The object model was largely based on the domain model made in part 1, with some modifications. The objects were retained from the concepts of the domain model. All the cardinalities were kept the same, as changing project abstraction does not change the numerical relationship between the objects.

The first addition in the object model was giving the Board object to determine the game state, which will be passed into the Game object. This was done to assist in Demeter's Law- we do not want the Game interacting with the Worker or Field(Tile) objects, due to their two degrees of separation.

Next is giving the Player an attribute that shows what Workers he/she uses instead of assigning player_id to each individual worker. This is because the Player winning or losing is related to the game state, while the Worker object should only be worried about the ability to move and build.

Additionally, the precondition methods to ensure a tile can be moved to or built on was moved to the Field Object, as the Worker should not have to worry about that. In the actual implementation, the Block or Dome objects may be moot, since I can just add a flag that makes that tower non-scalable.

The method calls were decided based on the interaction diagrams made in steps 2 and 4. The loops and return values in step 4 were made in accordance with the precondition and postcondition specs documented in step 3.

One method noted in the object model is validBoard(), which is just reworded from gameState(). Others include: ifValidTile(), getField() or getTile(), and tower addLayer().

Additional changes were made when moving from modeling to creating the actual implementation. To compromise between cohesion and unnecessary coupling, I have relegated to NOT put a tower in a separate class, as its only attribute is having a height and being built. Those things can just be done in the Tile class. The Board class also does not have a lot of attributes and methods, but it is still needed to keep track of all of the tiles and provides a way of measuring the game state. To alleviate over-coupling, game state checks and reset methods were done in the Tile and Worker classes as a basis. When checking bigger states, the Board and Player classes, respectively, would call upon the class with the basic implementation. And lastly, the Game class holds the ultimate game state checking methods.

As mentioned before, the state is called in the Game class, but its implementation relies on calls to other classes it has cardinality with, since the Game should not be immediately responsible for knowing the attributes of Player, Board, Worker, and Tile. Valid checks are housed in the Tile class, but are called at some points in the Worker class before relocation and building- so the Worker class performs the action update every turn.

Homework 5 Extensions

1. Extensibility

A lot of refactoring was done prior to the implementation of the two extensions for the sake of extensibility. The biggest change was separating out the game logic (or any method that can potentially be modified by the god cards) into its own game logic class. Originally, operations such as build, move, etc would be in the Tile and Player classes. As an intermediate solution, I placed the game logic into the Game object as opposed to the Tile, Player, and Board classes- but that still was not extensible when it came time to add the god cards. The game logic can access logic from the Game and Player classes.

Second, a lot of work was done to minimize unwanted coupling. The Tile class is isolated and only talks to the Board class, which only talks to the Game class. The worker class only talks to the Player class, which shares logic with the Game and GameLogic classes. Since the game state is now separated from the game logic (and the mutable fields of the gamestate are separate from each other), if I need to add new methods in the logic, I do not need to change anything in the game state- meaning Tile, Worker, Board, etc.

2. Design Patterns

The primary design pattern used for the god card implementation is the decorator pattern. This was done by creating a game logic interface. Two classes: the base game logic and abstract god card implements that interface. The four god cards created for this assignment extend the abstract card card class.

In the god card package, those classes were implemented using the template method pattern, where base (shared) logic is just put inside the abstract class. All modifiable behavior based on the four god cards is overridden in their respective class.