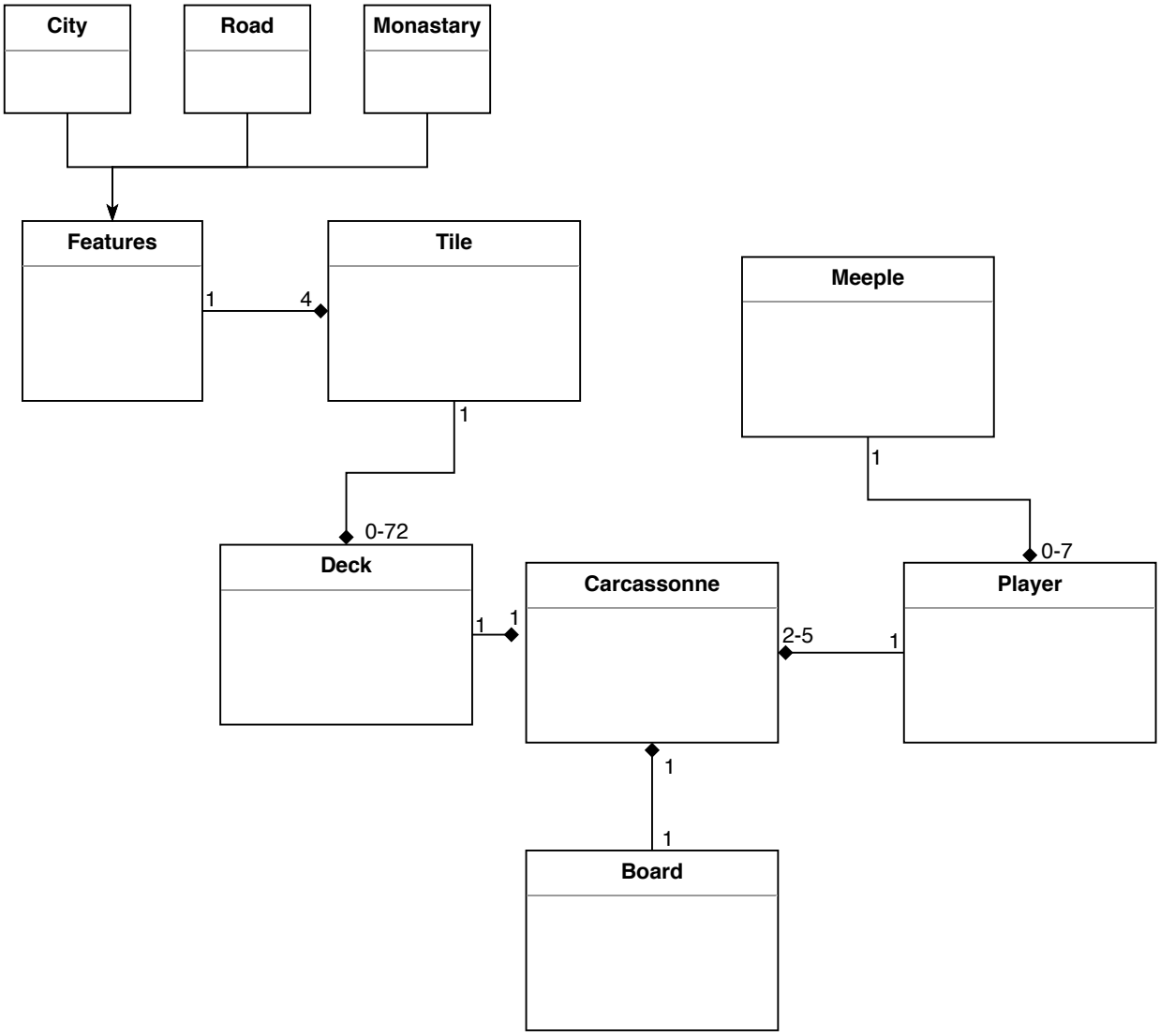
