1. How can a player interact with the game? What are the possible actions?

My implementation assumes that the users will be switching off as p1 and p2 when playing the game through the GUI (order is p1, then p2, repeat).

The players may begin the game by clicking the New Game button. Each player can pick a god card (Demeter, Minotaur, Pan) or the default player. The players start by selecting 2 locations each, placing their workers. For each round, a player can select a location with one of their workers and select another location to attempt to move them. After a successful move action, the game checks if the player has won. If so, the game is over, and the players cannot click the board to make actions anymore. If not, the player can then select a location to attempt building a tower/dome. After a successful build action, the game will switch it to the other player's turn.

The 3 things a user can do are click the New Game button, click a god card or player, and click a location on the board. The actual flow of the game is explained to the user through instructions displayed on the top bar.

The users' defined actions exemplify the design principle of a low representational gap, since the actions are exactly what the game instructions allow the user to do. As placing workers, moving, and building all require selecting a location, it's very clear to the user how to play the game with the GUI. Therefore, this accomplishes the design goal of understandability and change. If there is user input error (out-of-bounds location, etc), the method will not execute any actions or change the state, instead returning with an error message displayed on the top bar. This way, the user can try the action again. All of this accomplishes good abstraction, so the user does not need to know the details of the program to play the game.

2. What state does the game need to store? Where is it stored? Include the necessary parts of an object model to support your answer.

The game state contains the board grid, winner, current player's turn, and instruction to display. This information is managed in GameState class, and it's fetched from the Game object.

The Game class has attributes hasGameEnded, playerTurn, and nextAction to track the game status, current player, and next game action. It also stores the board grid and the two players. Additionally, a game object stores the selected location and instruction to display to the GUI. The Grid class has attributes gridState and occupiedFields. The grid itself is represented by a 2d matrix of tower objects. For the Tower class, each tower has 1 location object (row and col of the grid coordinates) and also attributes for its level and if it has a dome. For efficiency, the Grid class also has attribute occupiedFields which represents locations that workers cannot move to. This includes locations of towers with domes and locations that workers are currently occupying. The Player class has an attribute for the player's id. Each player stores their Worker objects in a java Map, where they can also be accessed by their ID. For ease, a player stores all their worker objects' positions (Location objects) in a java Map as well. The Worker class also has an attribute for the worker's id, and the worker objects store their current position.

The game tracks which player's turn it is with the attribute playerTurn. First, the winner is default set to none in the game state. When the game calls checkWin() with the current player after a successful move action, it changes the attribute hasGameEnded to true if the winning condition is met. To get the winner, the game gets the current playerTurn when hasGameEnded is true. The grid and instruction to display on the GUI is stored in the game. As the user executes actions, this information is updated to the game state.

The Game, Grid, Tower, Location, Player, and Worker classes are separated to use the design principle of high cohesion and to accomplish design goals of understandability, reuse, and change. Generally, this separation is more understandable, and if any class was changed to add features, those changes would not have drastic effects on other classes.

I made the design choice for each player to contain a Worker Map (like a dictionary), for the design goal of extensibility. This way, implementing more than 2 workers in the future is much easier.

