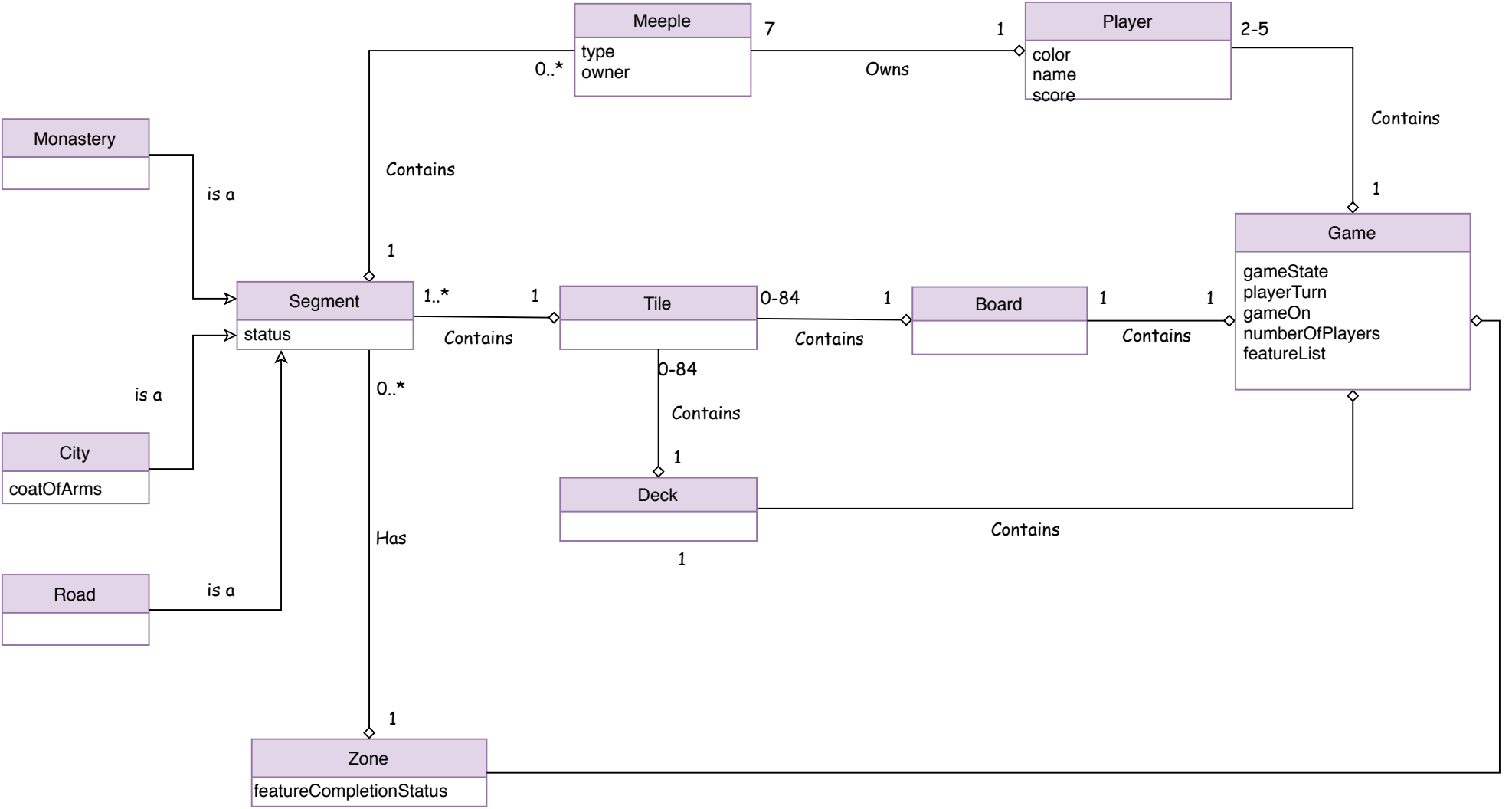
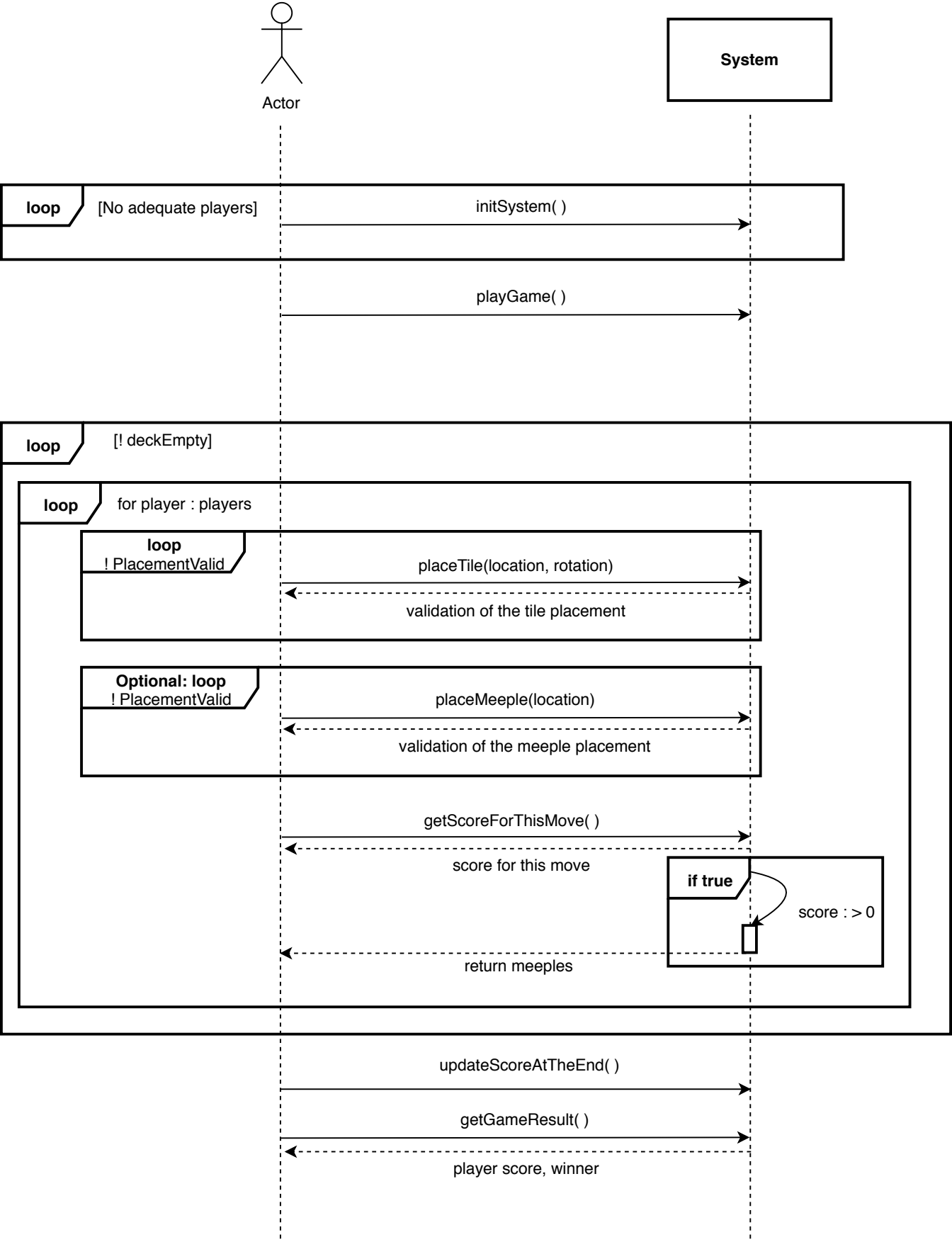