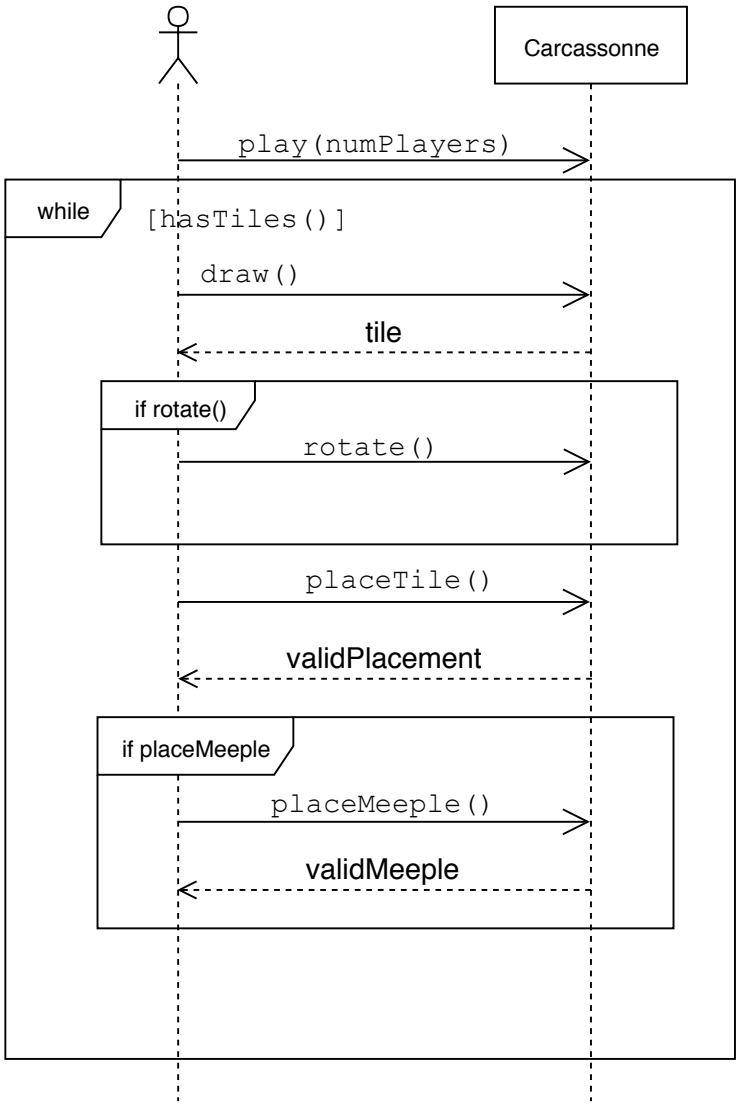