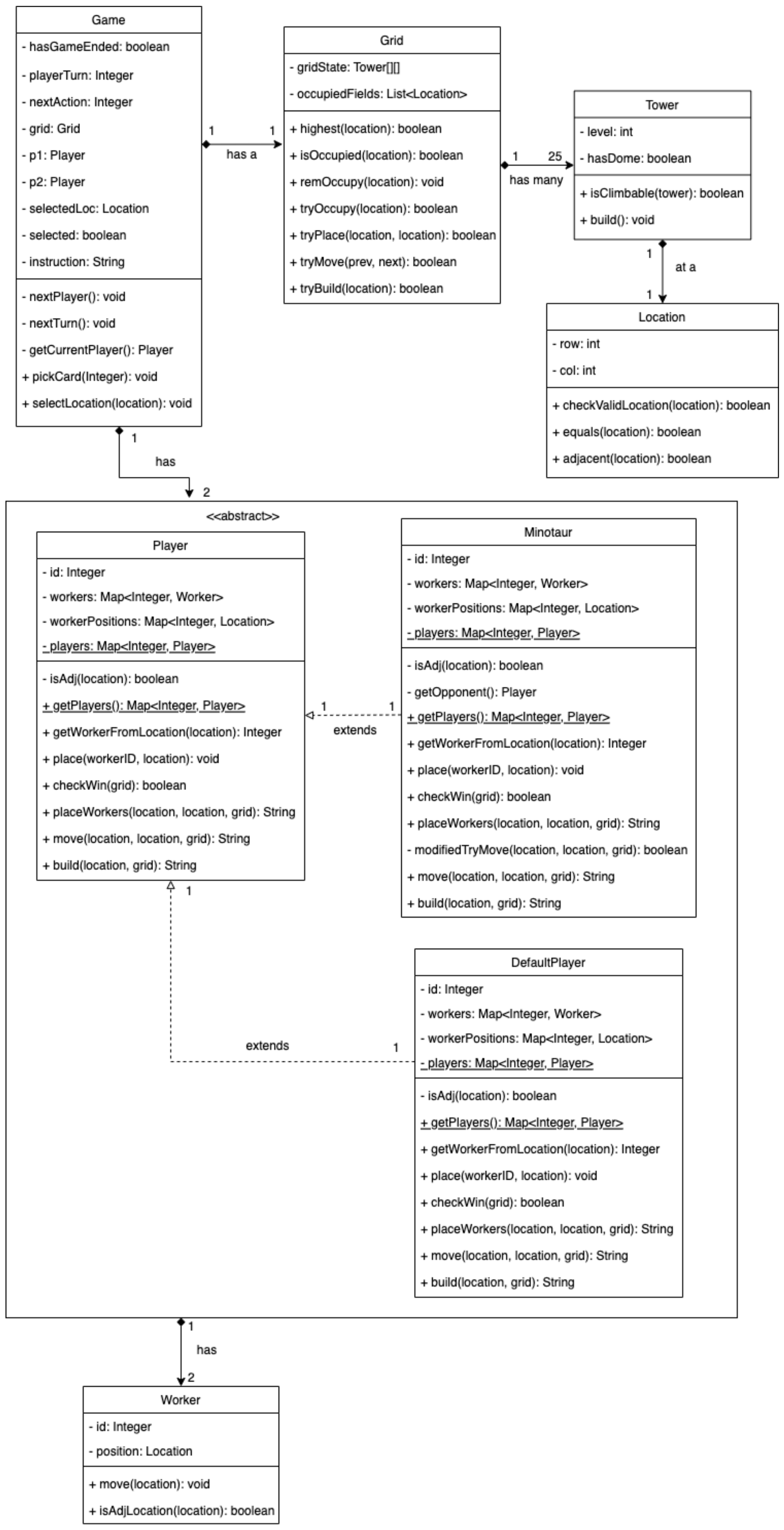
Additionally, I chose to design occupiedFields as one attribute instead of splitting into separate domeFields and workerFields, since they have similar functionality where a worker cannot move to that field. This design choice was made for the design goal of understandability and reuse, since there will be less attributes cluttering the Grid class and the isOccupied function can be used in many different contexts for the game.

Compared to the domain model, there is no Dome class in the object model. This was a decision since the dome is not a significant enough concept to warrant an extra class. Instead, domes are implemented as fields for the Tower object.

Although the Location and Tower objects are closely linked, they were implemented as separate objects because locations can be used by most other classes, while towers are mainly only used in the Grid class. This is also more intuitive for the user, which accomplishes design for understandability.

Compared to a previous iteration, all of the player action methods were moved from the Game class to the Player class. This is a design decision for understandability since now each player executes actions, which is more intuitive. In addition, having the Player class contain the important methods allowed me to implement God Cards more easily, so this also accomplishes the design goal of reuse and extensibility. The tradeoff was an increase in coupling, since the grid needed to be passed in to the Player objects and could be altered by the players. However, since this was the most efficient and intuitive way to implement God Cards, I believe this design decision is justified by the design goals it accomplishes.

Object Model parts (All Object Classes except Demeter and Pan)


