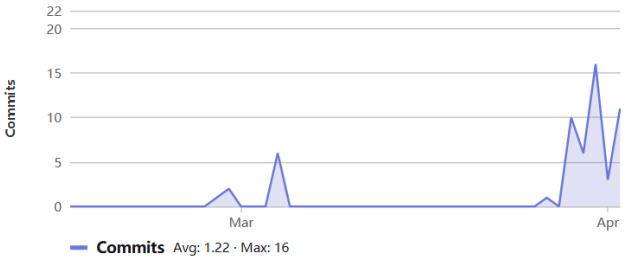
oher0004

82 commits (oher0004@student.monash.edu)



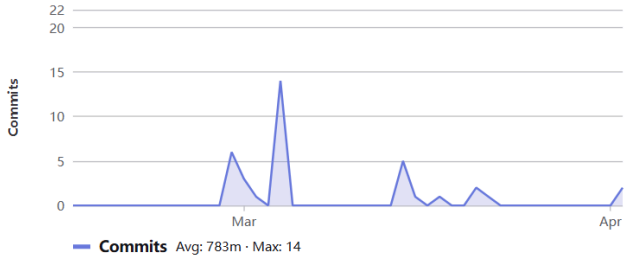
NicholasBallardDev

56 commits (nbal0023@student.monash.edu)



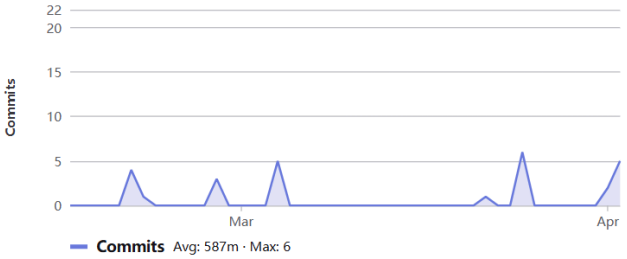
smor0055

36 commits (smor0055@student.monash.edu)

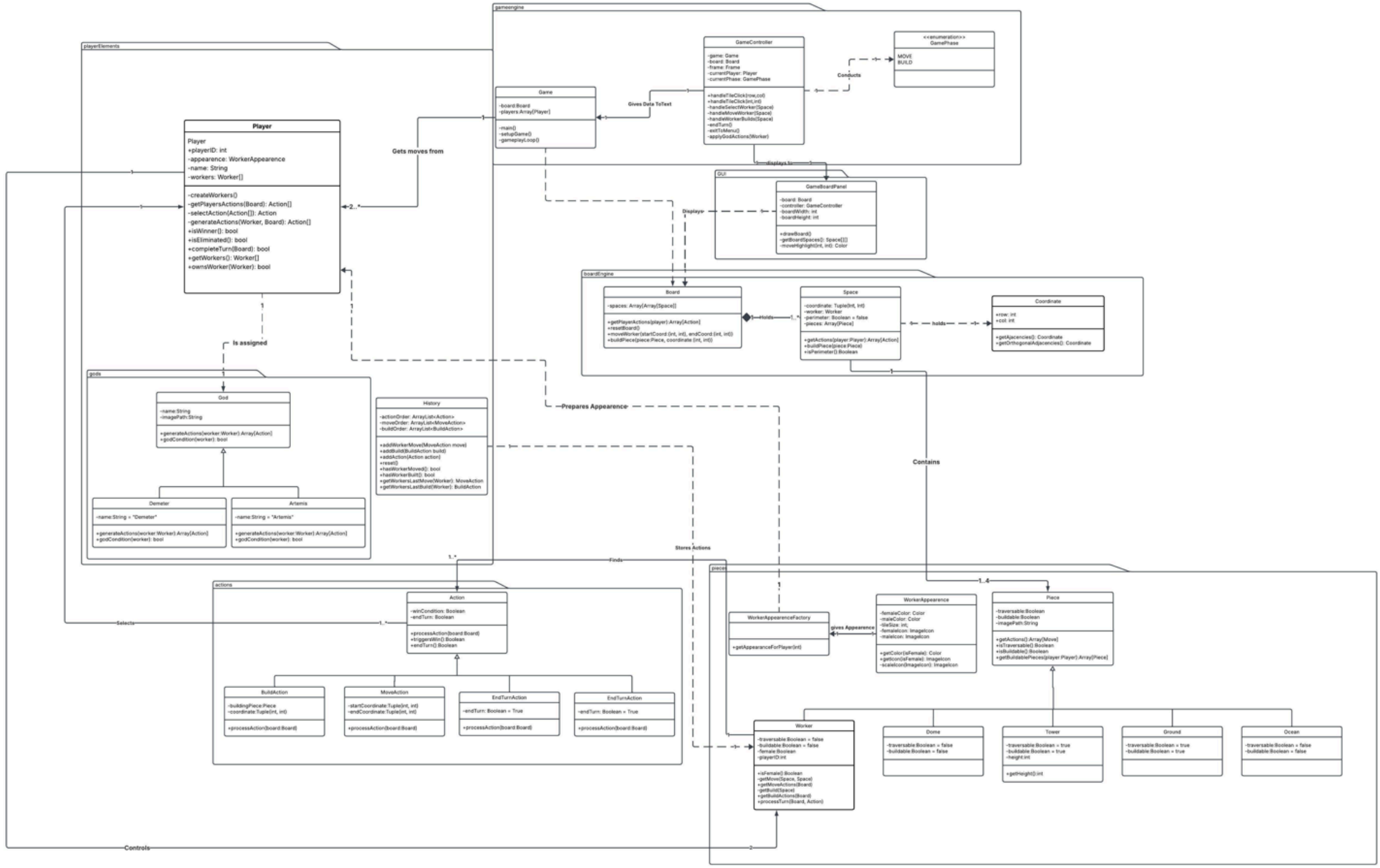


b-onbon

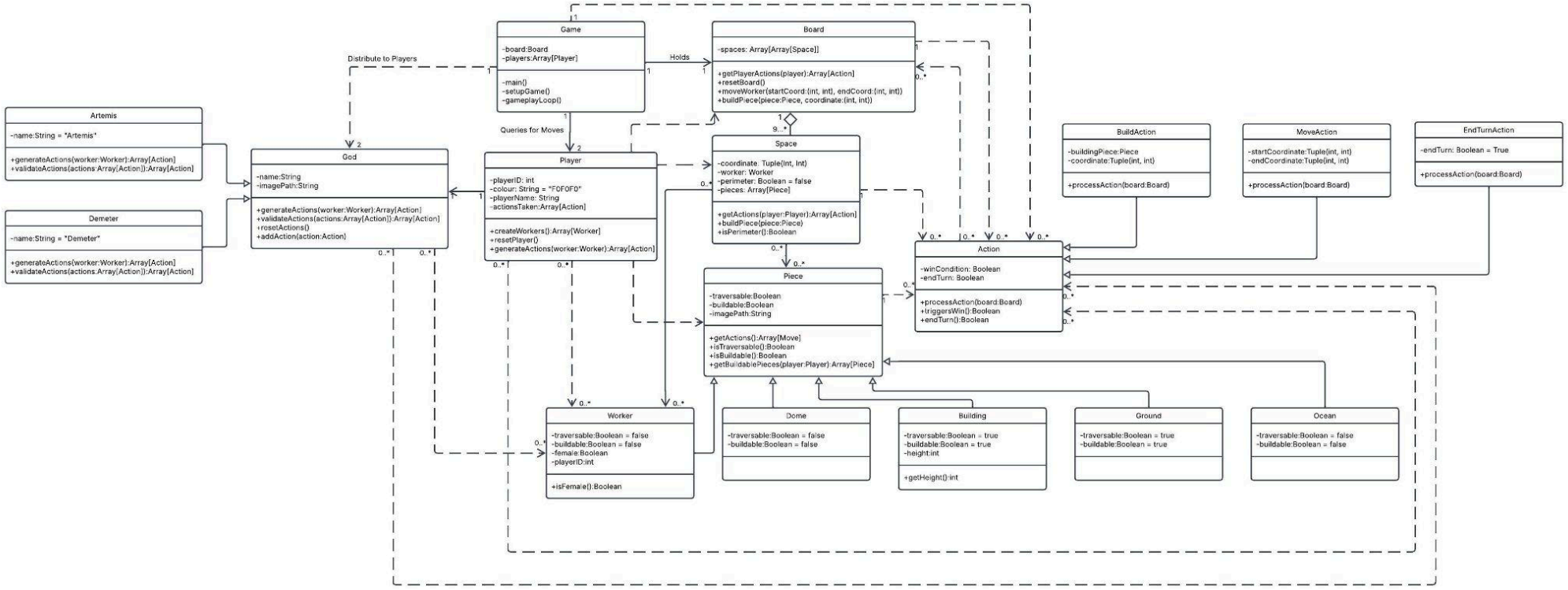
27 commits (shar0108@student.monash.edu)



Class Diagram: Lucid Chart Link



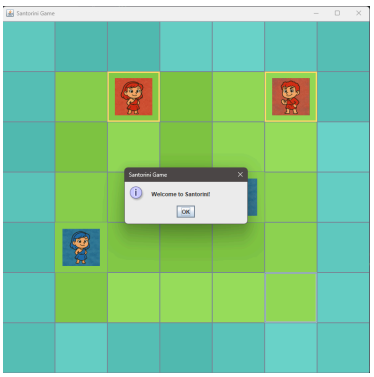
Class Diagram Drafts



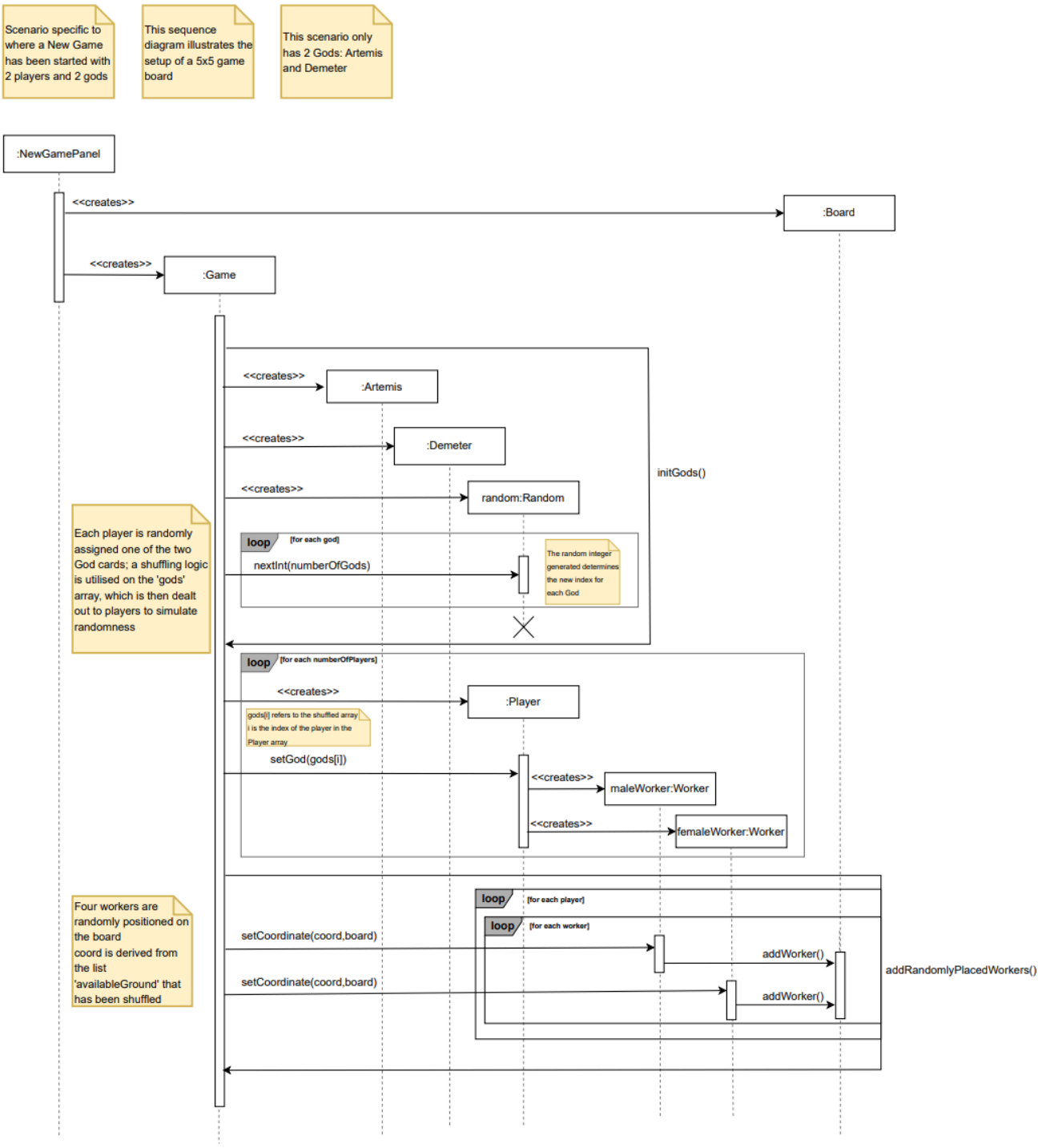
Game Functionality and Sequence Diagrams:

Game Setup:

GUI Output

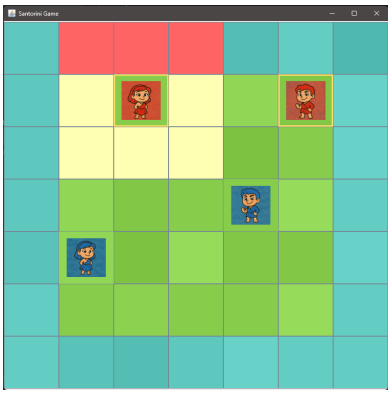
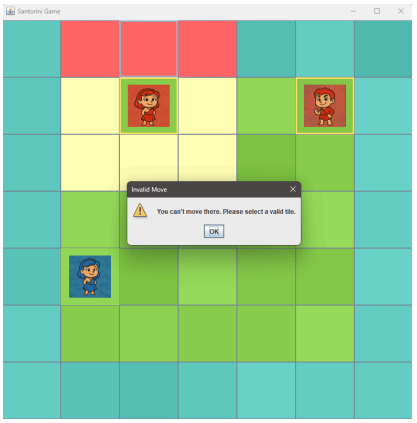
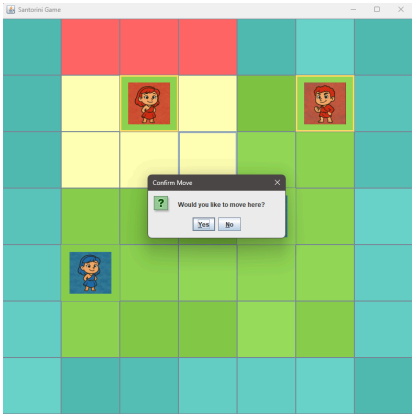
Action	Output
A 5x5 board is generated upon game start. Workers are assigned to players and randomly placed on the board. Highlighting present around the current players workers for easy identification	

Game Setup Sequence Diagram:

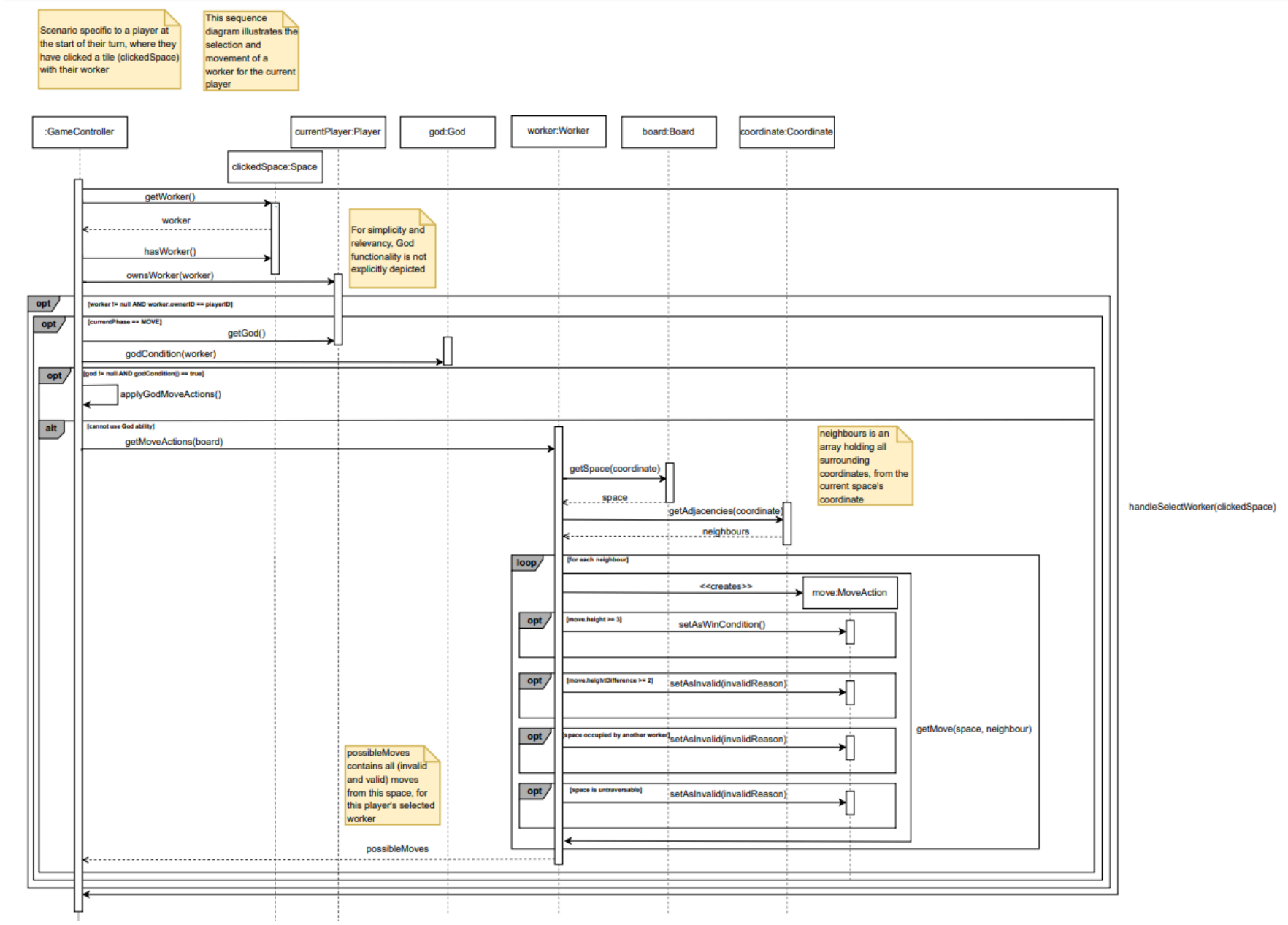


Worker Selection and movement with validation:

GUI Output:

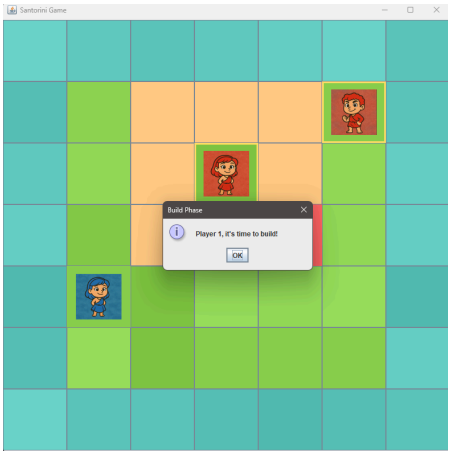
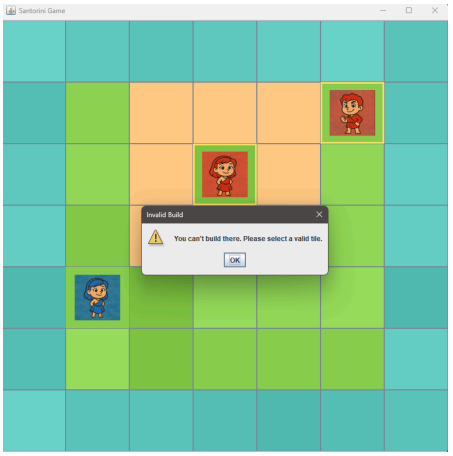
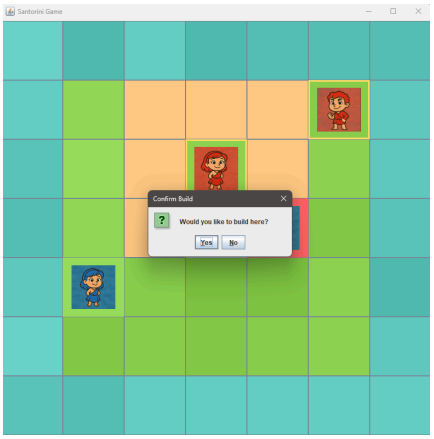
Action	Output
Move squares are highlighted upon clicking a worker. Valid move locations are highlighted in a light yellow colour while invalid spaces highlighted in red.	
A warning prompt outputted is an invalid space selected for movement.	
A move confirmation prompt outputted allowing user to check if they want to move their worker to the selected space.	