City
Road
Monastary
Features
Tile
Meeple
1
4
1
1
0-72
0-7
Deck
Carcassonne
Player
1
1
2-5
1
1
1
Board


Carcassonne
play(numPlayers)
while
[hasTiles()]
draw()
tile
if rotate()
rotate()
placeTile()
validPlacement
if placeMeeple
placeMeeple()
validMeeple

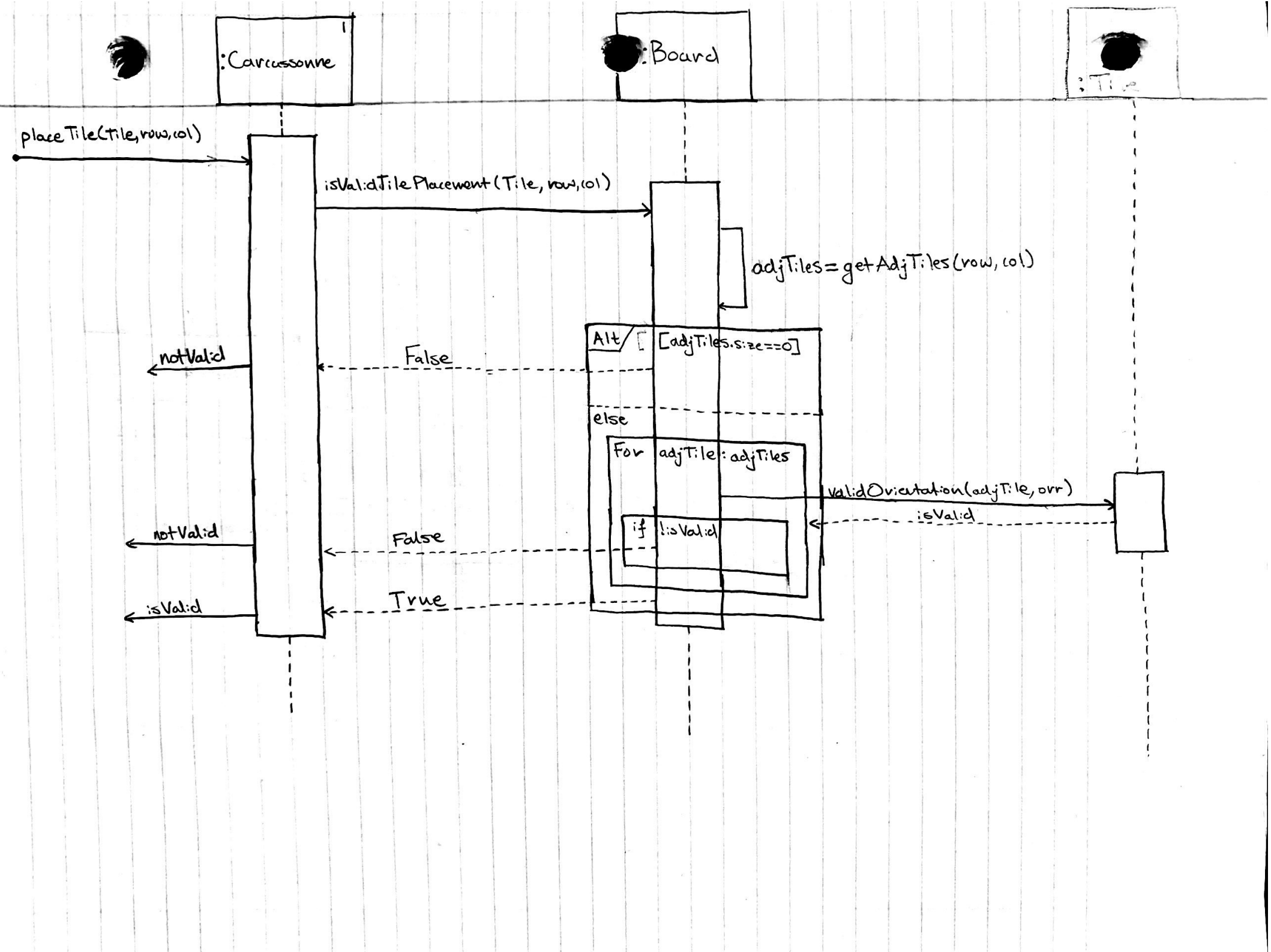
Operation: player is trying to play a tile without a meeple.