Game
- hasGameEnded: boolean
- playerTurn: Integer
- nextAction: Integer
- grid: Grid
- p1: Player
- p2: Player
- selectedLoc: Location
- selected: boolean
- instruction: String
- nextPlayer(): void
- nextTurn(): void
- getCurrentPlayer(): Player
+ pickCard(Integer): void
+ selectLocation(location): void
1
has a
1
Grid
- gridState: Tower[][]
- occupiedFields: List<Location>
+ highest(location): boolean
+ isOccupied(location): boolean
+ remOccupy(location): void
+ tryOccupy(location): boolean
+ tryPlace(location, location): boolean
+ tryMove(prev, next): boolean
+ tryBuild(location): boolean
1
25
has many
Tower
- level: int
- hasDome: boolean
+ isClimbable(tower): boolean
+ build(): void
1
at a
1
Location
- row: int
- col: int
+ checkValidLocation(location): boolean
+ equals(location): boolean
+ adjacent(location): boolean
1
has
2
<<abstract>>
Player
- id: Integer
- workers: Map<Integer, Worker>
- workerPositions: Map<Integer, Location>
- players: Map<Integer, Player>
- isAdj(location): boolean
+ getPlayers(): Map<Integer, Player>
+ getWorkerFromLocation(location): Integer
+ place(workerID, location): void
+ checkWin(grid): boolean
+ placeWorkers(location, location, grid): String
+ move(location, location, grid): String
+ build(location, grid): String
1
1
extends
Minotaur
- id: Integer
- workers: Map<Integer, Worker>
- workerPositions: Map<Integer, Location>
- players: Map<Integer, Player>
- isAdj(location): boolean
- getOpponent(): Player
+ getPlayers(): Map<Integer, Player>
+ getWorkerFromLocation(location): Integer
+ place(workerID, location): void
+ checkWin(grid): boolean
+ placeWorkers(location, location, grid): String
- modifiedTryMove(location, location, grid): boolean
+ move(location, location, grid): String
+ build(location, grid): String
1
extends
1
DefaultPlayer
- id: Integer
- workers: Map<Integer, Worker>
- workerPositions: Map<Integer, Location>
- players: Map<Integer, Player>
- isAdj(location): boolean
+ getPlayers(): Map<Integer, Player>
+ getWorkerFromLocation(location): Integer
+ place(workerID, location): void
+ checkWin(grid): boolean
+ placeWorkers(location, location, grid): String
+ move(location, location, grid): String
+ build(location, grid): String
1
has
2
Worker
- id: Integer
- position: Location
+ move(location): void
+ isAdjLocation(location): boolean

3. How does the game determine what is a valid build (either a normal block or a dome) and how does the game perform the build? Include the necessary parts of an object-level interaction diagram (using planned method names and calls) to support your answer.
assuming that the active player has the Demeter card (and the opposing player has no card).

In the Demeter class' build method, boolean firstBuild is checked to see if the user is performing the first build, and since firstBuild is true, the original build method is called.

Original Build Method
First, it checks if the location is invalid (out-of-bounds). If so, the build method will return an error message.

To determine a valid build, the original Player method isAdj(location) is called to determine if location is an adjacent location to either worker of the player. If not, the method returns an error message. Otherwise, the game calls the Grid method tryBuild(location). The Grid object's tryBuild method uses isOccupied(location) to check if the location is not occupied. Occupied means the location does not currently have a domed tower or a worker currently occupying it. If it's occupied, the tryBuild will return false.

Otherwise, the build will be performed. The Tower method build() is called, which increases the level of the tower at that location by 1. The exception is when the level is already 3; in that case, the tower's hasDome attribute is turned to true. Afterwards, the tryBuild method checks if a dome has been added to the tower with tower method isDomed(), and if so, it adds the location to occupiedFields since it is now occupied.