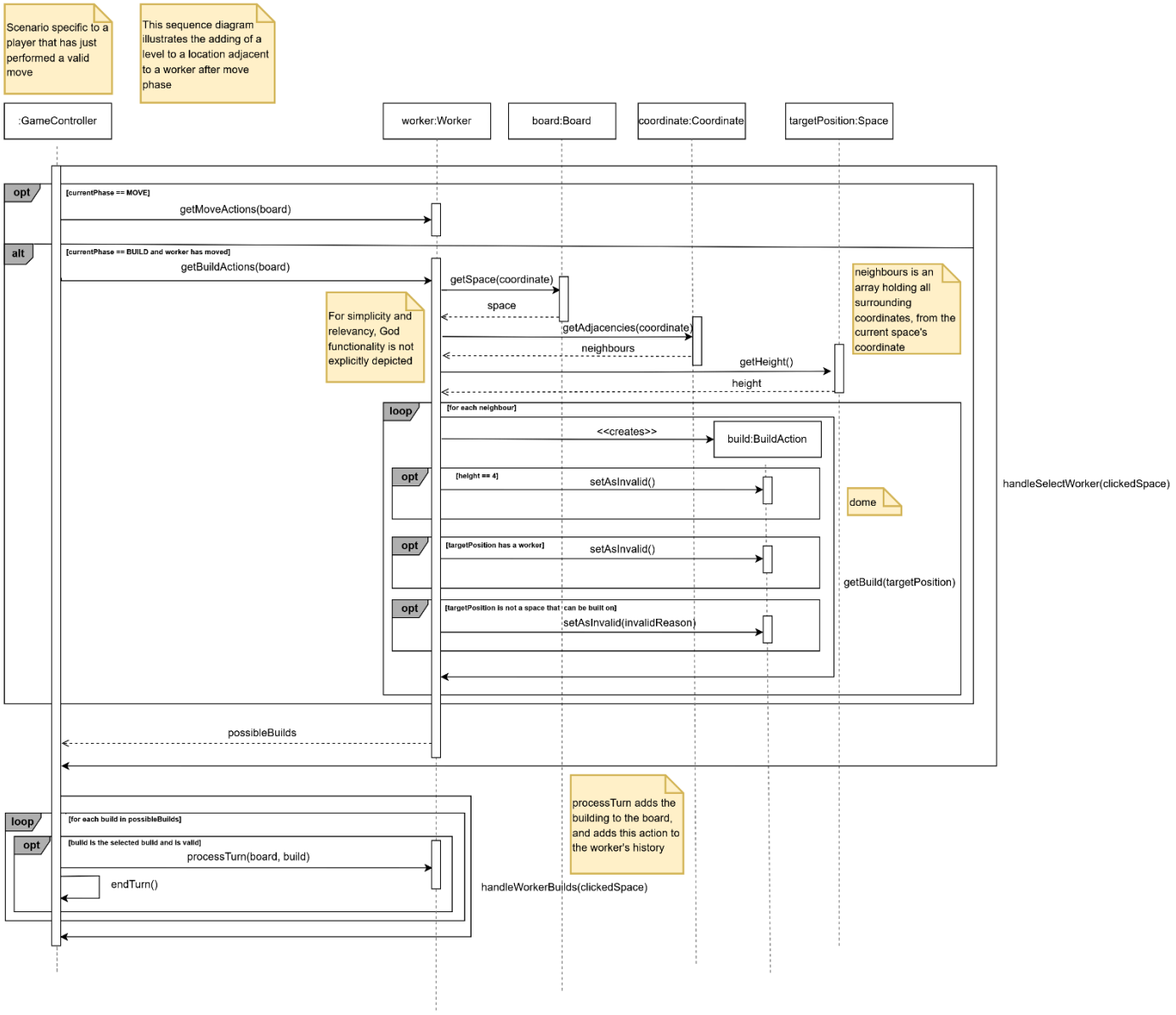
Worker Selection Sequence Diagram



Building:

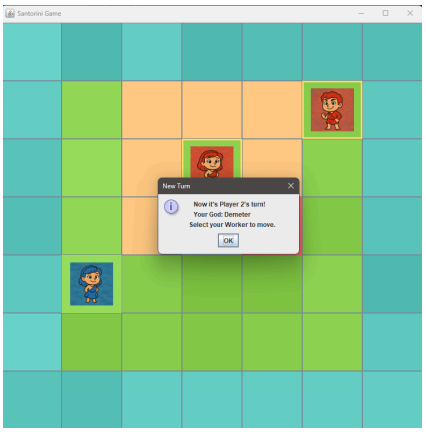
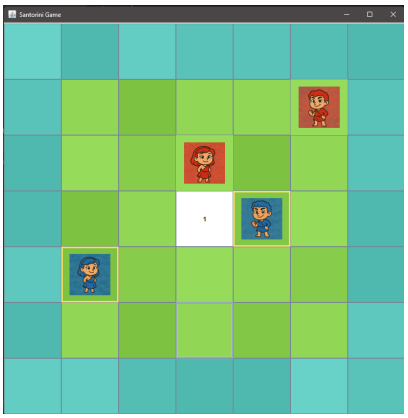
GUI:

Action	Output
Build phase transition prompt given upon completing move. Informs user that they can now build. Valid build locations highlighted in a pale orange colour while invalid spaces highlighted in a pale red.	
A warning prompt given to the user upon selecting an invalid square for building.	
Confirmation before building prompt given to user. Places a tower piece upon “Yes” selection with a number printed representing the tower height.	

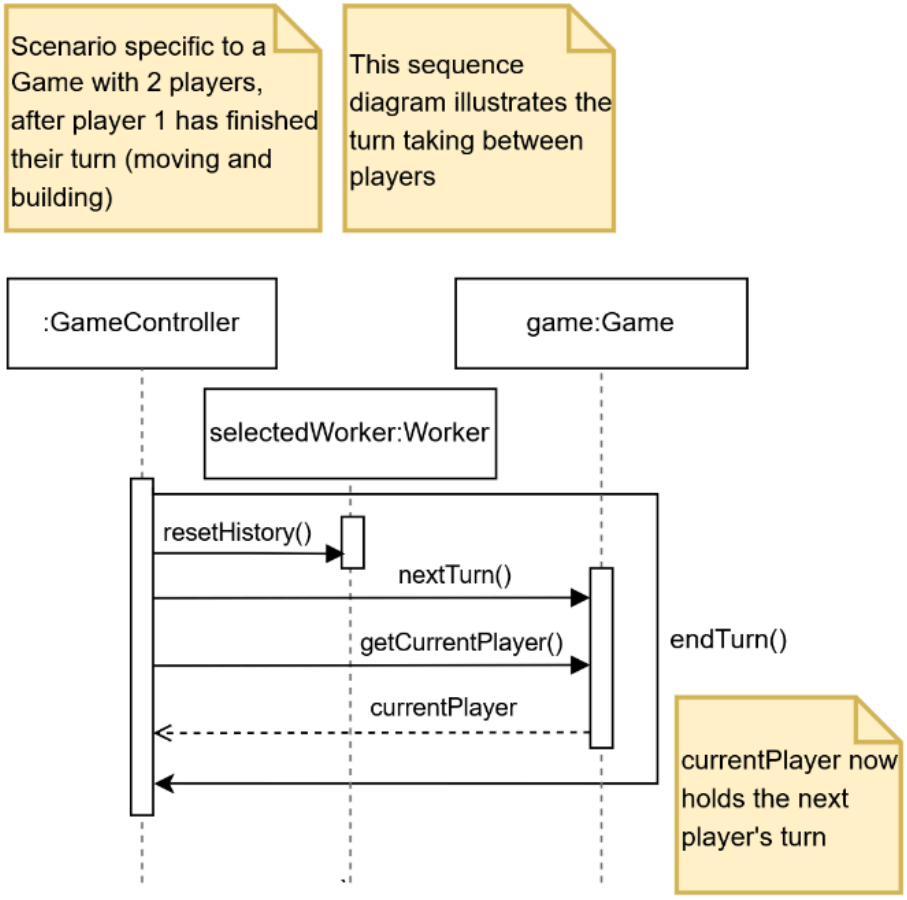

Scenario specific to a player that has just performed a valid move
This sequence diagram illustrates the adding of a level to a location adjacent to a worker after move phase
:GameController
worker:Worker
board:Board
coordinate:Coordinate
targetPosition:Space
opt
[currentPhase == MOVE]
getMoveActions(board)
alt
[currentPhase == BUILD and worker has moved]
getBuildActions(board)
getSpace(coordinate)
space
getAdjacencies(coordinate)
neighbours
getHeight()
height
For simplicity and relevancy, God functionality is not explicitly depicted
neighbours is an array holding all surrounding coordinates, from the current space's coordinate
loop
[for each neighbour]
<<creates>>
build:BuildAction
opt
[height == 4]
setAsInvalid()
dome
opt
[targetPosition has a worker]
setAsInvalid()
getBuild(targetPosition)
opt
[targetPosition is not a space that can be built on]
setAsInvalid(invalidReason)
handleSelectWorker(clickedSpace)
possibleBuilds
loop
[for each build in possibleBuilds]
opt
[build is the selected build and is valid]
processTurn(board, build)
endTurn()
processTurn adds the building to the board, and adds this action to the worker's history
handleWorkerBuilds(clickedSpace)

Change of Turn:

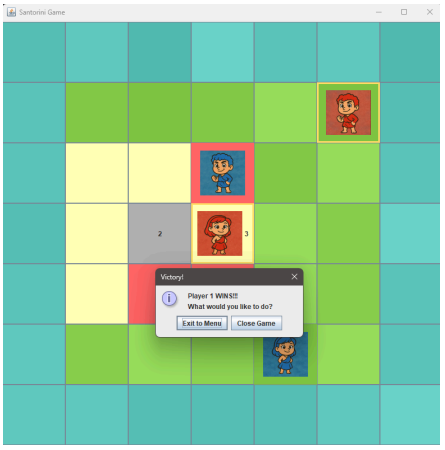
GUI:

Action	Output
Information message prompted upon build completion informing users that it is Player 2's turn. God power and abilities are also presented to remind Users their abilities.	
Player 2's workers are highlighted for selection	

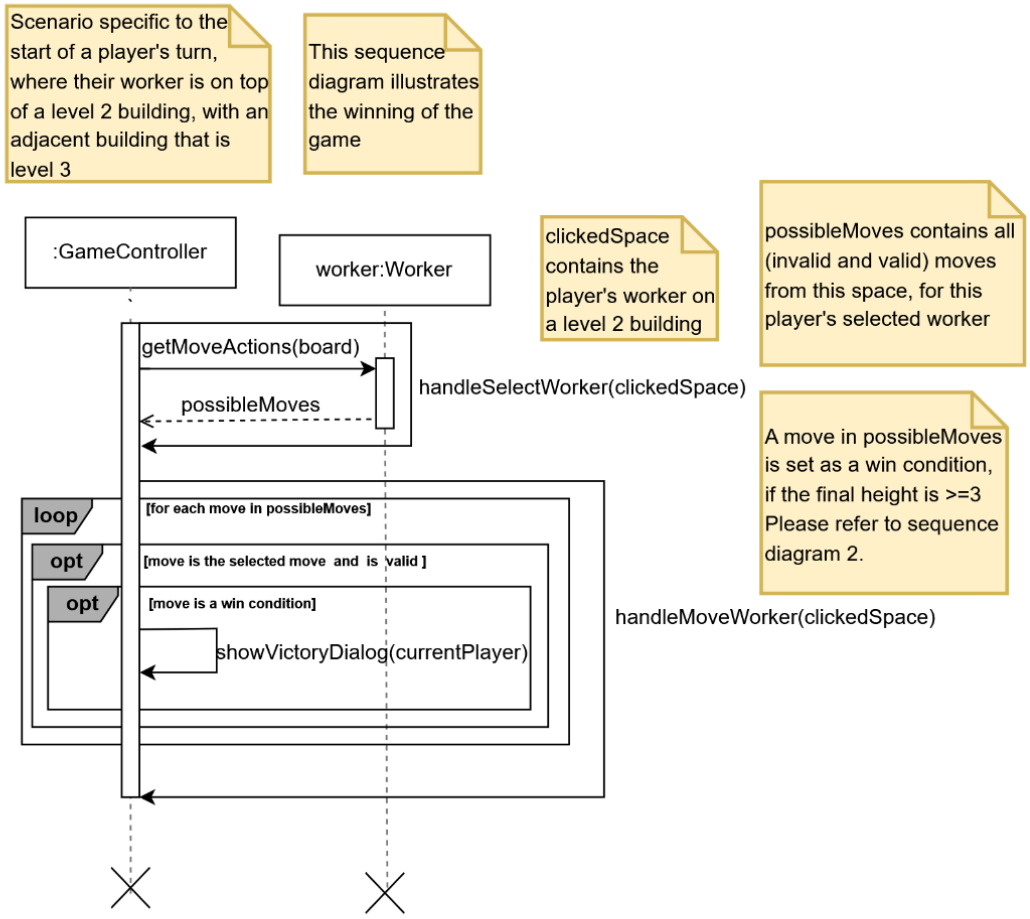
Sequence Diagram



Game Win:

Action	Output
Congratulations message printed for player informing the winning move. Users are prompted to move back to the Main Menu or exit the application.	

Sequence Diagram



Design Rationale:

Classes:

Action Class:

Class Description:

The Action class is an abstract class representing all possible operations that a player or worker can perform during the game, such as moving or building. It is designed to store meta data concerning whether an action ends a turn or if a winCondition can be met. It is extended by more specific classes such as MoveAction and BuildAction and supports double dispatch by allowing actions to interact with the board and having these interactions tracked through the history class. The action class removes the need for large if checks within the Player and Worker classes and greatly improves functionality.

Why we have included this class in the design:

- **Encapsulation of the game logic:**
    - Each Action instance contains all the necessary data and logic to execute itself, making the game engine more modular and easier to reason about.

- **Support for polymorphism and extension:**
    - By being an abstract class, new action types can be easily introduced without having to modify existing code. This adheres to the Open/Closed principle where code should be open for extension but closed for modification.

- **Improves cohesion:**
    - Keeping action-related functionality in one place prevents bloating of other classes like Player or Worker which would otherwise need to handle all the action logic.

- **Separation of concerns:**
    - The action class is responsible solely for the execution and state of individual actions, keeping other game elements like Board, Player and Worker focused on their own responsibilities. Adhering with Single Responsibility Principle

Why it was not appropriate to include it as a method.

- **Lack of reusability and scalability:**
    - If actions were implemented as plain methods (e.g. Player.performMove()), all the logic will need to be managed using large if-else or switch statements which would be hard to maintain and extend. It would break the Open-Closed principle.

- **No object-level state:**
    - Methods do not persist state, But actions like MoveAction may carry information such as the Worker involved, the result of the move and whether it ends the turn. Storing this within a method would require external tracking resulting in poor encapsulation.

- **Difficult to pass around or store:**
    - Actions can be passed as objects to other systems. This isn't possible with methods.

- **Violates Open/Closed Principle:**

- Introducing new actions would require altering core game logic if implemented as methods. Using the current design allows new Action subclasses to be introduced with no disruption to existing classes.

Space Class:

Class Description:

The Space class represents a single tile on the game board. It stores information such as its position (Coordinate) and the stack of piece objects. It can also hold a reference to a Worker, which is stored separately to support traversal checks and gameplay rules. This class encapsulates all the behaviour and state specific to a board tile such as checking buildability, height or whether a move is illegal.

Why we have included this class in the design:

- **Encapsulation of tile-level logic:**
  - Abstracting each board cell into its own class allows it to isolate behaviour related to a tile, such as building or movement checks and worker placement.

- **Separation of concerns:**
  - The Board classes focusses on high-level management between tiles, while Space handles local tile-specific details. This aligns with the Single Responsibility Principle which states that a class should be responsible for one thing.

Why is was not appropriate to include it as a method?

- **Would violate abstraction:**
  - If the tile logic were implemented as a series of helper methods inside the Board class it would tightly couple the board to all tile-level logic. This would violate low coupling and high cohesion principles.

- **Unscalable and error-prone:**
  - A method-only approach would require redundant code to manipulate coordinates and stacks. This would cause the Board class and thereby the game to be more complex and harder to maintain.

- **Encapsulation would be broken:**
  - Internal states would need to be exposed to other classes to allow manipulations. By using the Space class, we ensure that these details are hidden and manipulated only through well-defined methods.

History Class:

Class Description:

This History class is responsible for tracking the sequence of actions taken by a player during a turn. It stores this information in separate lists for general Actions. This record is important for validating rules, checking functionality, enabling god powers and allowing for potential undo functionality.

Why we have included this class in the design:

- **Isolates the responsibility of tracking turn-based behaviour:**
    - This would otherwise have to be tracked inside the Player class. This refactoring allows for creating god classes and supports the Single Responsibility Principle.
- **Allows for modular and lightweight communication between components:**
    - A god like Artemis must know whether the worker has already moved and the history class would provide this information without direct access to the Player internals.
- **Supports flexible architecture:**
    - Can easily test history based logic independently and be extended easily into new game features. Maintains low coupling between gameplay logic and state tracking.

Why was it not appropriate to include this as a method?

- **It would bloat the Player class:**
    - Keeping action history within the Player would introduce excessive state and behaviour, turning it into a god class.

- **Would result in tight coupling and low reusability:**
    - Method based history tracking would likely hardcode logic in the Player class or global game state, making it hard for other components to query or manipulate history without unnecessary dependencies.

- **Inflexible for future features:**

    - Undo systems, replay logs or god powers that depend on past moves would require reworking scattered logic spread across various classes. Having a dedicated History class keeps this logic centralised and extensible.

Relationships:

Why is board an aggregation of spaces?

Board is an aggregation of spaces and not composed of spaces because of the necessary relationship between spaces and board. Board purely exists as a collection of spaces, along with methods necessary for managing all of the spaces. However, in theory the spaces can exist independently of the board, as a single point holding pieces.

Why does space hold an association with workers and pieces separately?

Workers are a unique type of piece, and as such since there can only be one per space, and they get called frequently with methods that can't be found in their super class, they have are held separately from the rest of the pieces. The other pieces exist in the space class in their abstracted form, since their superclass is all that's needed to get their important values. This just makes it easy to interact with the worker, while simultaneously allowing for a degree of polymorphism with it. The alternative is to place worker in the stack of pieces in space, but this would mean either having to scan through all of the pieces and get their null moves or breaking open close principle in checking for workers.

Why does the game have a dependency on gods?