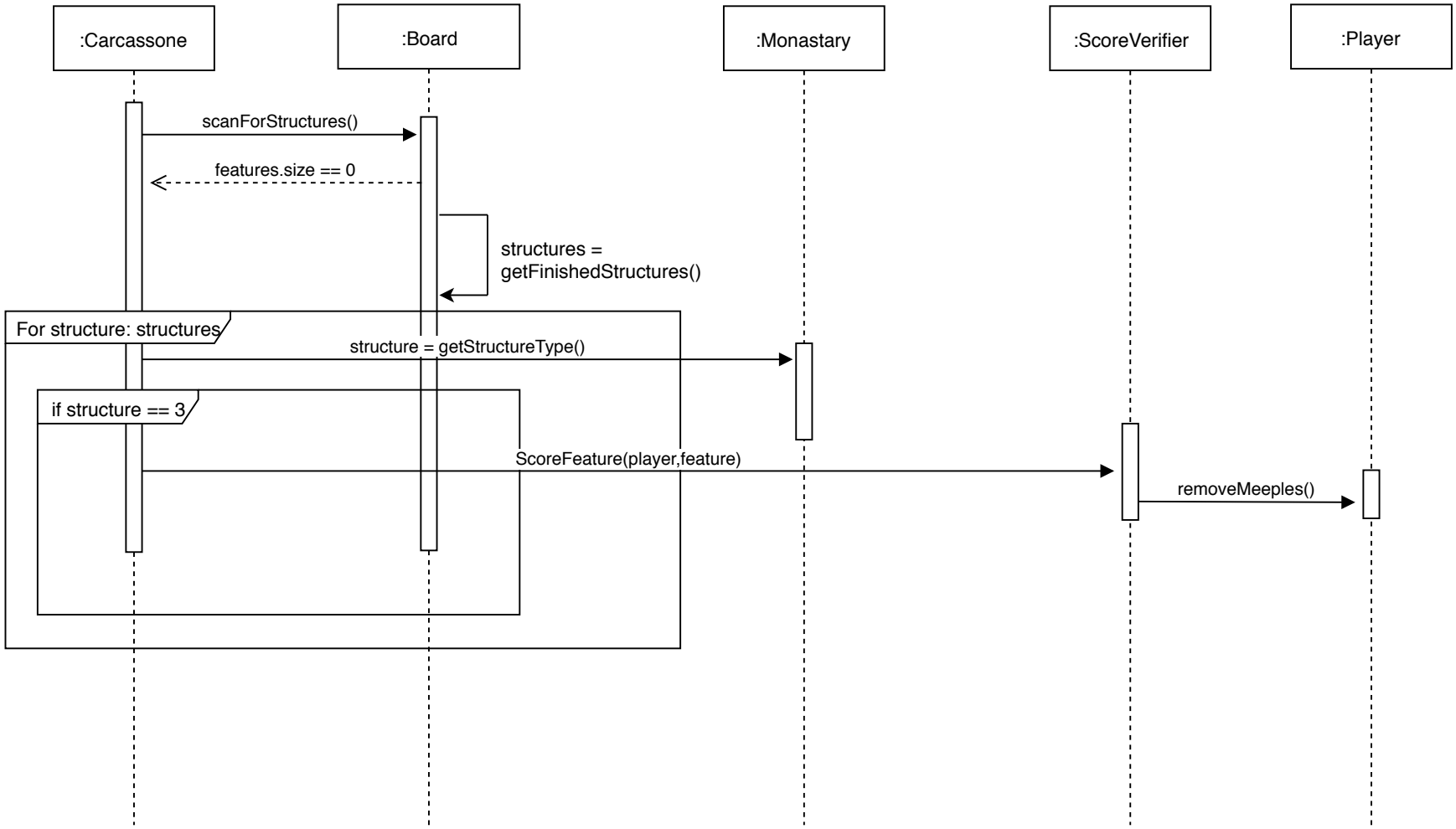
Pre-Conditions:

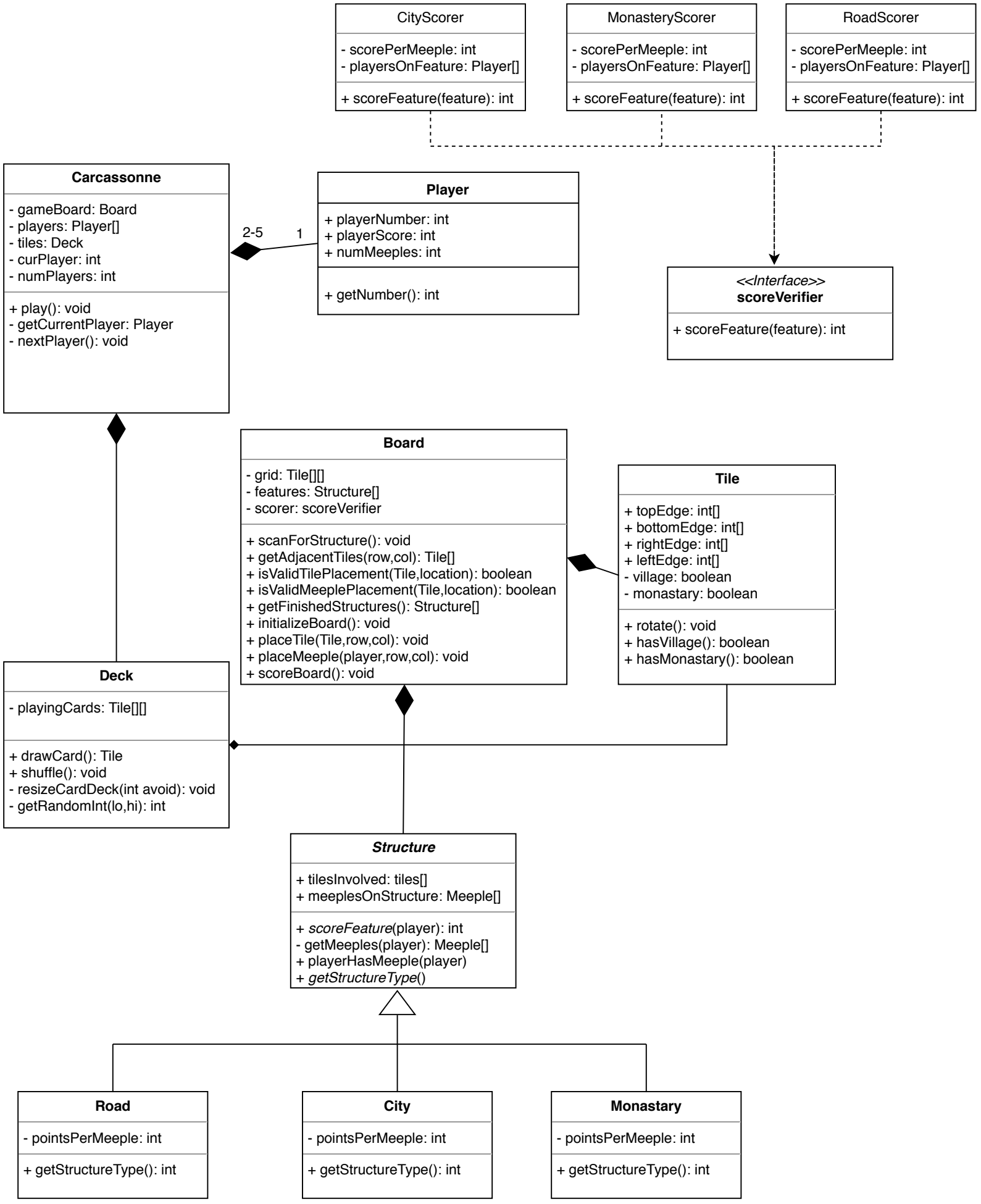
- Should be the turn of the player trying to place a tile.
- Location player is trying to place the tile, should be vacant and adjacent to at least one tile.
- Tile should be oriented such that any features that are adjacent to another tile match that tiles features.

Post-Conditions:

- The deck lost one instance of a tile, and the player no longer has the attribute tile.
- The board is modified, and now has a tile placed at the specified location.