Meeple
type
owner
7
0..*
Owns
1
Player
color
name
score
2-5
Contains
1
Monastery
is a
Contains
1
Segment
status
1..*
1
Contains
Tile
0-84
1
Contains
Board
1
1
Contains
Game
gameState
playerTurn
gameOn
numberOfPlayers
featureList
is a
City
coatOfArms
0..*
0-84
Contains
1
Deck
1
Contains
Has
Road
is a
1
Zone
featureCompletionStatus


Actor
System
loop
[No adequate players]
initSystem( )
playGame( )
loop
[! deckEmpty]
loop
for player : players
loop
! PlacementValid
placeTile(location, rotation)
validation of the tile placement
Optional: loop
! PlacementValid
placeMeeple(location)
validation of the meeple placement
getScoreForThisMove( )
score for this move
if true
score : > 0
return meeples
updateScoreAtTheEnd( )
getGameResult( )
player score, winner

Contract CO1: joinGame

Operation:	joinGame(playerName)
Cross References:	Use Cases: Join Game
Preconditions	1. Number of players < 2 2. Player name is valid
Postconditions	1. One more player added into the game 2. Total number of players in the game increases by one

Contract CO2: playGame

Operation:	playGame( )
Cross References:	Use Cases: In Game
Preconditions	1. 2 <= Number of players <5 2. Board is ready for the game 3. Tiles are ready for the game 4. Deck is ready for the game 5. Players are ready for the game
Postconditions	1. GameOn is set to true