The game class functions as the centralised controller of all of the logic in the game, and as such organises things that the decentralised players need coordination for. This includes distributing gods, since there is only 1 copy of each god in play, and also certain restricted matchups of gods. As such, the central game class generates the god classes in play and distributes them to the players, making sure there are no double ups, and as such holds a dependency but not association. This is much more preferable to the alternative of a decentralised approach, in which players would need to pass around the gods that they have chosen and are remaining, requiring the players to know who's turn it was next, if a god class has been chosen, polling other players to see if their god card is banned, it's just simpler to use a centralised system.

Inheritance:

Inheritance is a key part of our design. While we wanted to prioritise composition over inheritance where we could, there were instances where it was clear that inheritance was the superior way of proceeding. For example, using inheritance with the action class allows us to utilise double dispatch, so that each different action could process the board in a different way. Similarly with the god class, using inheritance and overriding base methods was easier than passing a god commands to apply, since each god had widely different behaviours from each other, and there aren't ever any exchanging of parts of god abilities, so it's better to force all aspects of the god abilities to be conjoined.

Cardinalities:

Worker cardinalities: 0..2

In Santorini, the most workers a player can have is 2. This is a limit of the board game, as you cannot pull out more worker pieces from the aether. This is the justification for the upper bound of 2, since a player will never own more than 2. As for the lower bound, there exists a god named Bia, who has the ability to delete workers from the board. Bia has the capability to remove 1 or both of the workers a player owns, so therefore the range of workers a player could own at any point in time is between 0 and 2.

Board to spaces cardinalities: 9..*

At an absolute minimum, we decided that the smallest shape a board should be in Santorini is 3x3. This is enough space for players to be able to build just enough towers, and to be able to climb up them. As such, since any smaller than 9 tiles would result in a non-functional game, we set 9 spaces as the lower bounds. Since we have implemented the board in a way to be a dynamic size, we can have in theory an infinite size of board, thus the upper bounds is many.

Design Patterns:

Worker Factory - Factory Pattern:

There's a lot of data that the worker holds that is dependent on external conditions. For example, the colour of the worker, the players id number, the gender of the worker, different graphics for different workers, etc. Instead of trying to construct the worker object by pooling all of this information separately, we instead make a factory class that manages the construction. The benefit of a factory class like this is that it can handle the logic for how many workers need to be constructed, what gender of the worker, what colour, etc. This helps abstract all of the messy implementation, so that whenever we need to create a worker in the code, we don't need to duplicate all of the tricky logic, and just use a seamless interface.

Action Class (And inherited classes) - Command Pattern (and double dispatch)

The action class is a prime example of the command pattern, as each action is a possible event that can trigger on the board to modify it. The exact method that the modification is triggered utilises double dispatch across its inherited classes. For example, the build class takes in the board class, and calls on it the build piece function. Keeping the functions encapsulated in an object means that we place them into a list to track their order, which can allow for an undo function down the line, and also gives a succinct list to the player of options that they can take. The alternative would require some hacky functional programming of passing around structs of data that would then need to be processed by the player onto the board, which would require more dependencies.

Action Class (And inherited classes) and Player / God Class - Builder Pattern

The action class is also an example of the builder pattern. Basically, although the worker generates all of the move and build actions, it doesn't know the history of other workers, or whether the player's god class activates. That's where the builder pattern comes into play. The player and god class can modify aspects of the action, like whether or not it ends the turn, and whether or not it causes the player to win, allowing for flexibility in how actions function but also separation between making and modifying the possible actions. The alternative is for the object to obsess about all of the relevant information at construction, which is impractical since then the worker would need knowledge of everything.

Design Changes:

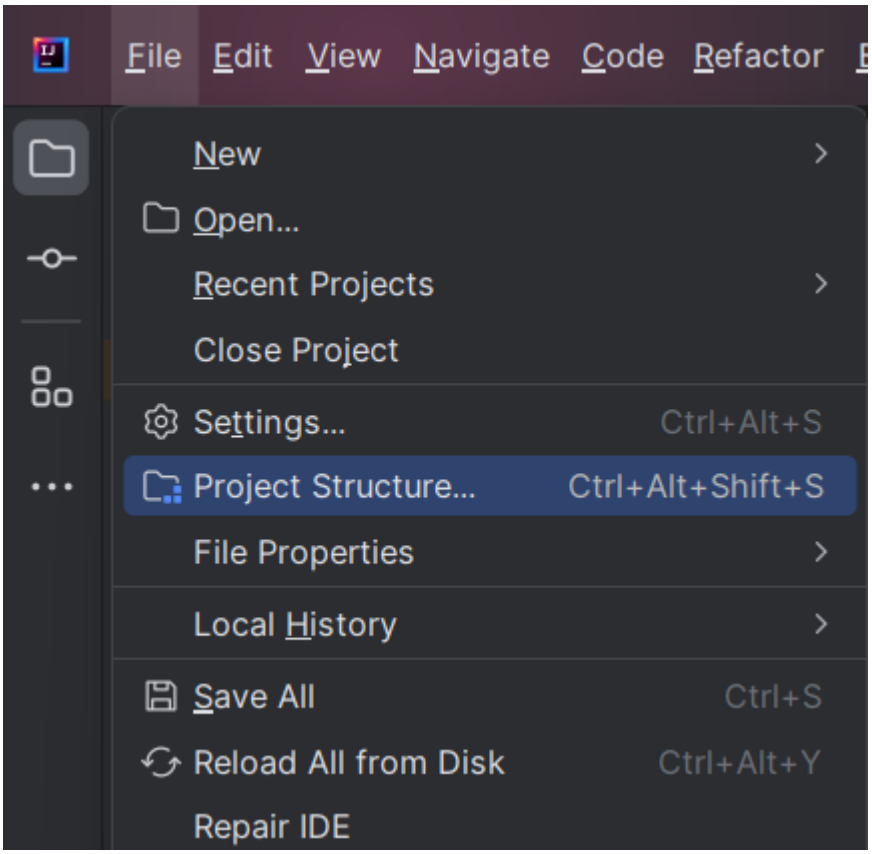
One of the concerns when we started implementing the player class was that it was becoming too much of a god class. It was generating a lot of actions, and handling a lot of the logic to do with those actions. So we made the decision to move all of the action generating functions into the worker class, since it is the worker that generates these actions. This helps better distribute the workload, and reduces the over complication of the player class, which now just handles minor modifications to the actions and move selection.

Another minor change was creating a dedicated history class to hold the data and methods for managing the players move histories. This is mostly to hide some of the messy coding required for checking and managing histories, and to reduce the responsibilities of the player class.

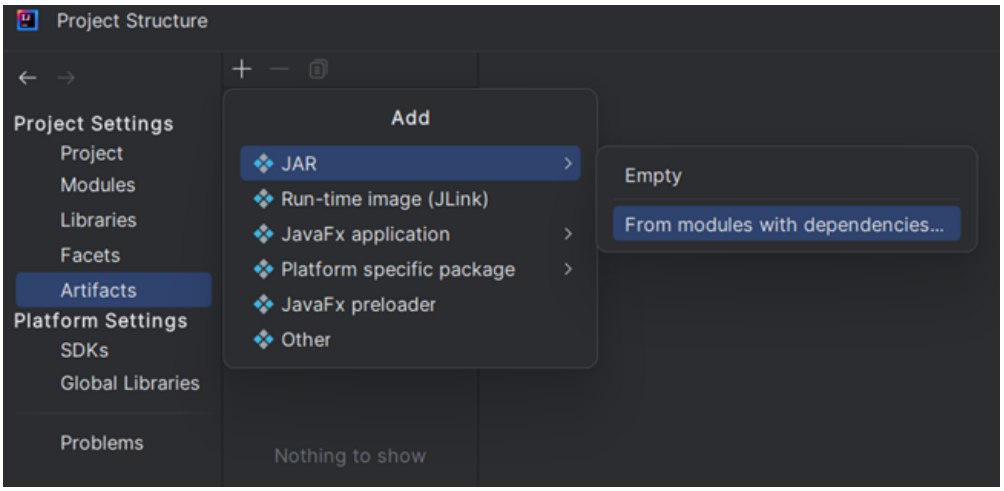
Building and running an executable:

version: Oracle OpenJDK Version 24

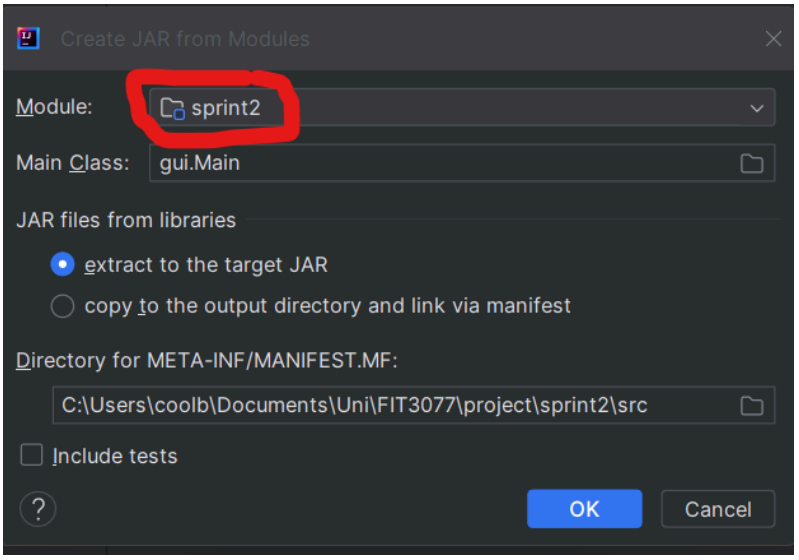
1. Open the folder ‘sprint2’ in IntelliJ.
2. Navigate to Project Settings by selecting File -> Project Structure.



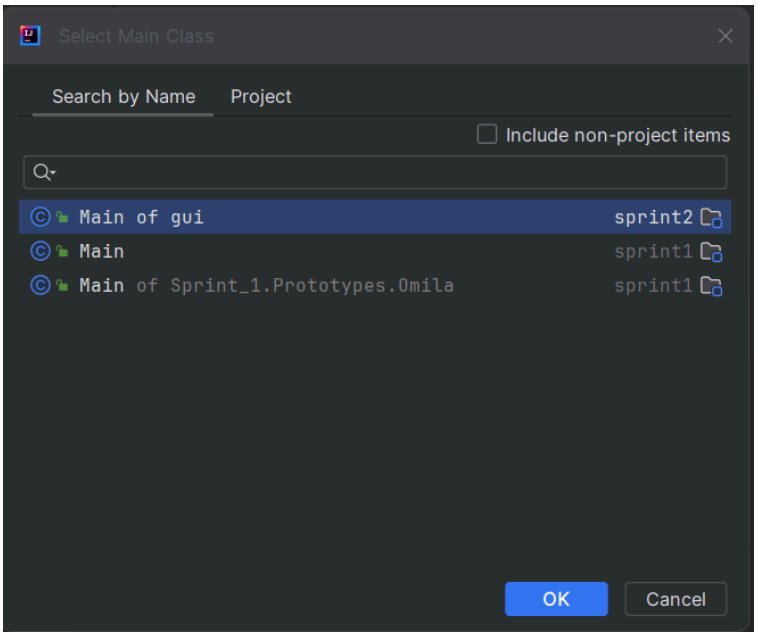
3. Click on 'Artifacts', then select the '+' icon to generate a JAR file.
4. Select 'from modules with dependencies'.



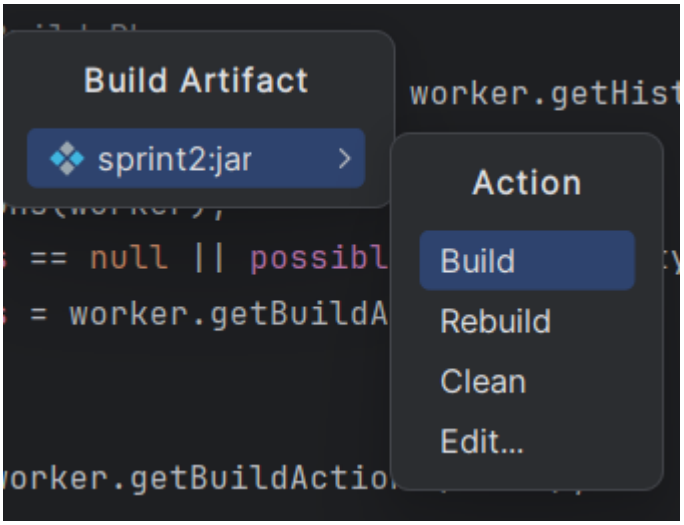
5. Select ‘sprint2’ as the module, then click on the 'Main Class' text box



6. Select ‘Main of gui’ and press OK.



7. Press OK again to close the window, then select Build -> Build artifacts -> build.



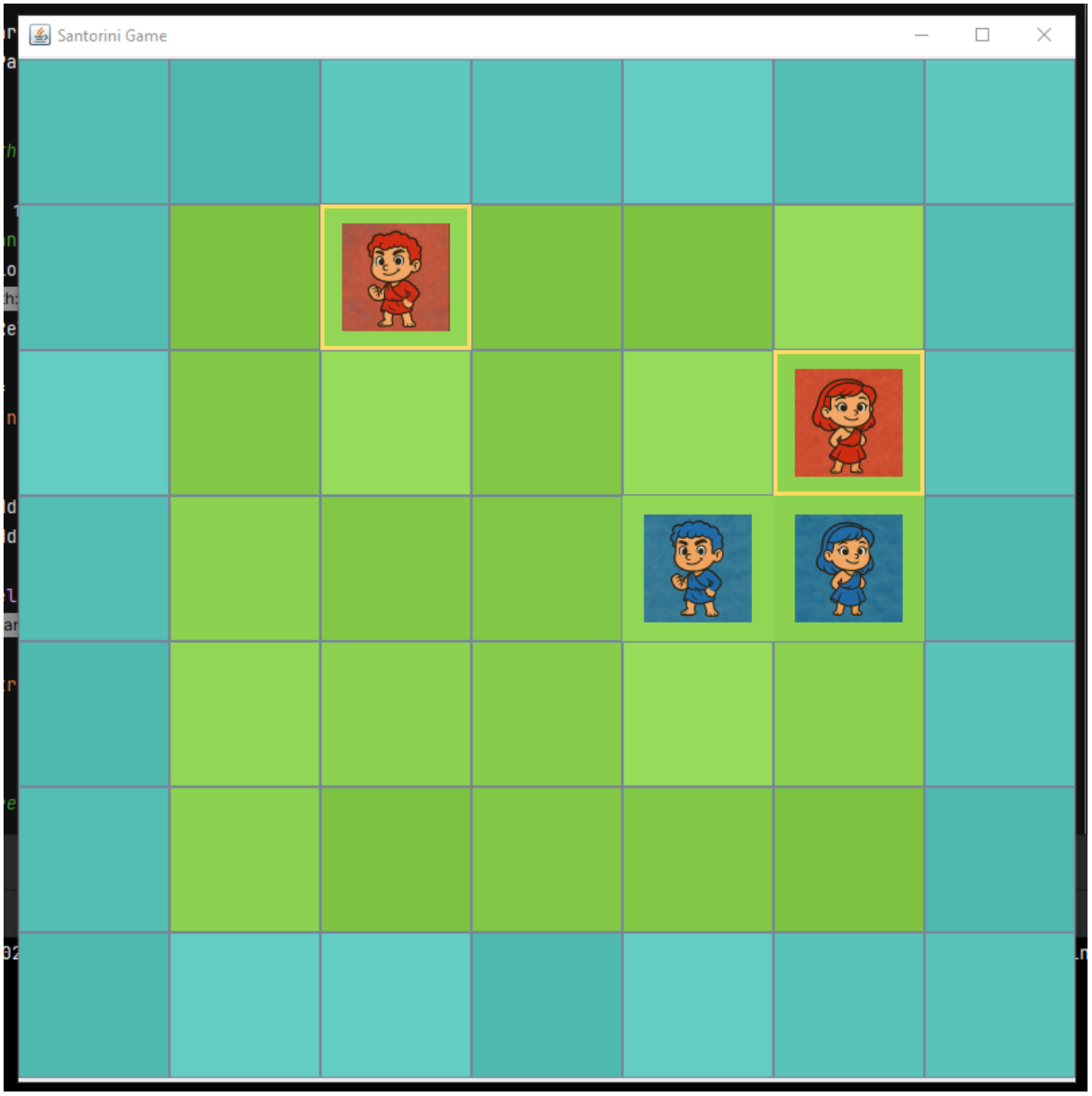
8. Next, open your terminal and navigate to the jar file built. This file will be inside the 'out' folder (project\out\artifacts\sprint2_jar)

9. Next, type 'java -jar sprint2.jar' in the terminal.

Documented Game Testing:

Initial board init:

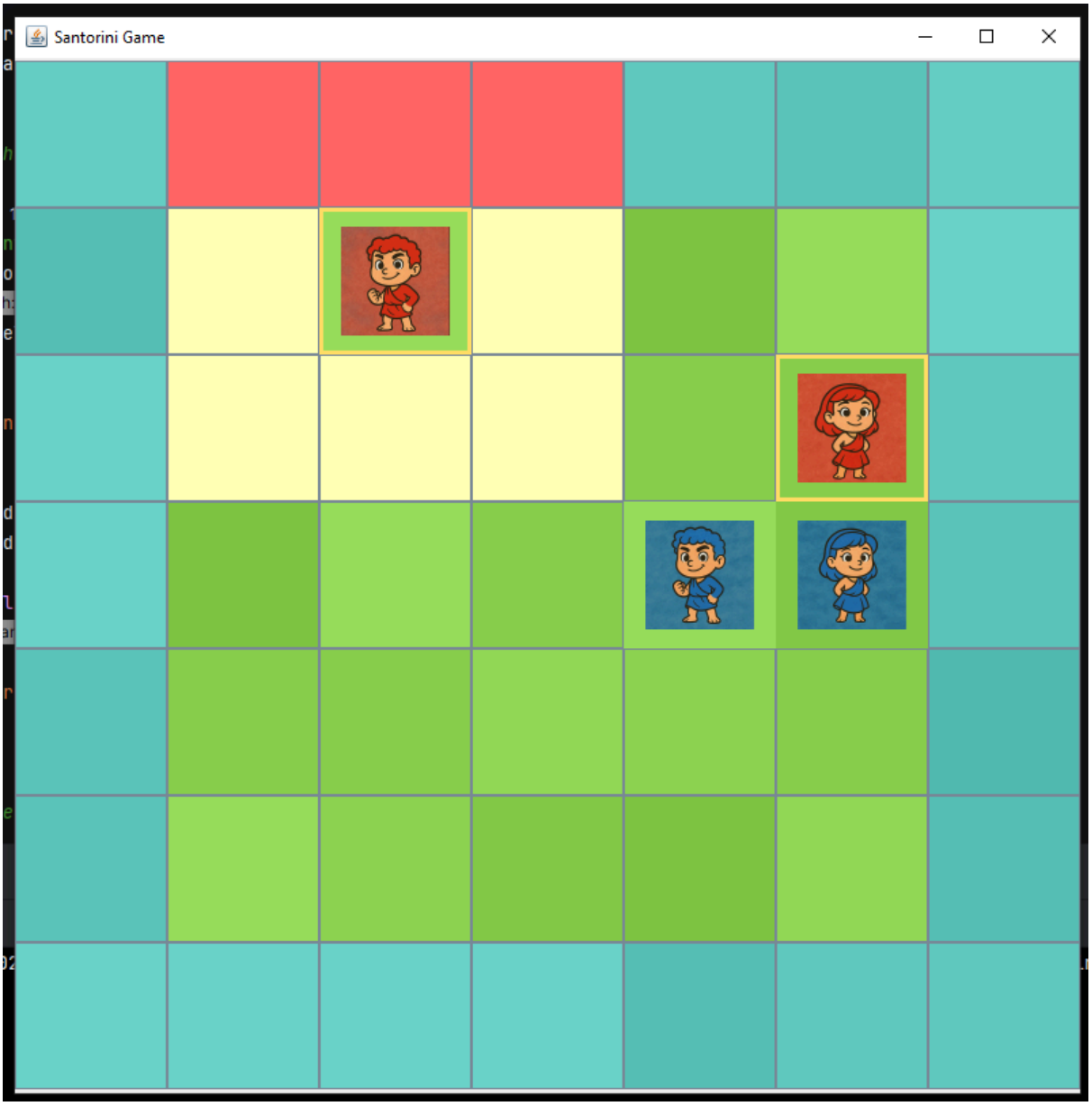
Expected outcome:
5x5 green tiles, surrounded by a ring of ocean tiles, with workers at random positions.
Actual Outcome:



Notes:
Working as expected.