:Carcassone
:Board
:Monastary
:ScoreVerifier
:Player
scanForStructures()
features.size == 0
structures =
getFinishedStructures()
For structure: structures
structure = getStructureType()
if structure == 3
ScoreFeature(player,feature)
removeMeeples()


CityScorer
- scorePerMeeple: int
- playersOnFeature: Player[]
+ scoreFeature(feature): int
MonasteryScorer
- scorePerMeeple: int
- playersOnFeature: Player[]
+ scoreFeature(feature): int
RoadScorer
- scorePerMeeple: int
- playersOnFeature: Player[]
+ scoreFeature(feature): int
Carcassonne
- gameBoard: Board
- players: Player[]
- tiles: Deck
- curPlayer: int
- numPlayers: int
+ play(): void
- getCurrentPlayer: Player
- nextPlayer(): void
2-5
1
Player
+ playerNumber: int
+ playerScore: int
+ numMeeples: int
+ getNumber(): int
<<Interface>>
scoreVerifier
+ scoreFeature(feature): int
Board
- grid: Tile[][]
- features: Structure[]
- scorer: scoreVerifier
+ scanForStructure(): void
+ getAdjacentTiles(row,col): Tile[]
+ isValidTilePlacement(Tile,location): boolean
+ isValidMeeplePlacement(Tile,location): boolean
+ getFinishedStructures(): Structure[]
+ initializeBoard(): void
+ placeTile(Tile,row,col): void
+ placeMeeple(player,row,col): void
+ scoreBoard(): void
Tile
+ topEdge: int[]
+ bottomEdge: int[]
+ rightEdge: int[]
+ leftEdge: int[]
- village: boolean
- monastary: boolean
+ rotate(): void
+ hasVillage(): boolean
+ hasMonastary(): boolean
Deck
- playingCards: Tile[][]
+ drawCard(): Tile
+ shuffle(): void
- resizeCardDeck(int avoid): void
- getRandomInt(lo,hi): int
Structure
+ tilesInvolved: tiles[]
+ meeplesOnStructure: Meeple[]
+ scoreFeature(player): int
- getMeeples(player): Meeple[]
+ playerHasMeeple(player)
+ getStructureType()
Road
- pointsPerMeeple: int
+ getStructureType(): int
City
- pointsPerMeeple: int
+ getStructureType(): int
Monastary
- pointsPerMeeple: int
+ getStructureType(): int

Object Model Justification:

When I started my design for the object model for Carcassonne, I asked myself what parts of the game are going to be more difficult to implement in code. I decided that actions such as scoring features, scanning the board for features and returning meeples were going to be the most difficult. To start I decided that the best way to score these different structures was to follow the strategy design pattern and create an interface called ScoreVerifier that has a method that takes in a feature and figures out how to score the given feature. The method that gets implemented by all 3 structures is the scoreFeature method that finds out how many tiles are in the structure and what players have meeple on that structure and then to calculate the score based on the rules. I also decided to use the strategy design pattern because I can easily implement new methods that I might have not thought about during the design process. I decided to use the strategy design pattern over the template method design pattern in this instance because the shared functionality between scoring a structure would simply be finding that structure and players who have meeples on it, but this method is handled in the board class.

Another design choice I made during my design process was to let the Tile class have 4 instances of the feature class. This really made sense to me representationally because in the actual game a tile is essentially made up of 4 smaller squares that are either fields cities or roads. This decision I feel will also make the process of finding different structures a lot simpler, because then I can analyze each tile in detail, instead of a tile in general.

When I decided to design how I was going to handle the different structures such as the cities monasteries and roads, I decided it would make the most sense for these structures to be different classes. I decided it would be best to use inheritance in this situation because there is a lot of shared functionality for each structure, for example every structure has , or could have players meeples on them, also every single structure has tiles that make up that structure. Whe using inheritance I can program methods and attributes that handle how to get these tiles and meeples and wont program these for every structure. Where they do not share functionality is what structure they are which is what the abstract method will handle differently for each structures. It will also handle how many points each structure is worth, because different structure's are worth different points.

Overall the design process for this implementation takes advantage of the board class containing a lot of the attributes of the game. For example scanning for finished structures is handled by the board, because the board class knows where all the tiles are, and where the player's meeples are

located. I decided to handle things this way so that the a lot the information can remain hidden, and wont need to be passed from class to class with the chance of being aliased accidentally and corrupting that data.