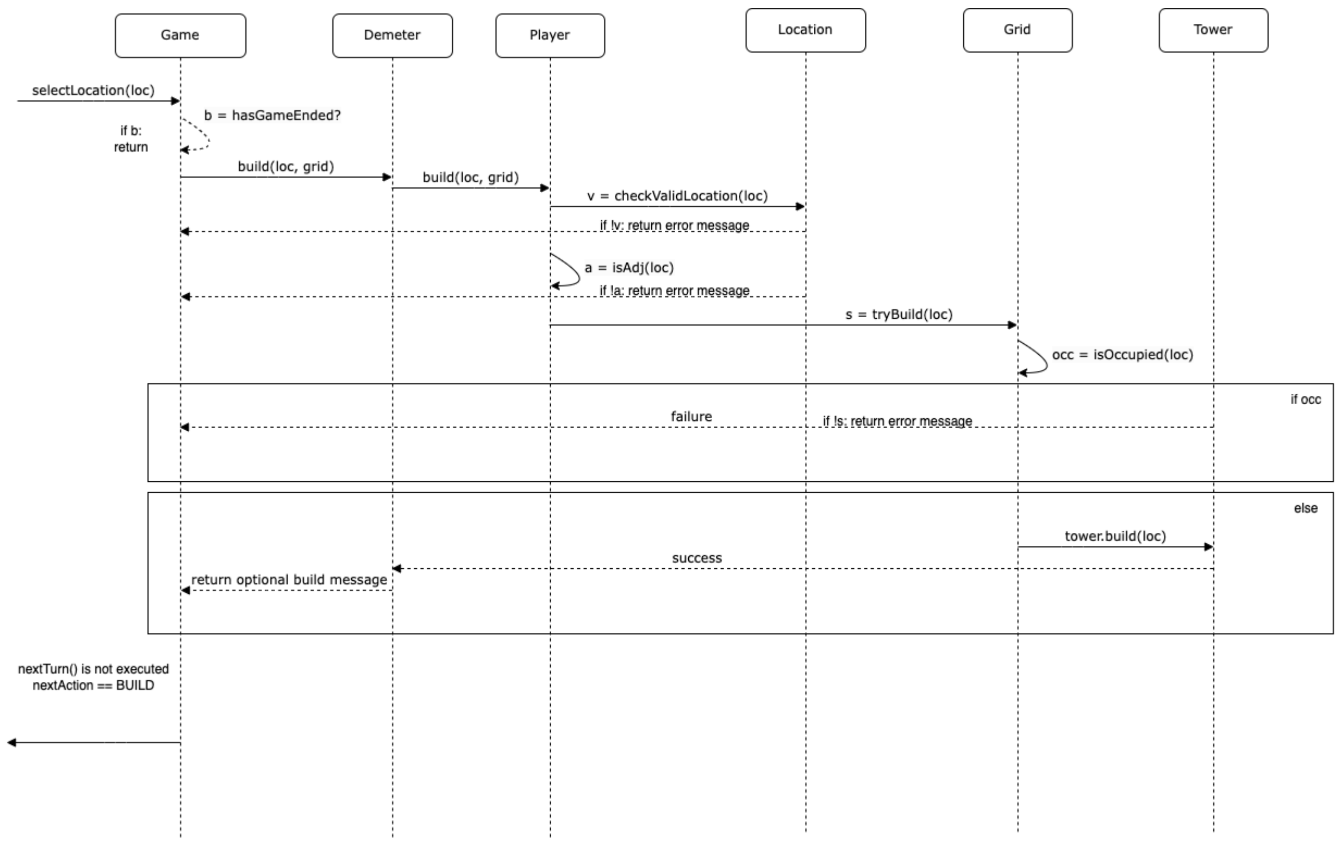
If tryBuild(location) returns false, the method returns an error message. Otherwise, the build was successful, and the method will return a success message.

Back in the Demeter's build method, the success (or error message) is stored in the attribute instruction. Additionally, firstBuild is set to false, and the build location is stored. Then, Demeter's build method returns a message that states the user is given the option to build again. If the user chooses to build again, Demeter's build method checks that it's the second build action with firstBuild. If the build location is a location occupied by one of the player's workers, it skips the second build and returns a success message. If not, then, it checks if the build location is the same as the previous build location, and if so, returns an error message. Otherwise, it'll attempt to perform the second build (calling the original build method again) and return the result.