Worker Selection:

Expected outcome:
Ring of possible selections around the clicked on worker, with yellow representing valid moves and red representing invalid moves.
Actual Outcome:

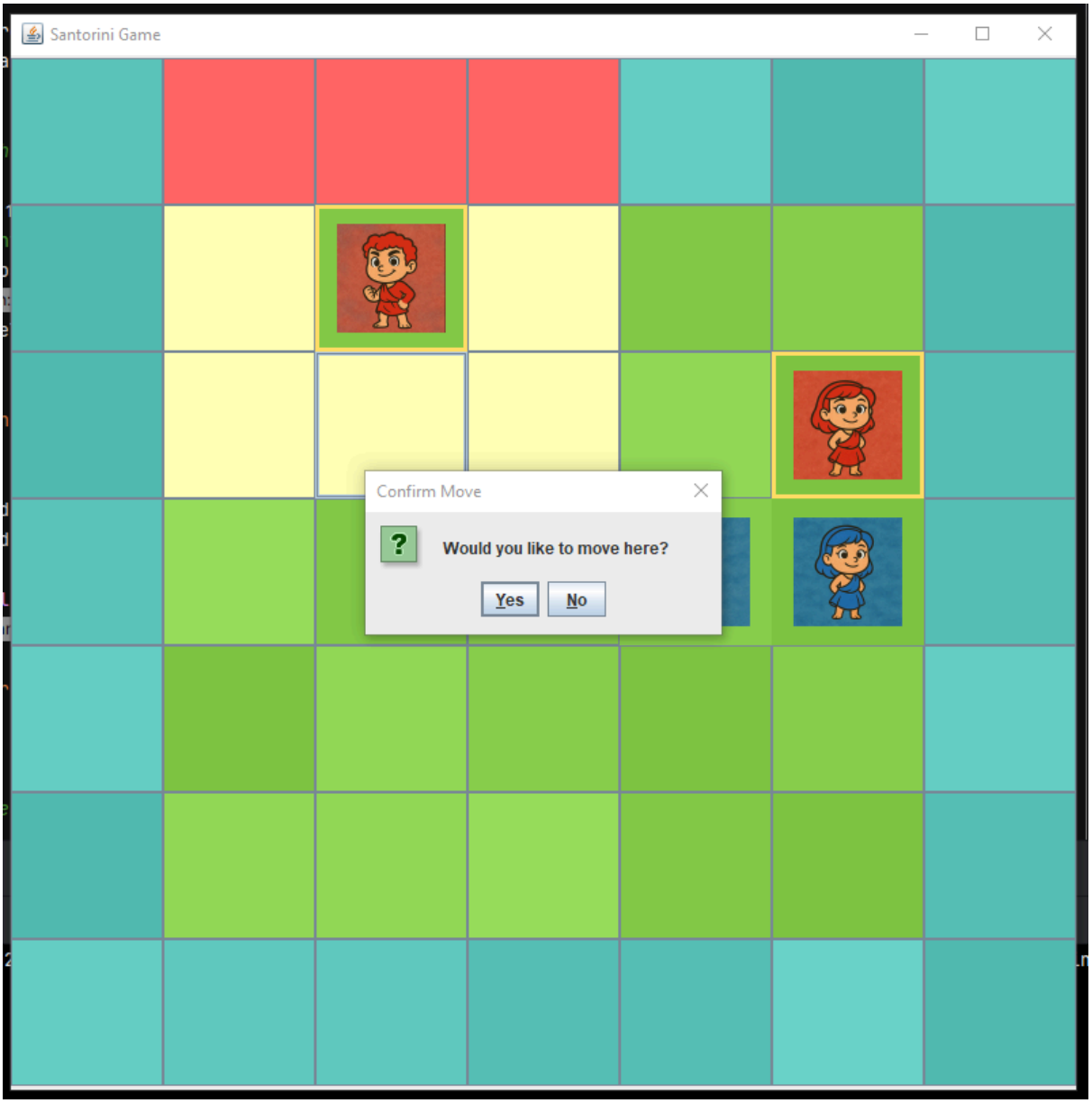


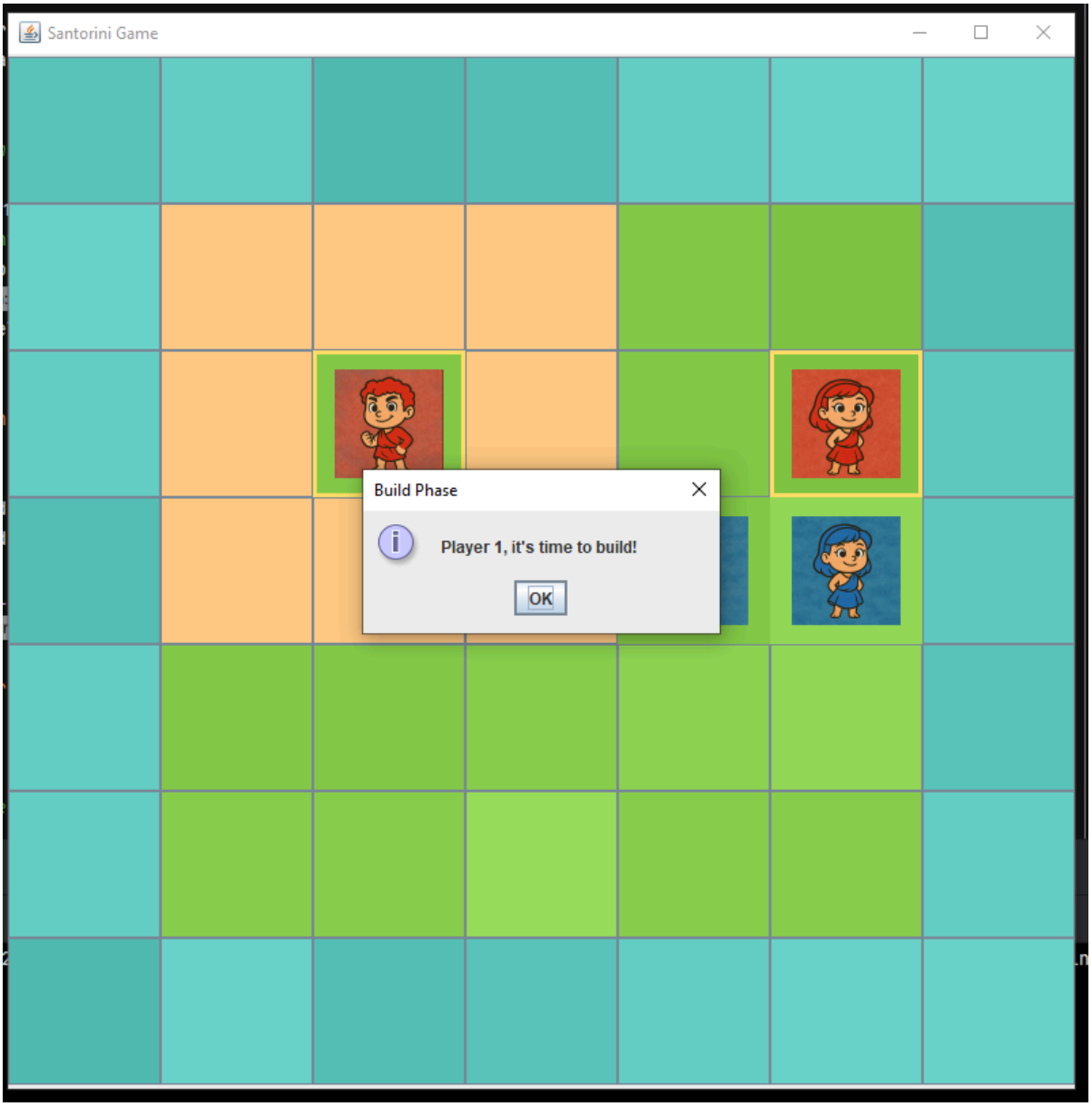
Notes:
Working as expected

Valid Selections:

Valid Move Selection:

Expected outcome:
Valid moves are highlighted in yellow. Clicking on them brings up the confirm move popup. Clicking yes moves the worker to that position.
Actual Outcome:

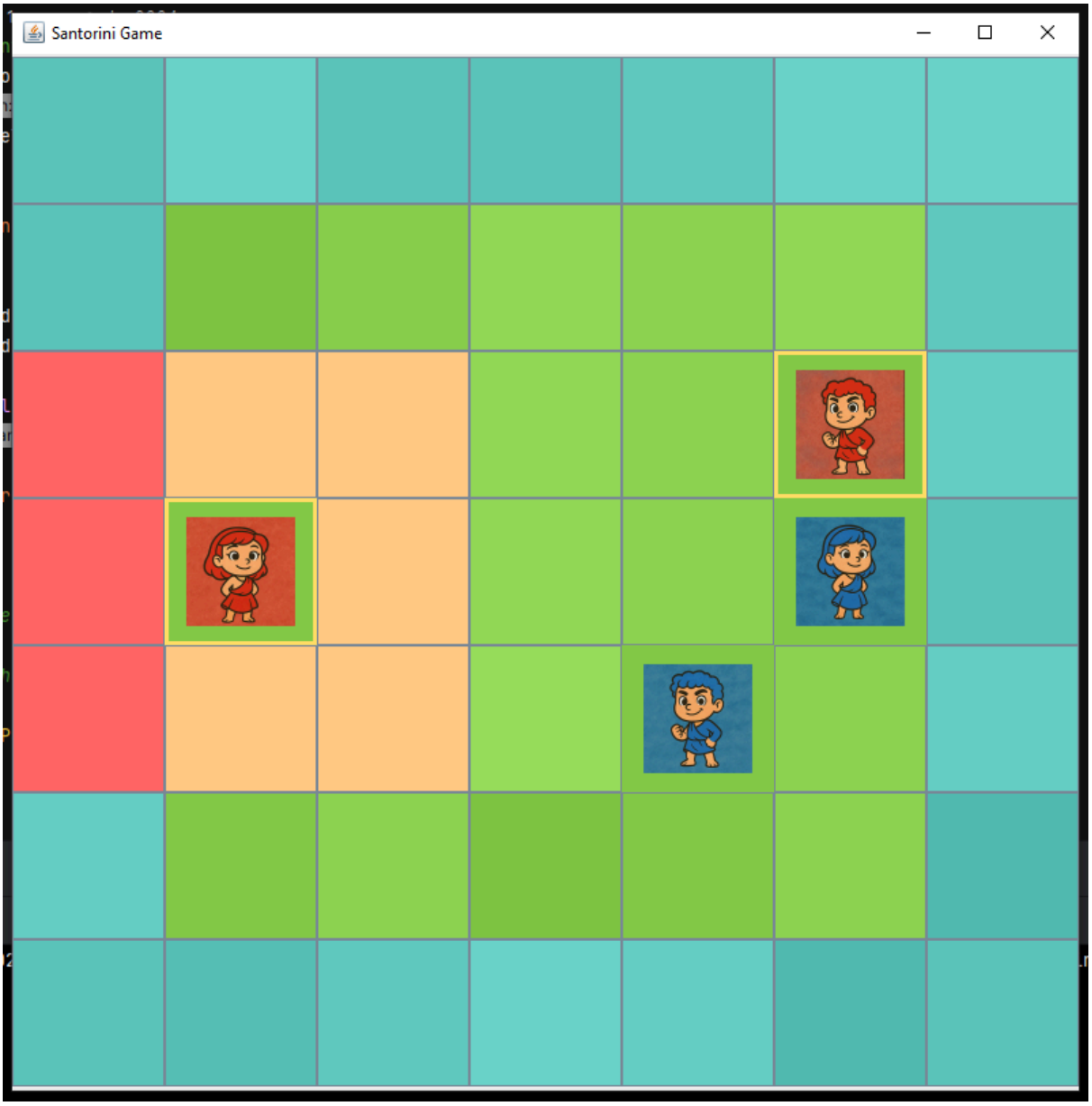

Santorini Game
Confirm Move
Would you like to move here?
Yes
No

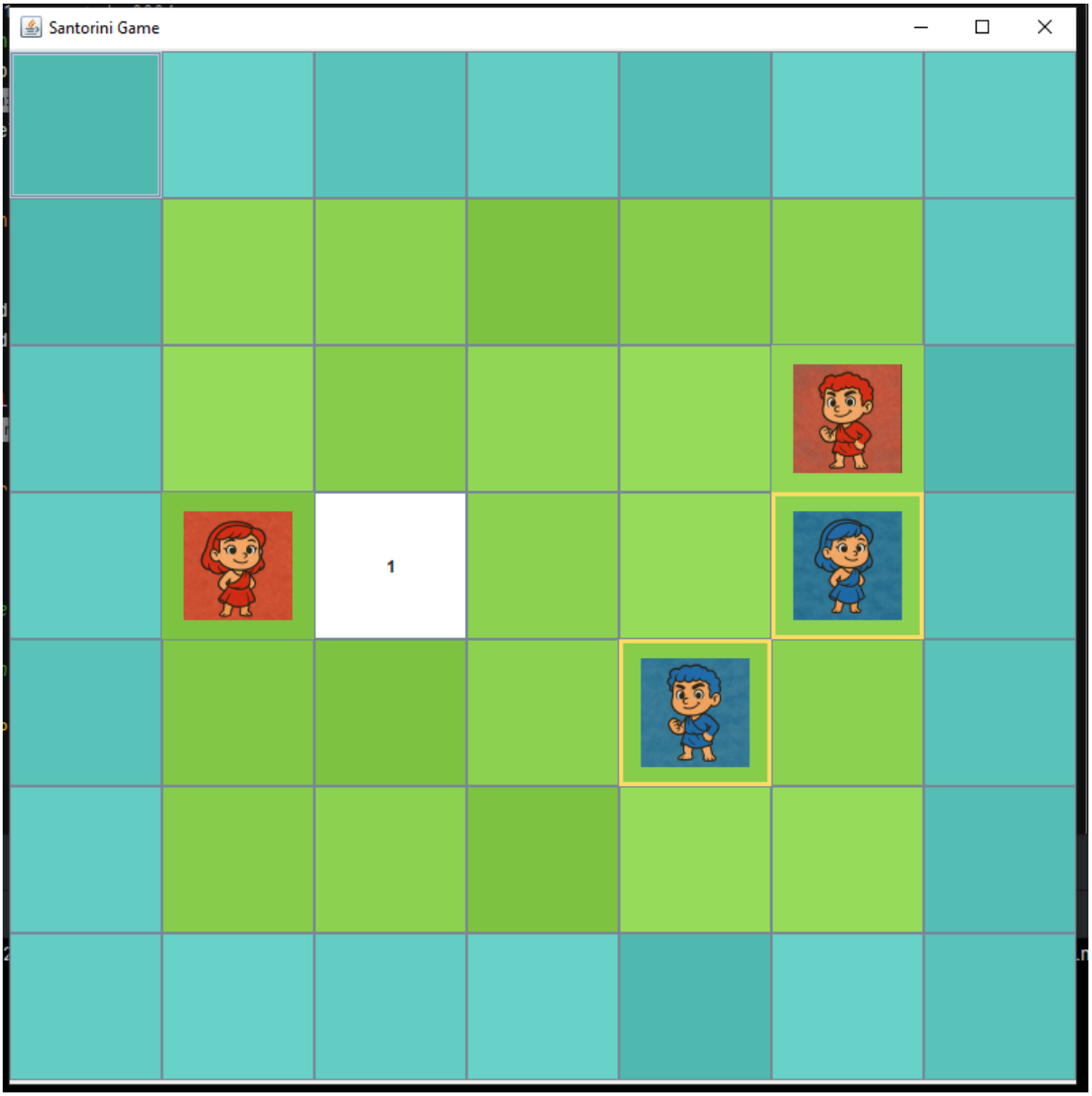


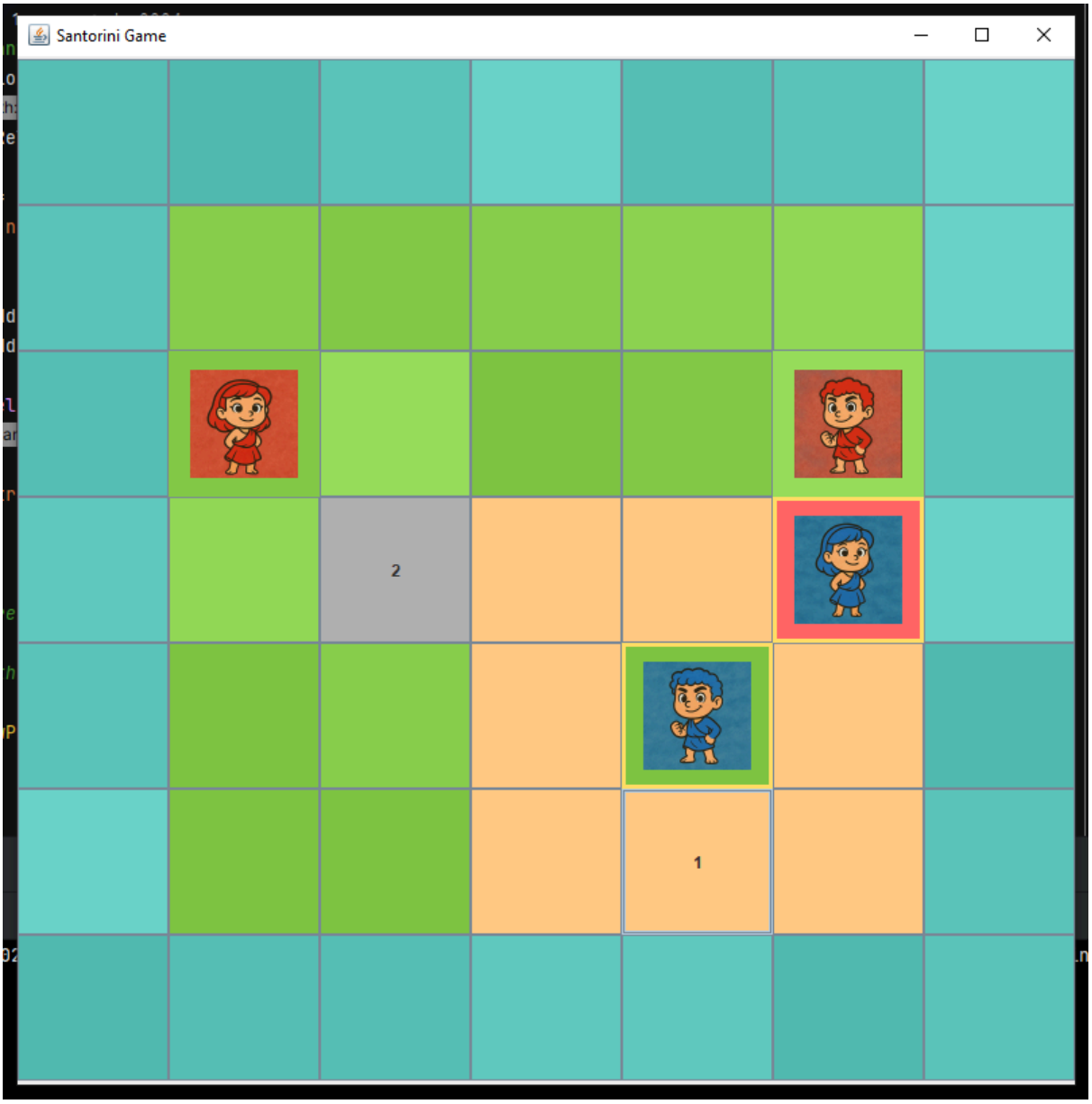
Notes:
Working as expected

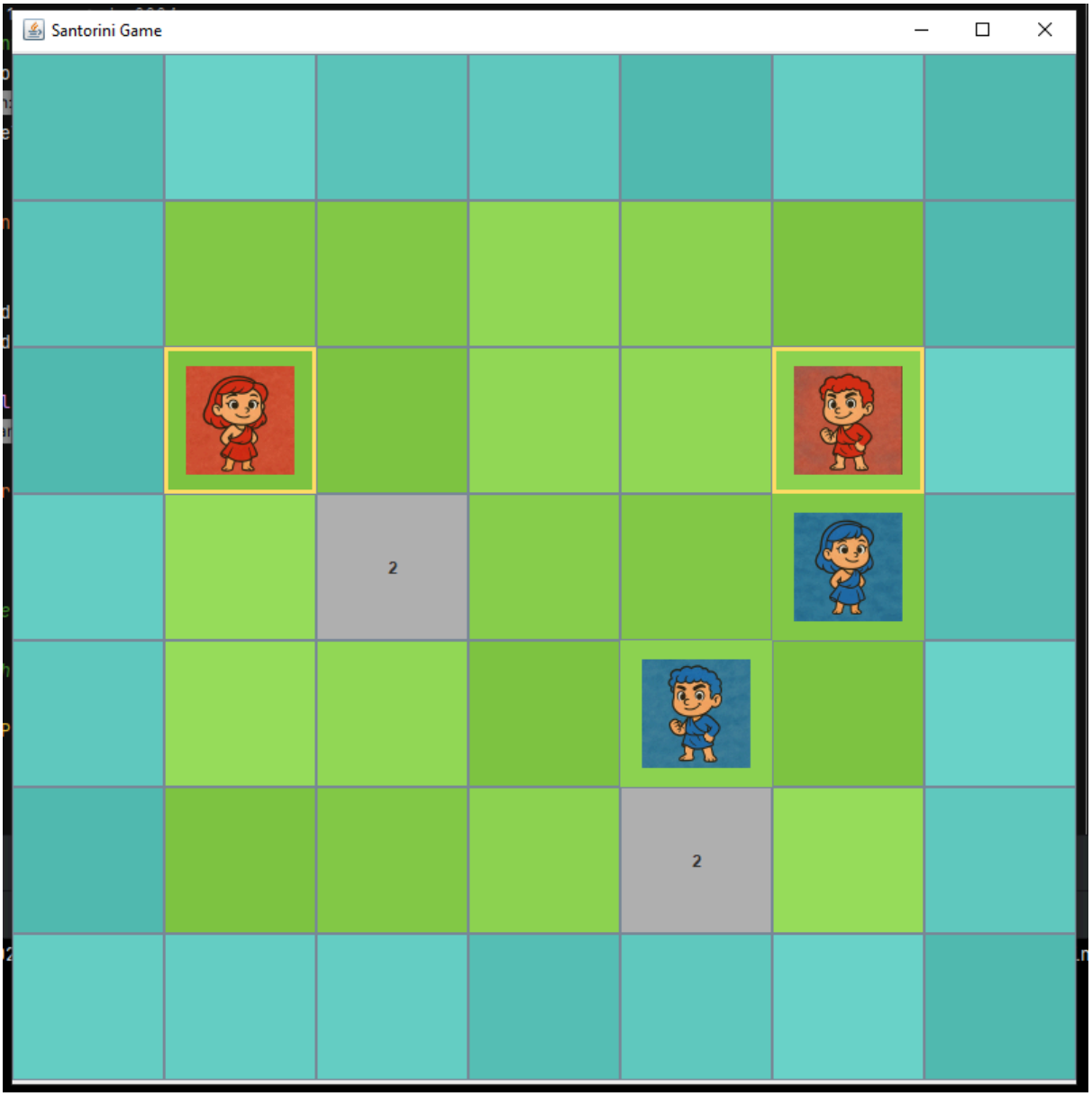
Valid Build Selection:

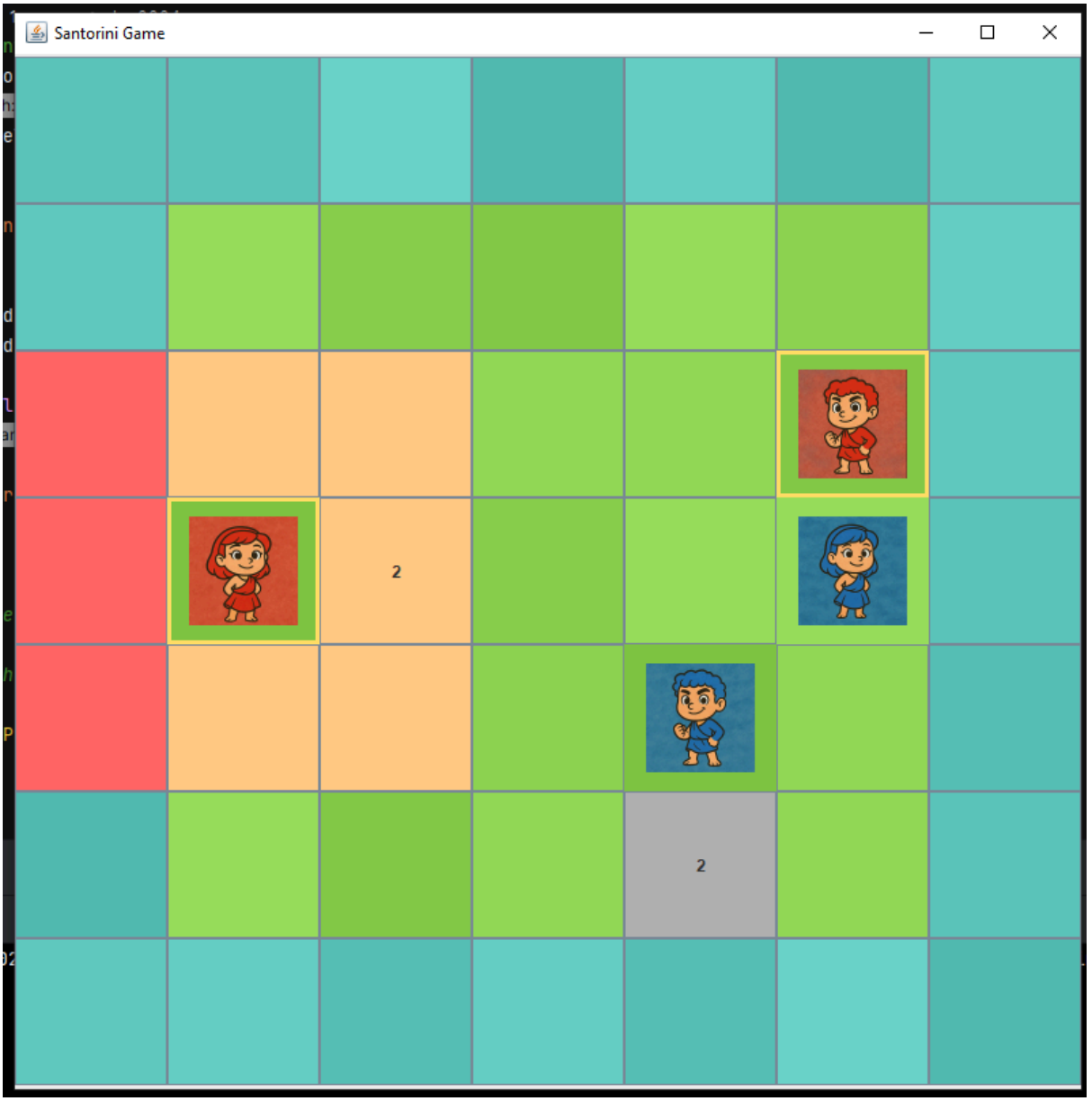
Expected outcome:
Building on an empty ground places a 1 block. Building on 1 makes it 2. Building on 2 makes it 3. Building on 3 makes a dome.
Actual Outcome:

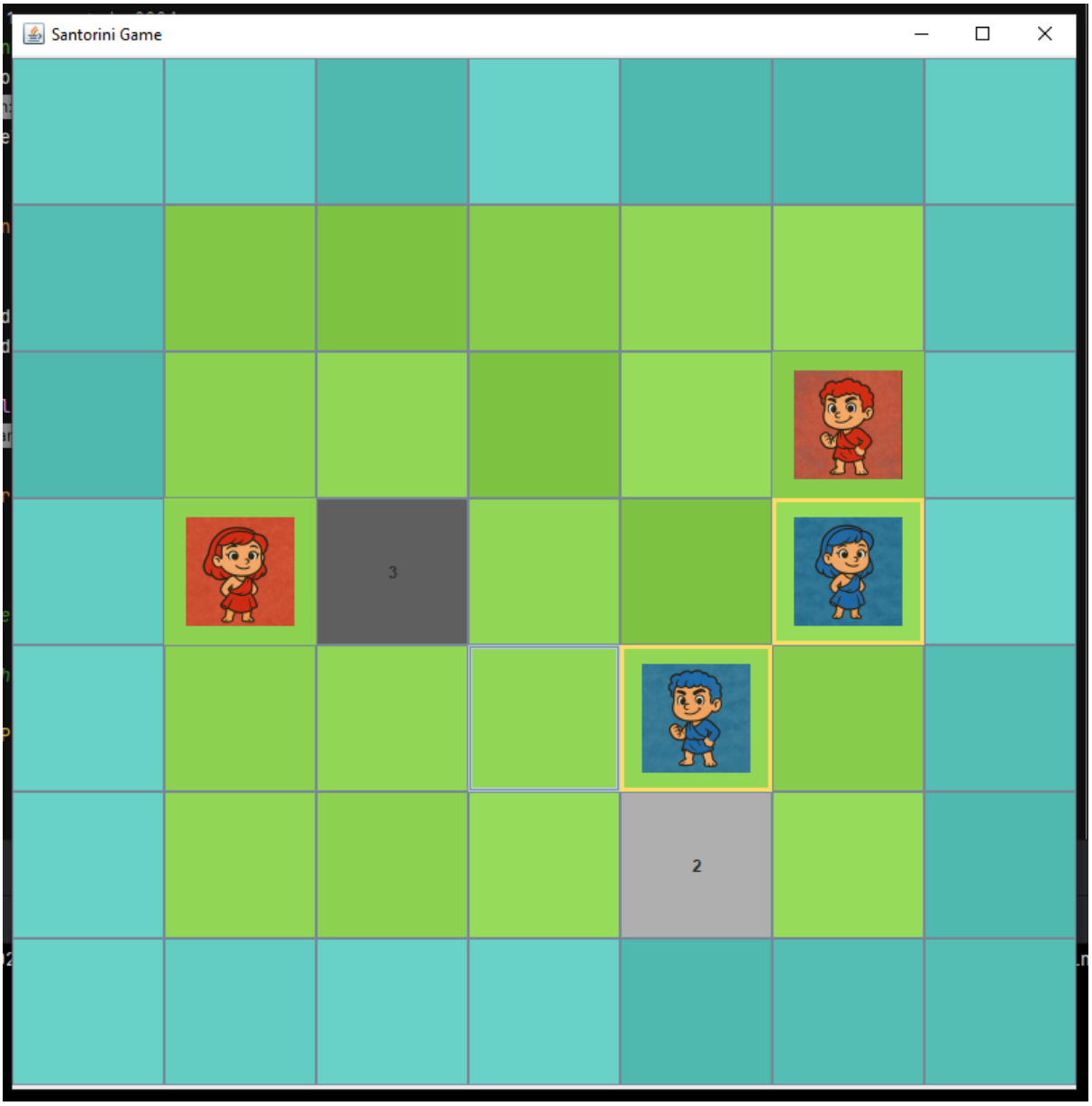

Santorini Game

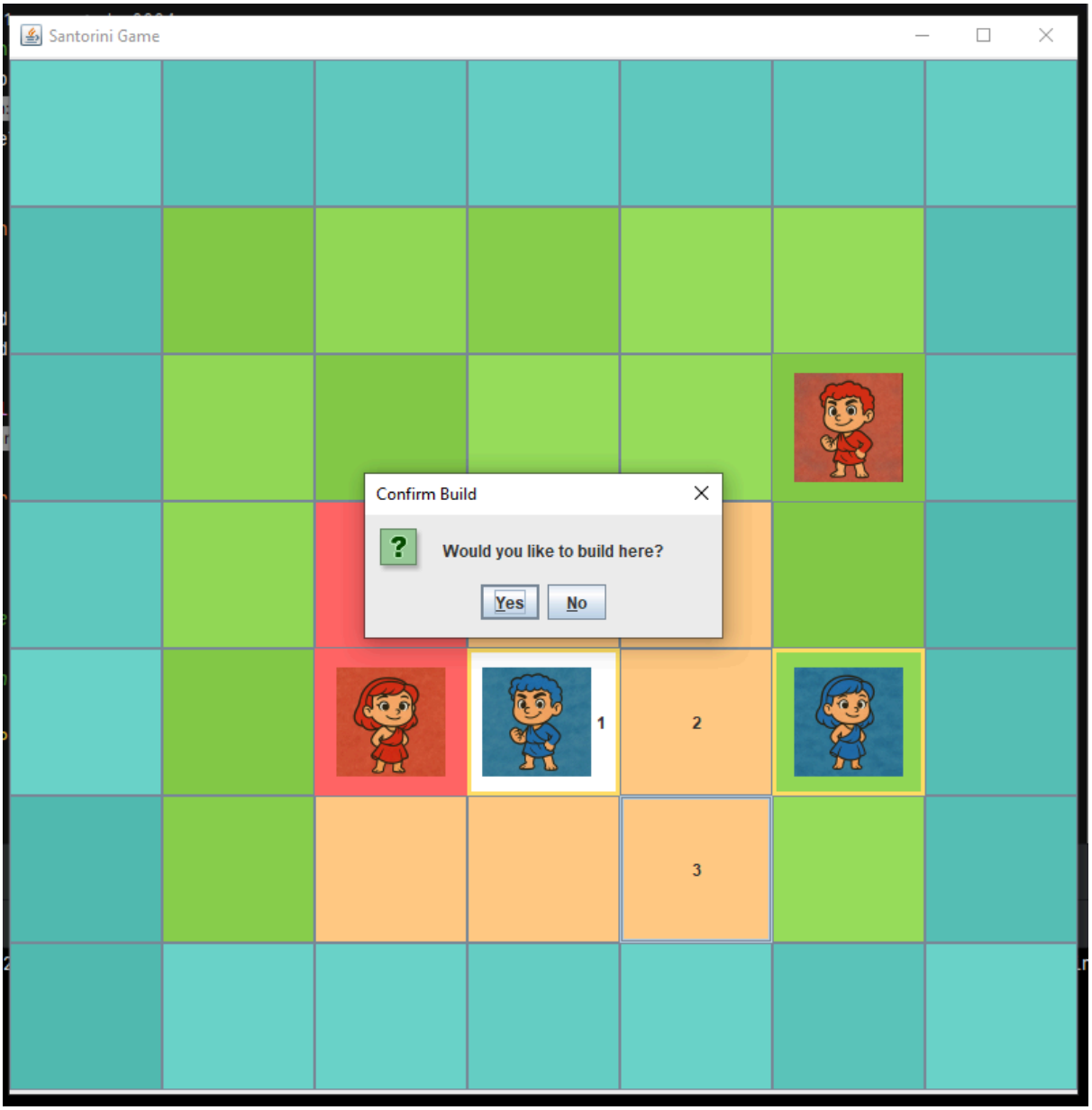

Santorini Game
1

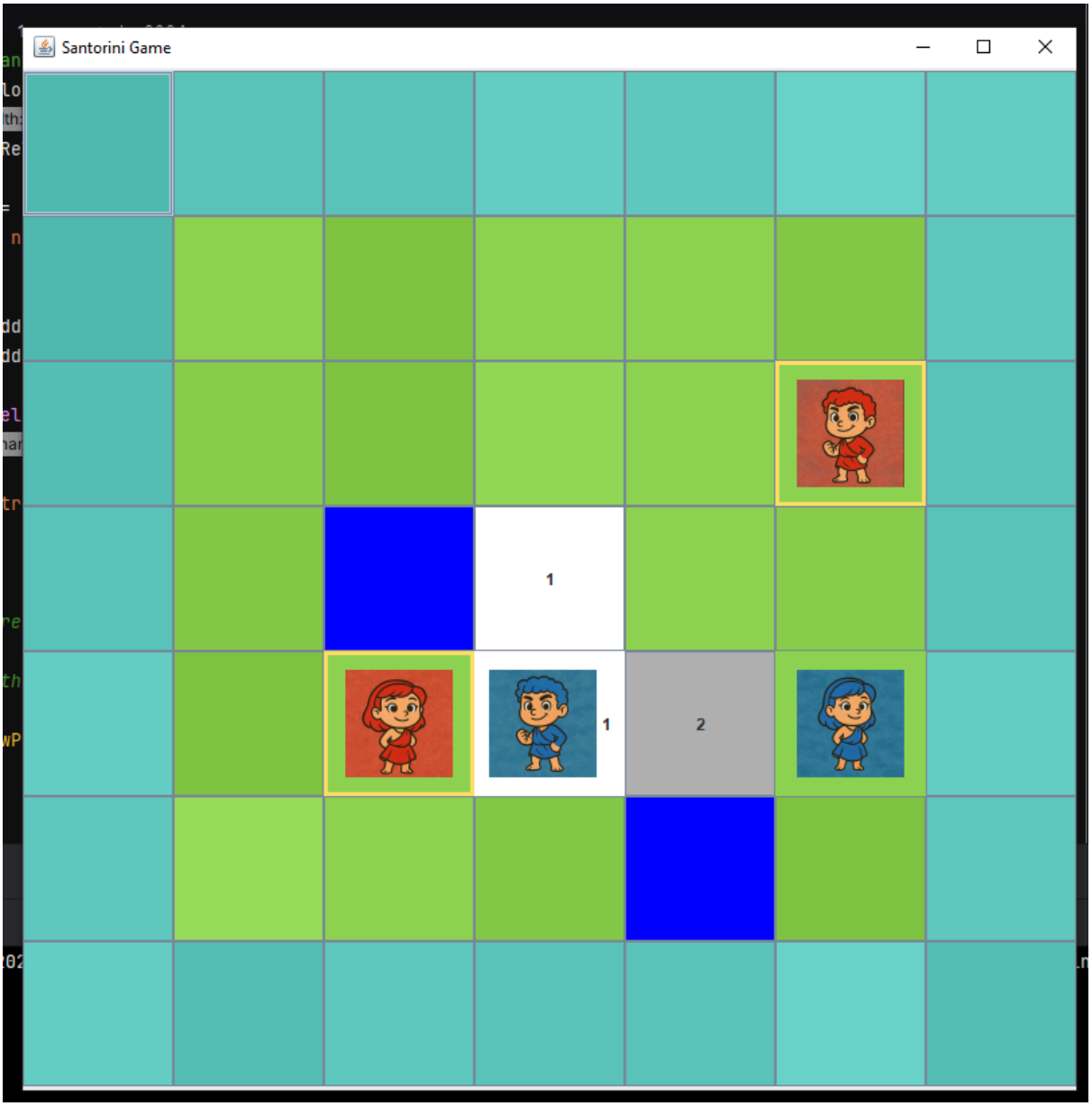

Santorini Game
2
1


Santorini Game
2
2


Santorini Game
2
2


Santorini Game
3
2


Santorini Game
Confirm Build
Would you like to build here?
Yes
No
1
2
3



Notes:
All working as expected

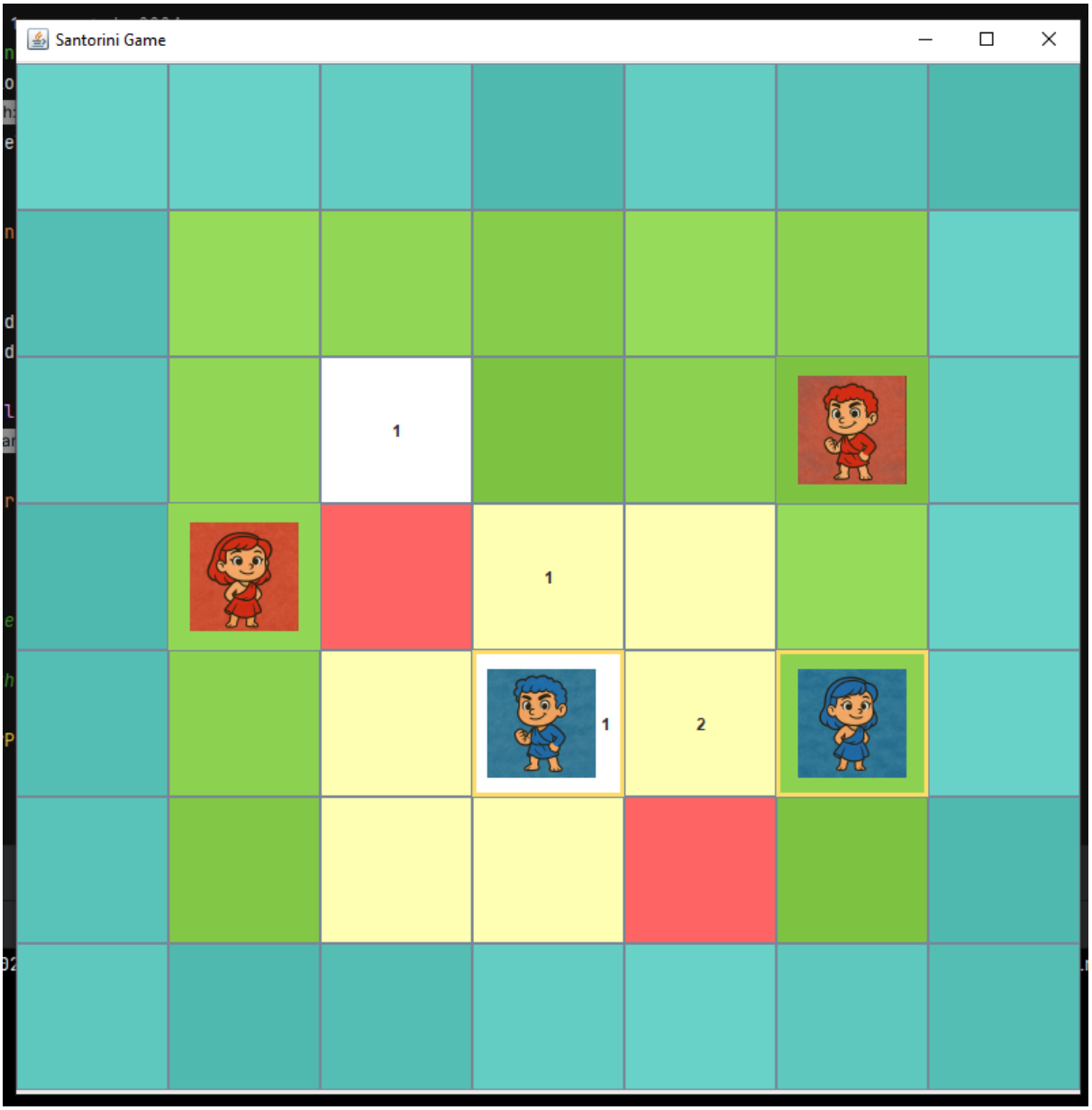
Change of Turn:

Expected outcome:
When a player places their final building, the game turns over to the next player.
Actual Outcome:


Santorini Game
Confirm Move
Would you like to move here?
Yes
No
1
2


Santorini Game
Confirm Build
Would you like to build here?
Yes
No
1
2


Santorini Game
New Turn
Now it's Player 2's turn!
Your God: Demeter
Select your Worker to move.
OK
1
2



Notes:
Player change working as expected.

Invalid Selections:

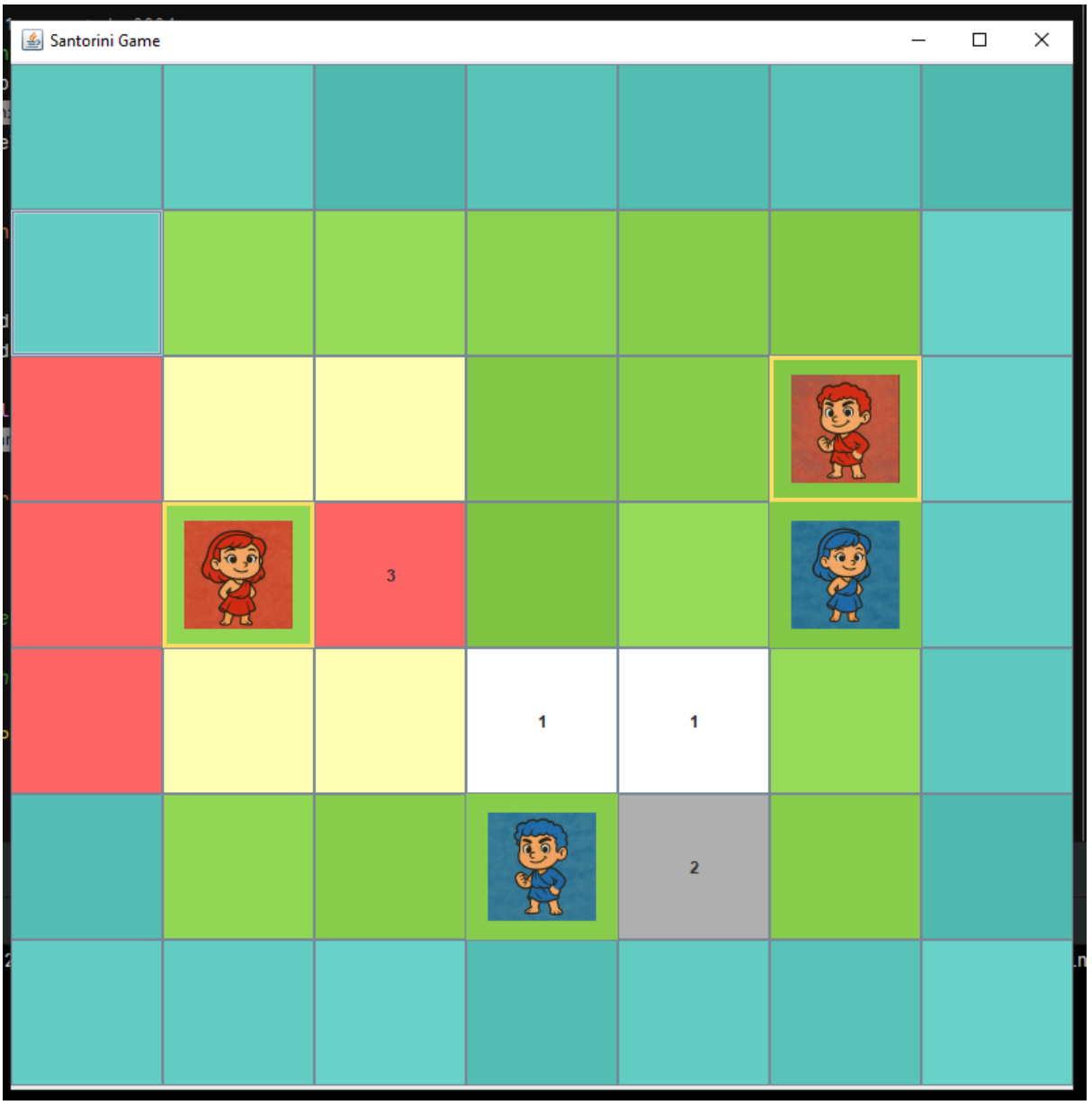
Moving onto worker:

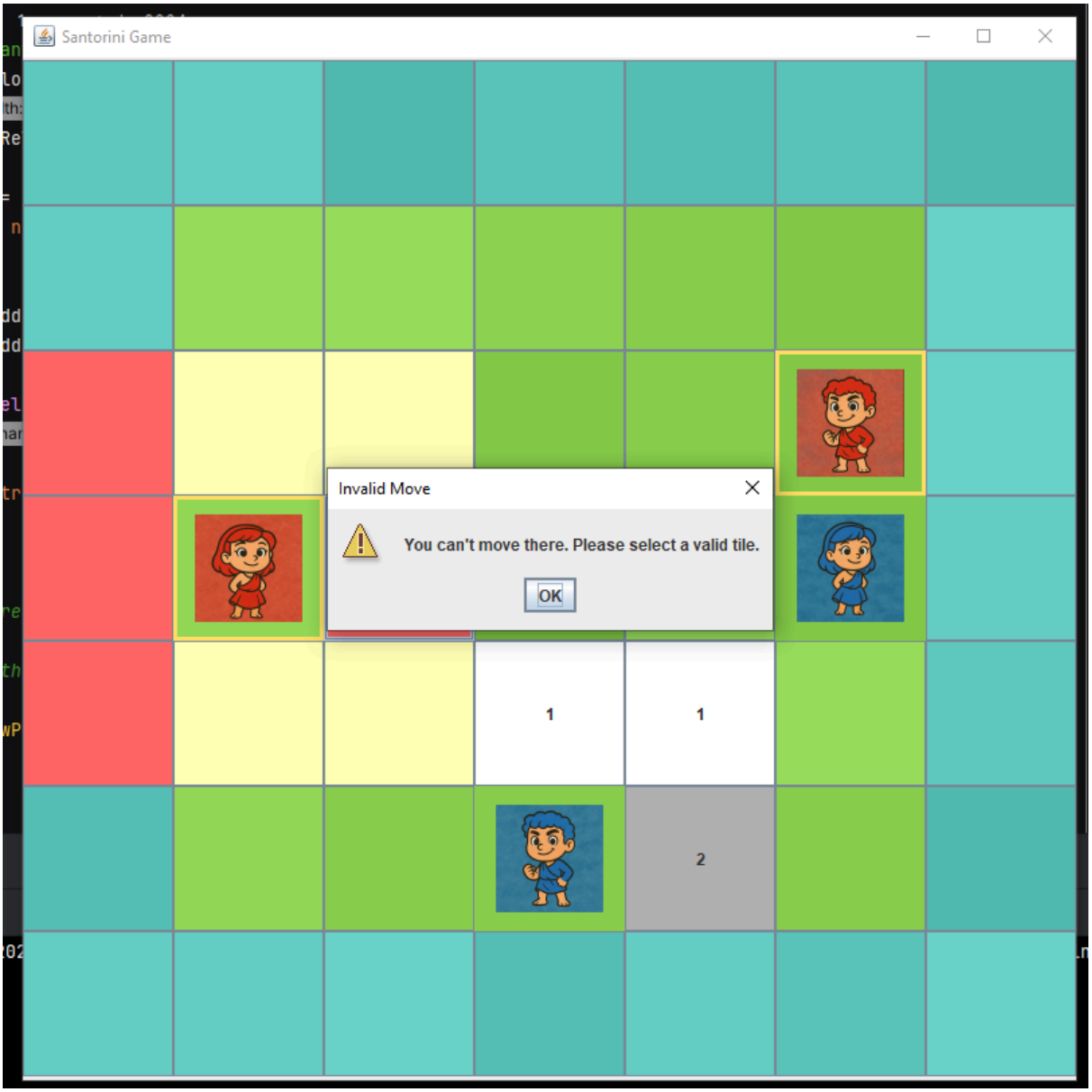
Expected outcome:
Actual Outcome:
Notes:

Moving onto too high of a tile:

Expected outcome:

Actual Outcome:

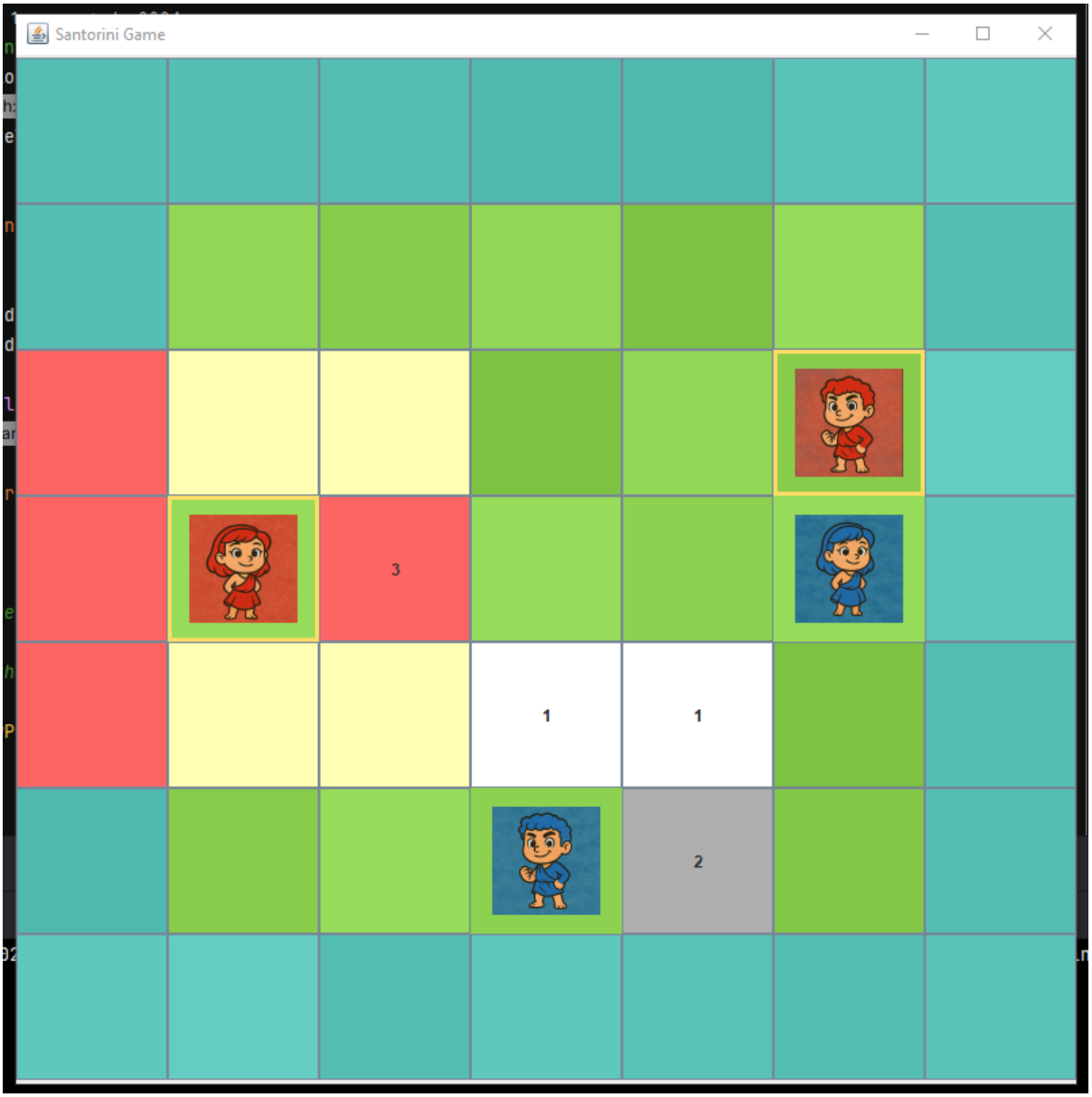


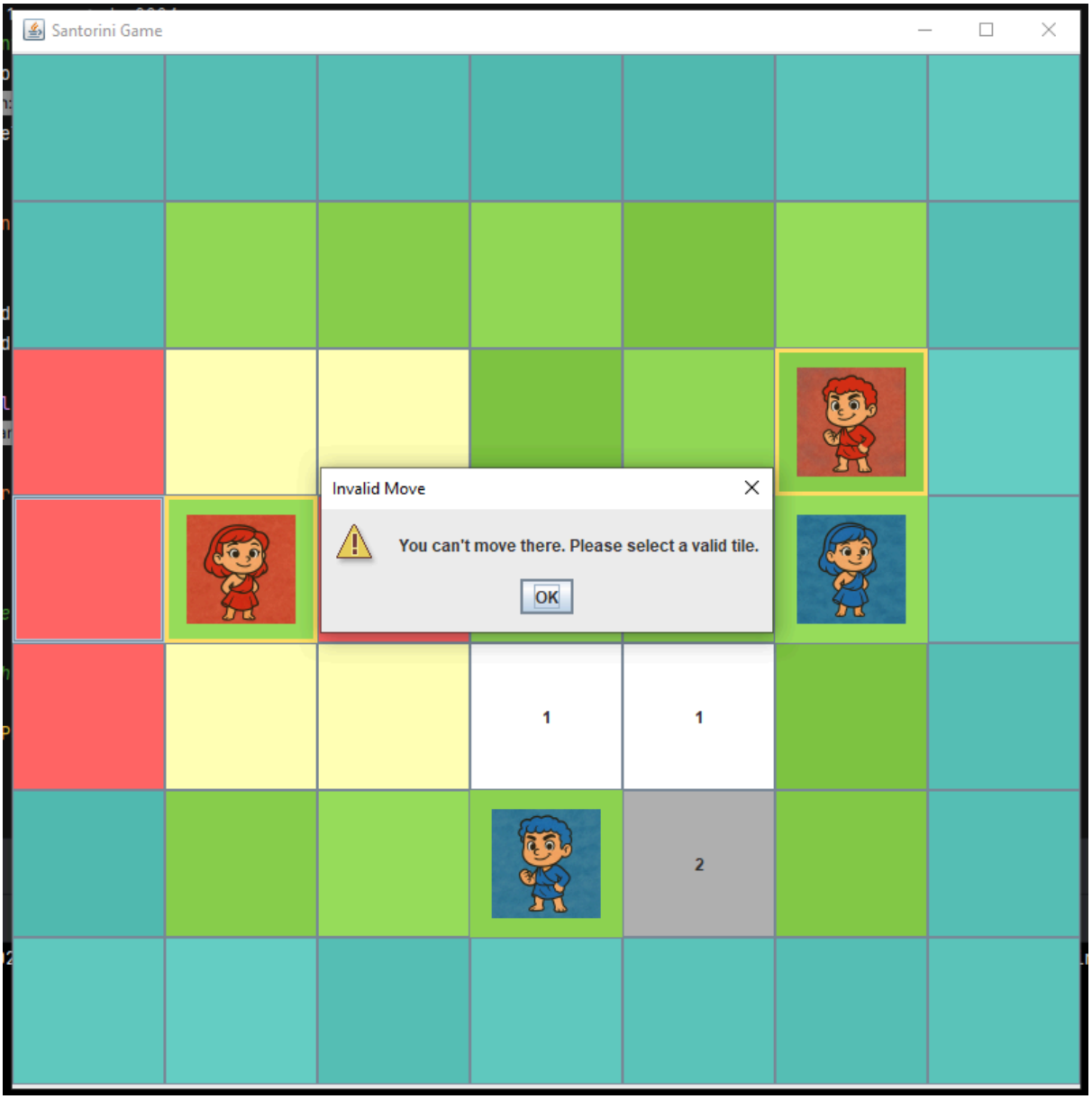


Notes:
Working as expected

Moving onto ocean:

Expected outcome:
Actual Outcome:

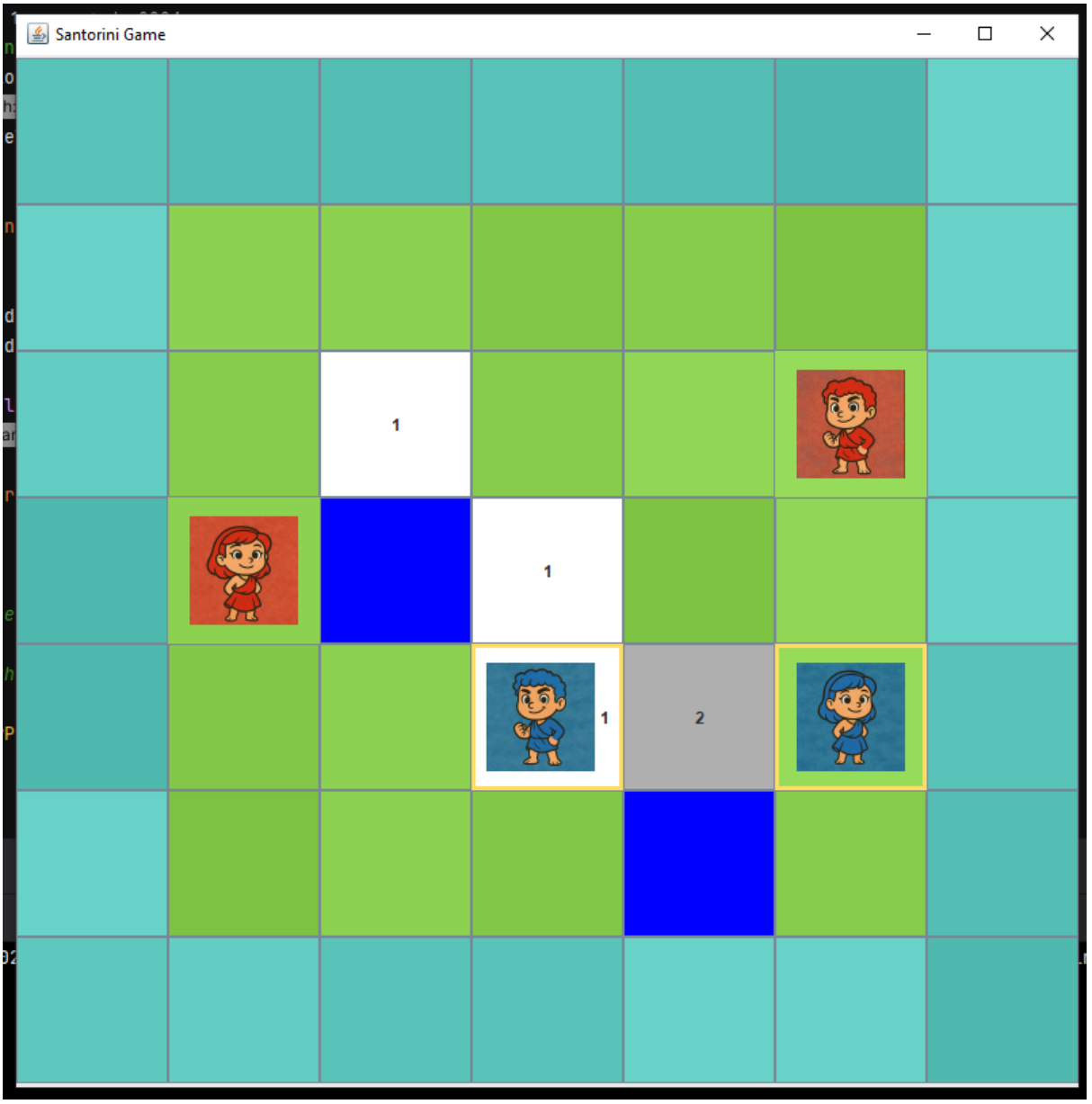

Santorini Game
3
1
1
2

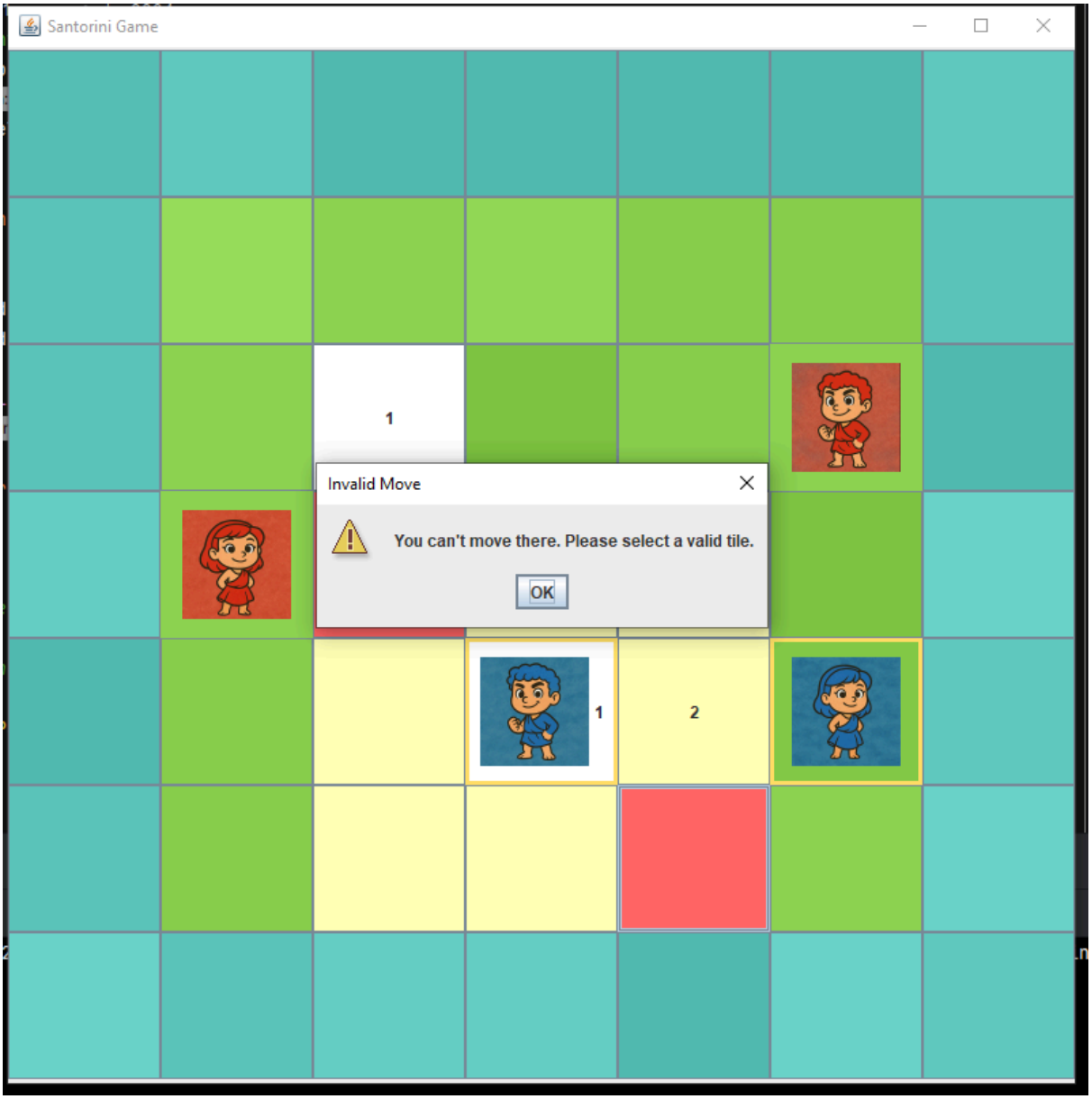


Notes:
Working as expected

Moving onto dome:

Expected outcome:
Player is not allowed to move worker onto a dome
Actual Outcome:

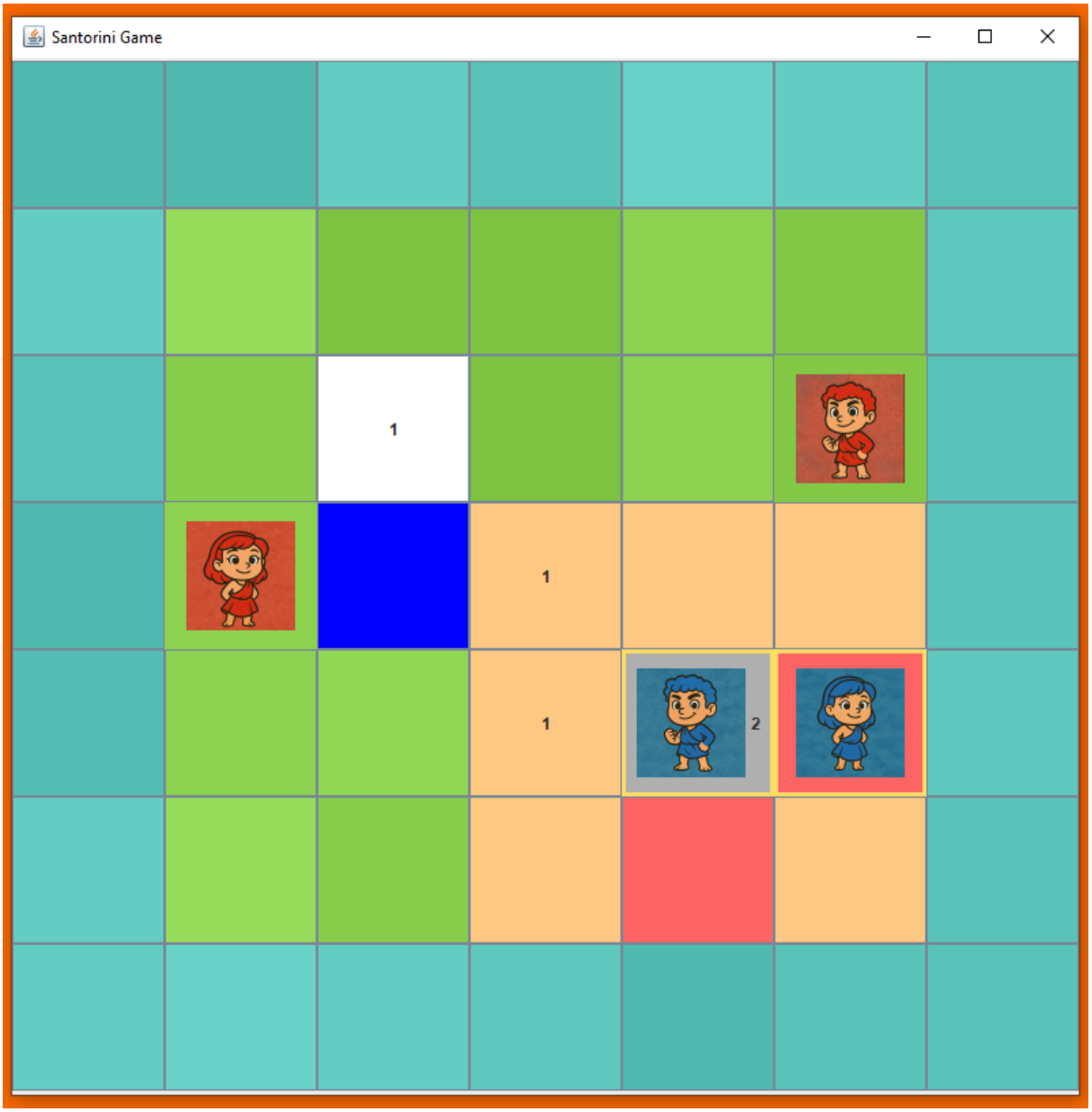

Santorini Game
1
1
1
2



Notes:
Working as expected

Building onto dome:

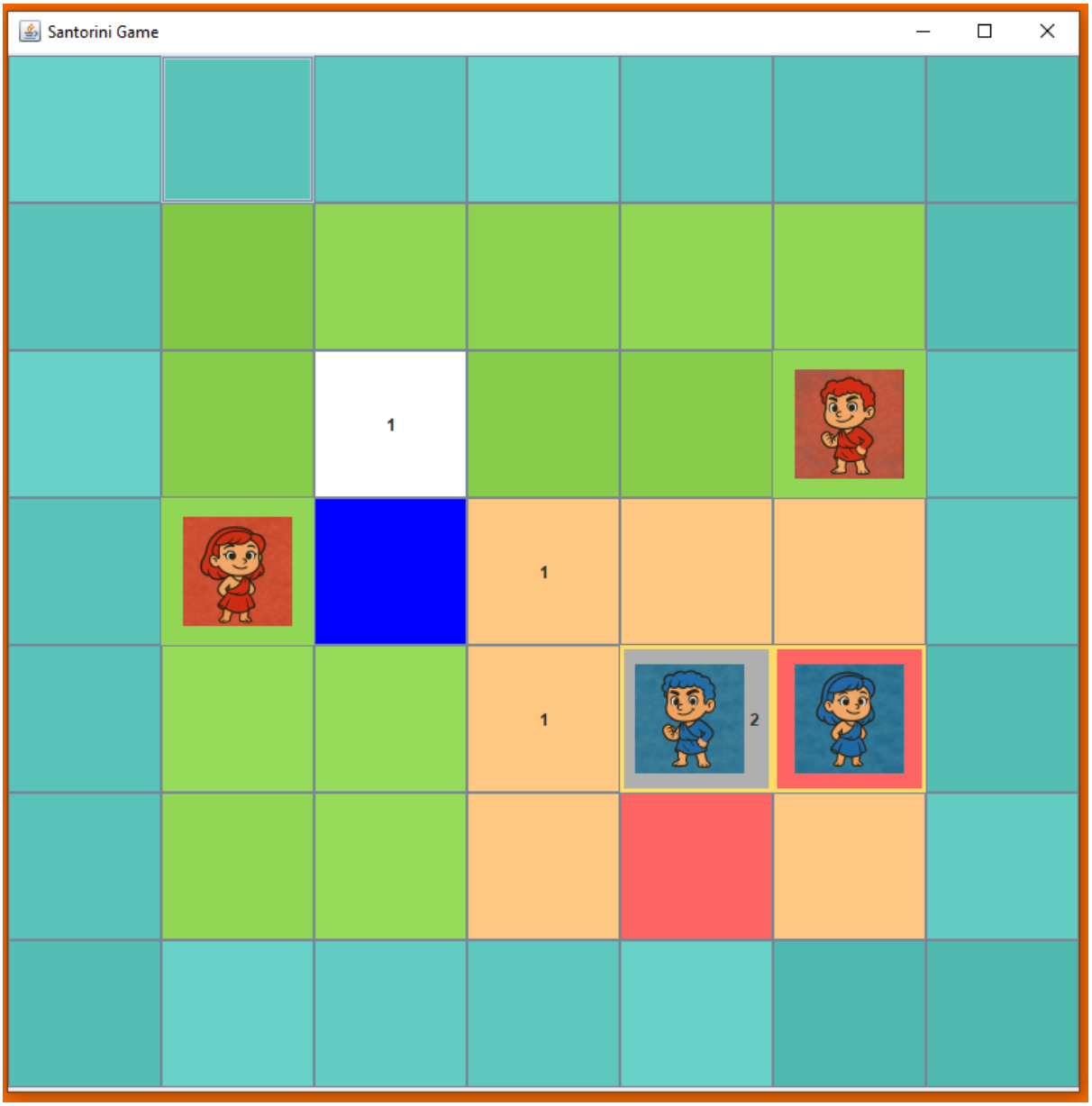
Expected outcome:
Player is not allowed to build ontop of dome
Actual Outcome:

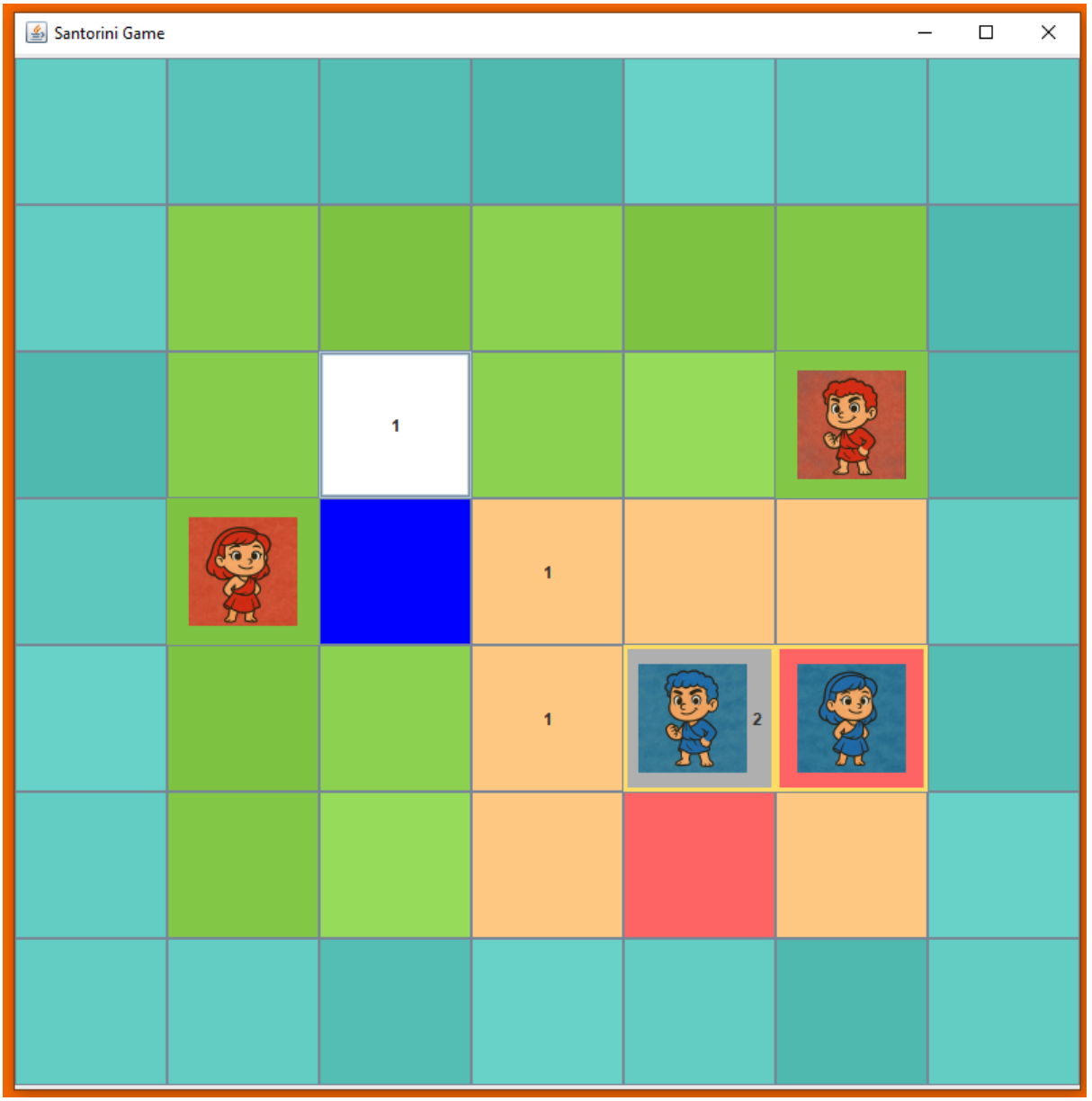

Santorini Game
1
1
1
2

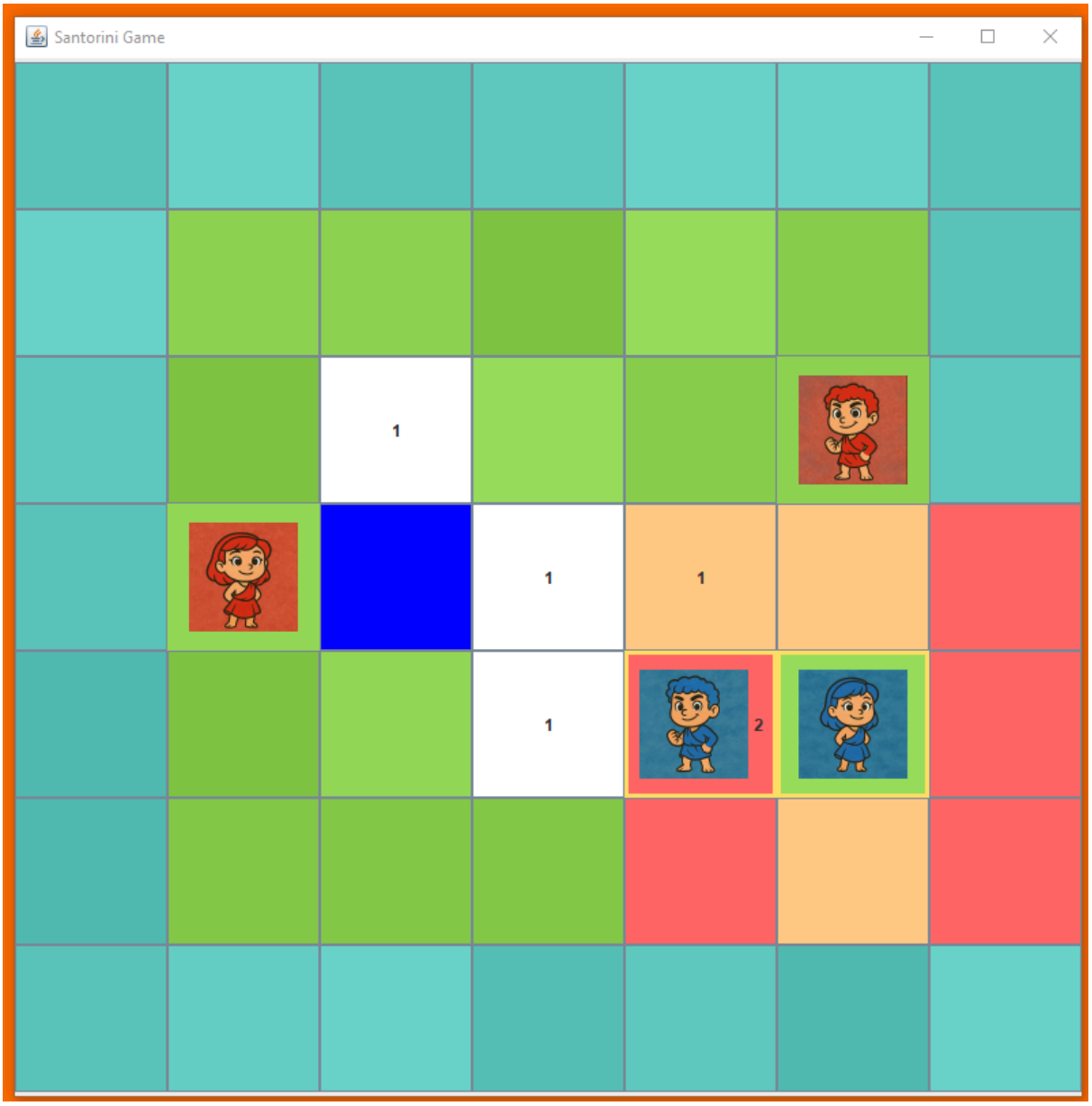
Notes:
Working as expected

Building onto worker:

Expected outcome:
Player cannot build on top of the worker.
Actual Outcome:


Santorini Game
1
1
1
2


Santorini Game
1
1
1
2



Notes:
Player is able to click on other workers that are adjacent, even if they haven't built. If the player then makes attempts to make a build action with an adjacent worker, it gets stored in that worker's history, meaning if you have demeter, you then build with that worker instead of the original.

Building onto ocean:

Expected outcome:
Building on top of ocean tile is invalid
Actual Outcome:

Notes:
Working as expected.

Winning:

Moving onto 3 tall tower:

Expected outcome:
This triggers the active players win condition, ending the game.
Actual Outcome:

Notes:
Working as expected.

God Power:

Demeter God Power:

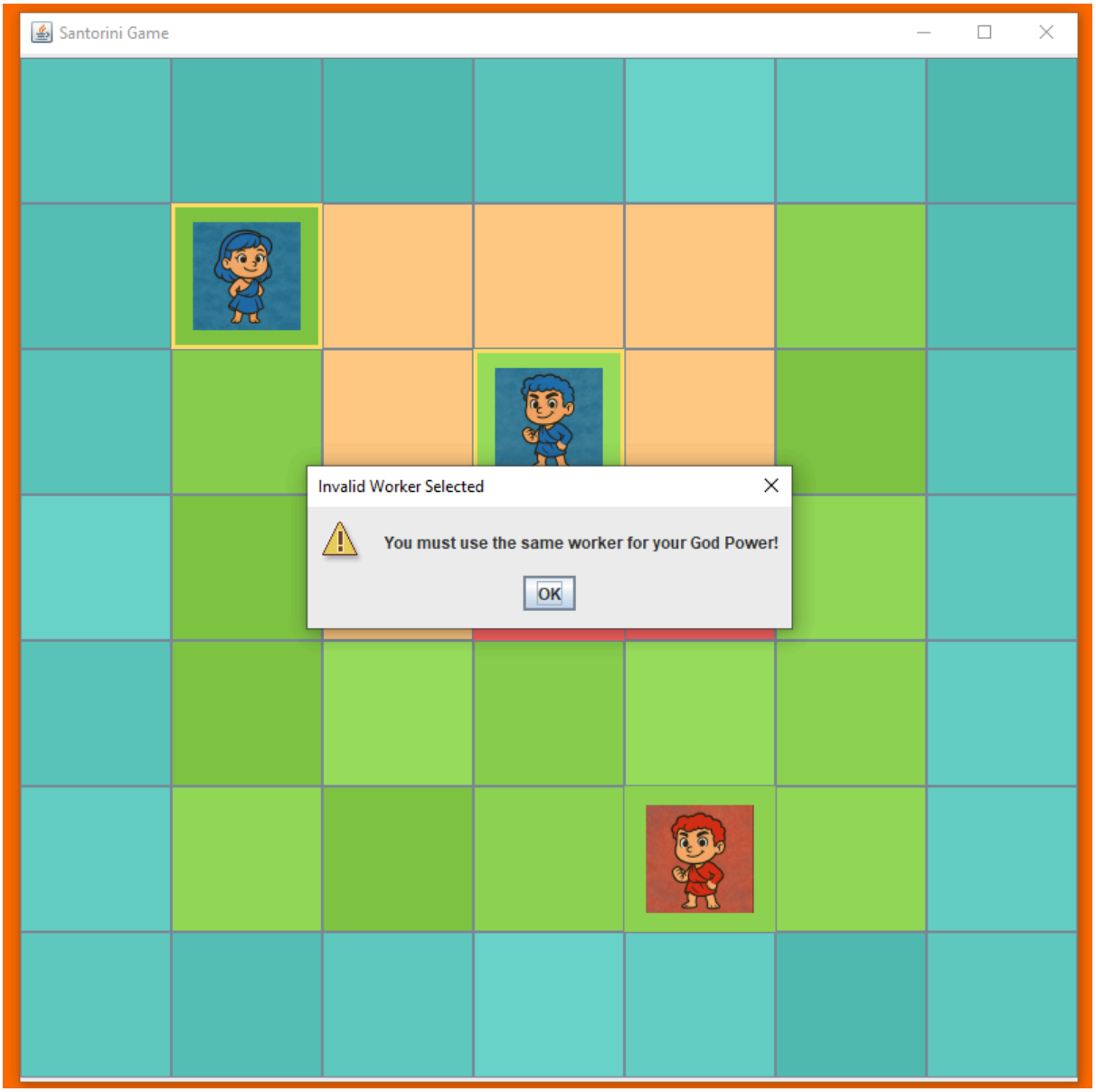
Expected outcome:
Can have the option of building a second time with the same worker at a different position.
Actual Outcome:


Santorini Game
1
Confirm Build
Would you like to build here?
Yes
No

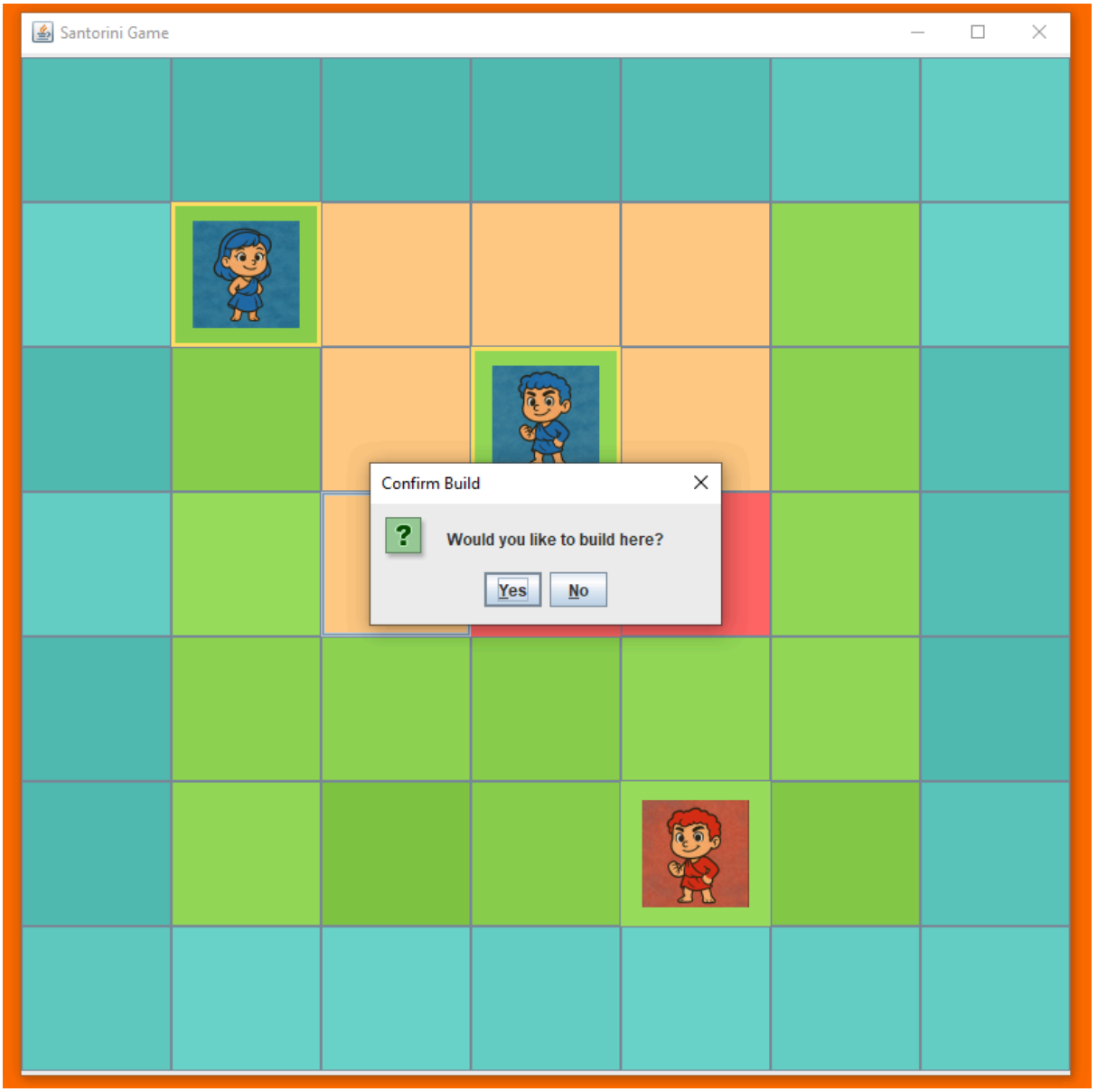

Santorini Game
Demeter Power
Do you want to use Demeter's power?
Yes
No

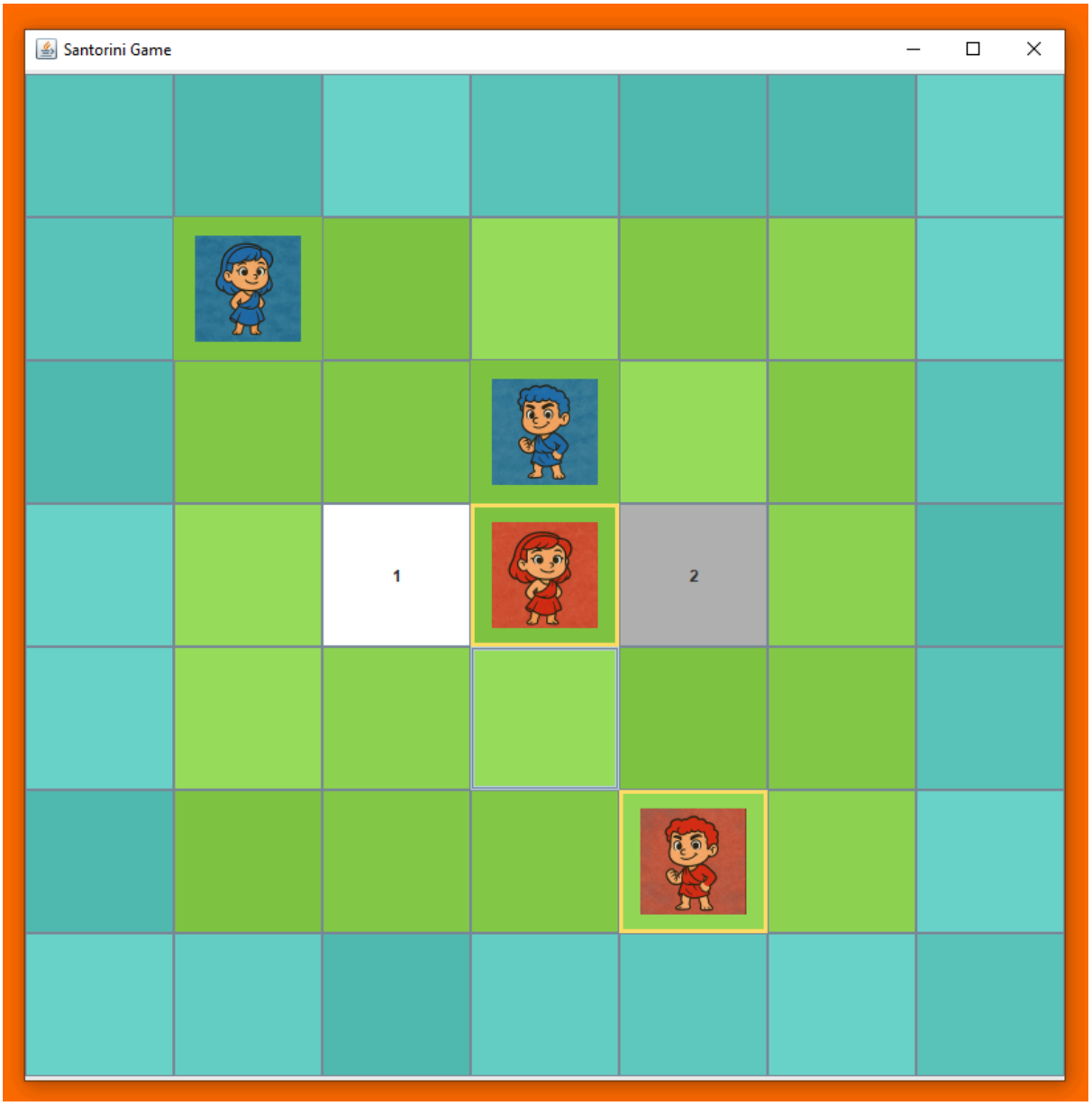

Santorini Game
Demeter's Power Activated
You can build twice this turn!
But not in the same space.
OK

(On attempting to build in same position twice)



(On clicking on other worker)


Santorini Game
Confirm Build
Would you like to build here?
Yes
No



Notes:
Working mostly as expected, but if demeters two workers are next to each other, the player can click on the other worker and make the build as them, but using the original workers surrounding tiles. Then the worker, for the second build, can place tiles around themselves.
Notes:
Working as expected.

God Power:

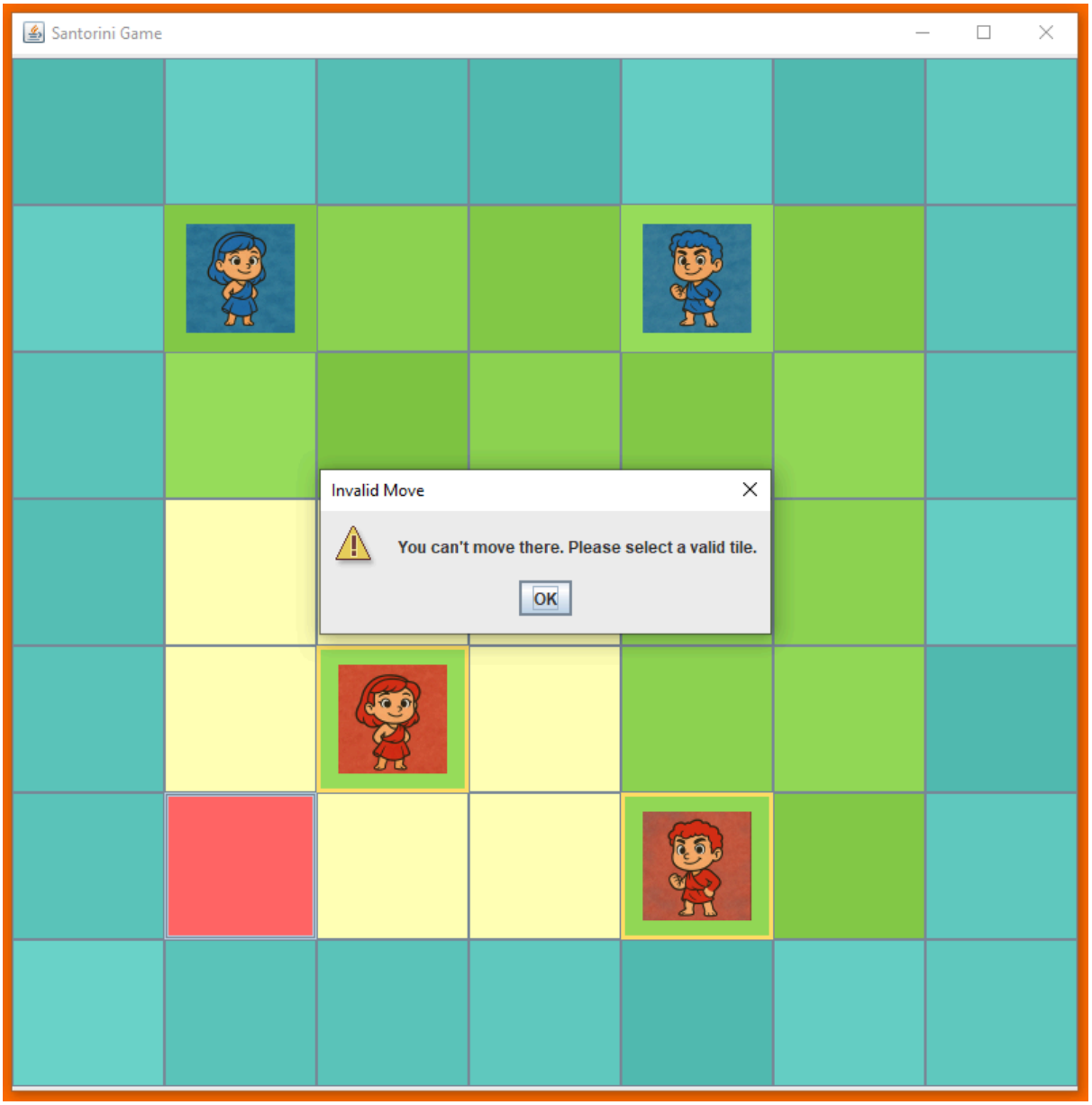
Artemis God Power:

Expected outcome:
Can move a second time, but gets blocked from returning where the worker came from.
Actual Outcome:


Santorini Game
Confirm Move
Would you like to move here?
Yes
No


Santorini Game
Artemis Power
Do you want to use Artemis's power?
Yes
No


Santorini Game
Artemis's Power Activated
You can move twice this turn!
But not back to the space you started from!
OK



(On trying to go back to prior position)

(On clicking other worker)


Santorini Game
Build Phase
Player 1, it's time to build!
OK