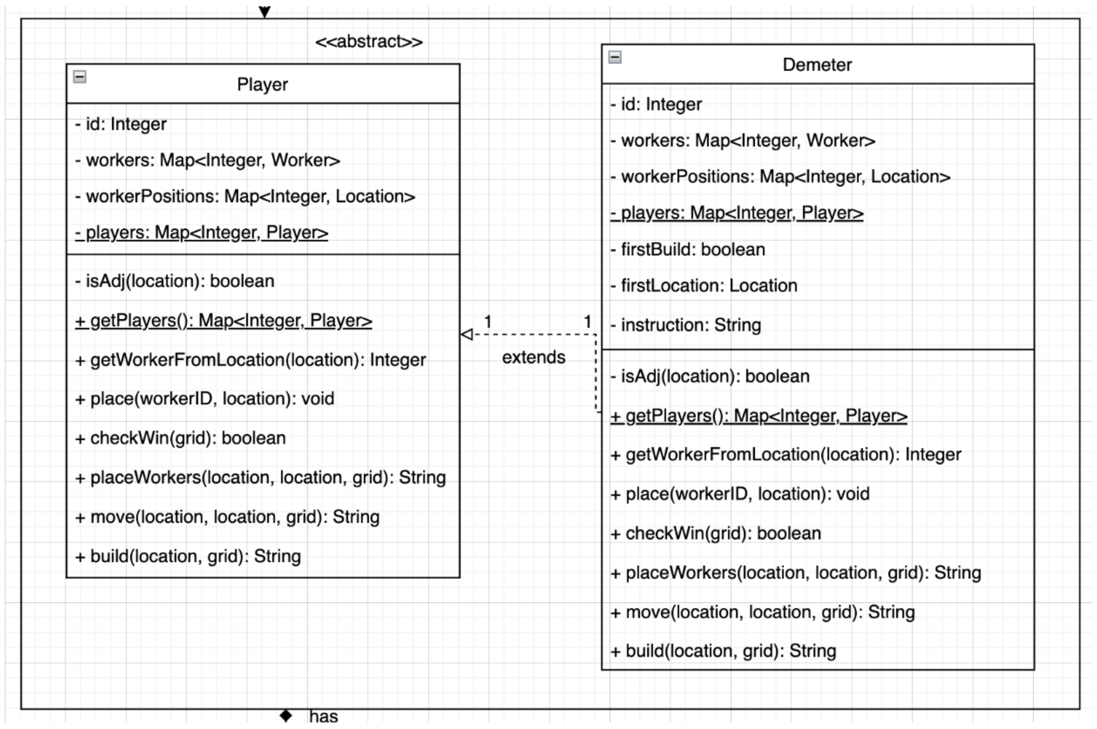
I decided to incorporate the dome into the tower class, since the dome is part of the tower. Therefore, the design goal of understandability is accomplished because keeping them together in the same class is more intuitive. In addition, I decided to put the build method in the Tower class, since this incorporates the design principle of low coupling, as building a block/dome does not depend on attributes in other classes such as Grid, Location, or Player. The other necessary methods (location.checkValidLocation(), grid.tryBuild(), grid.isOccupied(), tower.isDomed(), etc) were implemented in their respective classes for the same reason and because they're in the class most suited for their purpose. This accomplishes the design goals of reuse and change with a more modular approach.

My implementation of the Demeter build method makes use of the game's way of executing actions. If the game doesn't get a String return value of "success", the interpretation is that an error String was returned, and the user can repeat the same action with no extra external effects. Thus, my Demeter's build method returns a new String indicating an optional build, which makes my game change the execution of game flow to allow the user to build again. This way of implementing build is much easier (and more clever I'd say), since it doesn't require any more coupling of the Game and Player classes. If I implemented it in a more naive way, for example having a flag in the Game class for indicating how many times a user can perform an action, it'd increase coupling and give more power to the Player class, which can decrease cohesion and eventually make the Player class like a God class.

<u>Object-level Interaction Diagram (only first build)</u>



Object Model

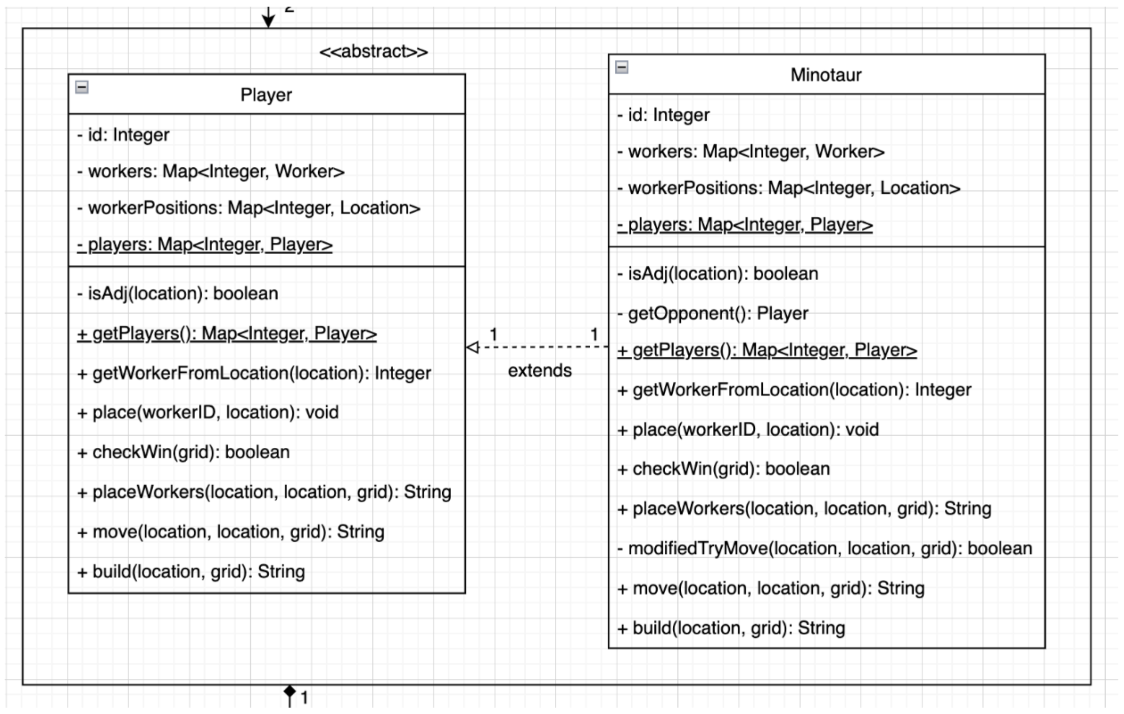


<<abstract>>
Player
- id: Integer
- workers: Map<Integer, Worker>
- workerPositions: Map<Integer, Location>
- players: Map<Integer, Player>
- isAdj(location): boolean
+ getPlayers(): Map<Integer, Player>
+ getWorkerFromLocation(location): Integer
+ place(workerID, location): void
+ checkWin(grid): boolean
+ placeWorkers(location, location, grid): String
+ move(location, location, grid): String
+ build(location, grid): String
1
1
extends
Demeter
- id: Integer
- workers: Map<Integer, Worker>
- workerPositions: Map<Integer, Location>
- players: Map<Integer, Player>
- firstBuild: boolean
- firstLocation: Location
- instruction: String
- isAdj(location): boolean
+ getPlayers(): Map<Integer, Player>
+ getWorkerFromLocation(location): Integer
+ place(workerID, location): void
+ checkWin(grid): boolean
+ placeWorkers(location, location, grid): String
+ move(location, location, grid): String
+ build(location, grid): String
has

4. How does your design help solve the extensibility issue of including god cards? Please write a paragraph (including considered alternatives and using the course's design vocabulary) and embed an updated object model (only Minotaur is sufficient) with the relevant objects to illustrate.

My design has the Player class implemented as an abstract class. In addition, since most of the game functionality is contained in the Player class (placeWorkers, move, build, checkWin), it's easier to make changes by overriding methods based on the abilities of certain god cards. Other options I considered were making a God Card class that can modify aspects of the game and making a God Card class that served as a wrapper for the whole game. The former option proved too difficult to implement, since the abilities of god cards are extremely varied meaning there would have to be a different control option for each god card. This would be working against the design goal of extensibility. The latter option would require giving this God Card class too much power and work, which means a large increase in coupling and very low cohesion.

Object Model



2
<<abstract>>
Player
- id: Integer
- workers: Map<Integer, Worker>
- workerPositions: Map<Integer, Location>
- players: Map<Integer, Player>
- isAdj(location): boolean
+ getPlayers(): Map<Integer, Player>
+ getWorkerFromLocation(location): Integer
+ place(workerID, location): void
+ checkWin(grid): boolean
+ placeWorkers(location, location, grid): String
+ move(location, location, grid): String
+ build(location, grid): String
1
1
extends
Minotaur
- id: Integer
- workers: Map<Integer, Worker>
- workerPositions: Map<Integer, Location>
- players: Map<Integer, Player>
- isAdj(location): boolean
- getOpponent(): Player
+ getPlayers(): Map<Integer, Player>
+ getWorkerFromLocation(location): Integer
+ place(workerID, location): void
+ checkWin(grid): boolean
+ placeWorkers(location, location, grid): String
- modifiedTryMove(location, location, grid): boolean
+ move(location, location, grid): String
+ build(location, grid): String
1

5. What design pattern(s) did you use in your application and why did you use them? If you didn't use any design pattern, why not? Note that you need to update your answers to the questions of Task3 from HW3.

I used the template method design pattern, by having an abstract Player class and extending God Card classes from the original abstract class. Many of the god card functions were purely additive, and as stated in my Q4 answer, my design had most of the core game functionality in the Player class. In addition, I still needed some side effects from methods in the original abstract implementation, which was made easier through inheritance. Due to these reasons, I decided that inheritance and the template method would be most efficient. Implementing this way accomplishes the design goals of reuse and extensibility, especially since it allowed me to reuse the code from the original abstract class.