Contract CO3: placeTile

Operation:	placeTile(location, rotation)
Cross References:	Use Cases: In Game
Preconditions	1. GameOn is true 2. Deck is not empty 3. Previous placement is not valid
Postconditions	1. Return the validity of this tile placement

Contract CO4: placeMeeple

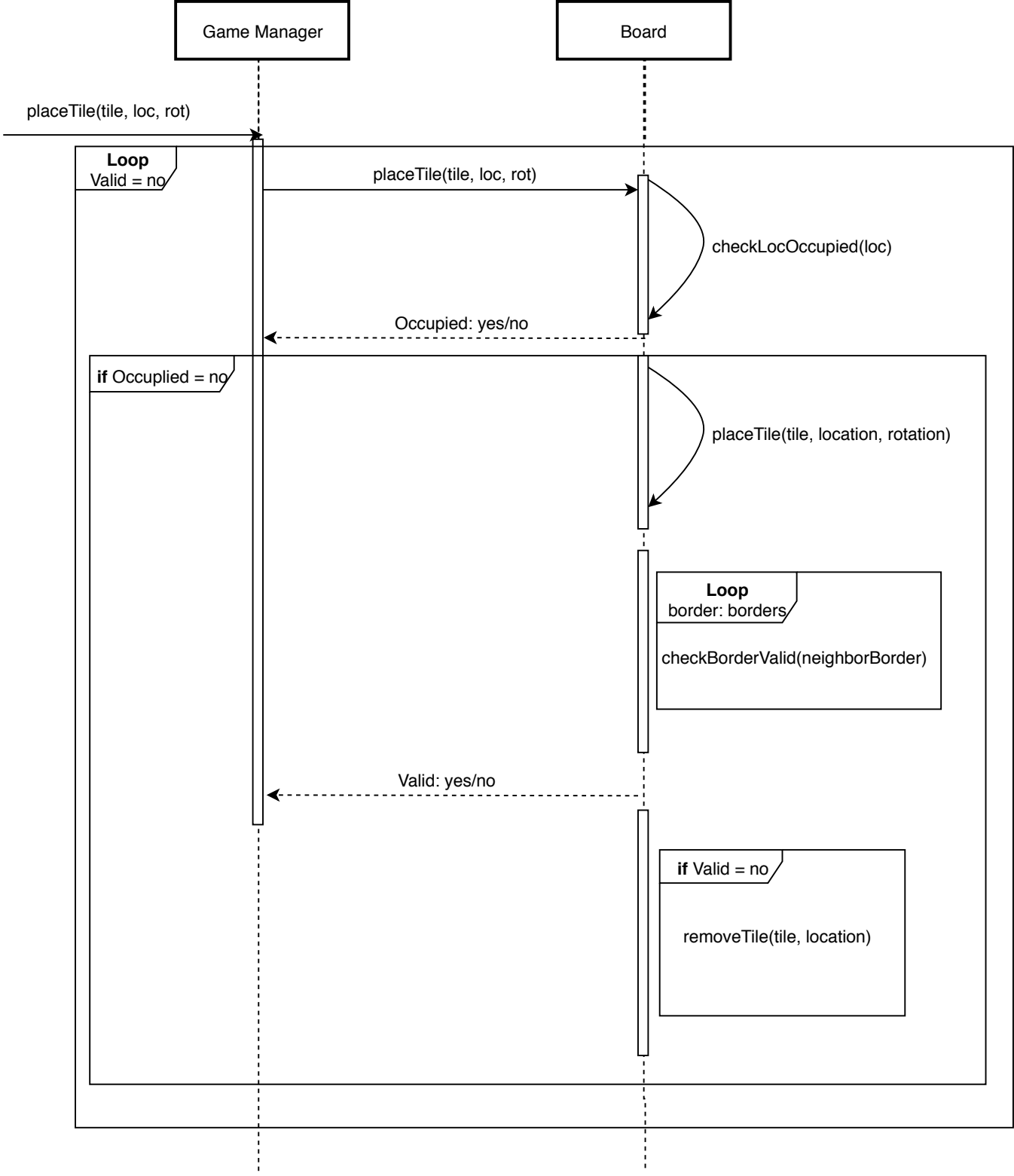
Operation:	placeMeeple(location)
Cross References:	Use Cases: In Game
Preconditions	1. GameOn is true 2. Deck is not empty 3. Previous placement is not valid
Postconditions	1. Return the validity of this meeple placement

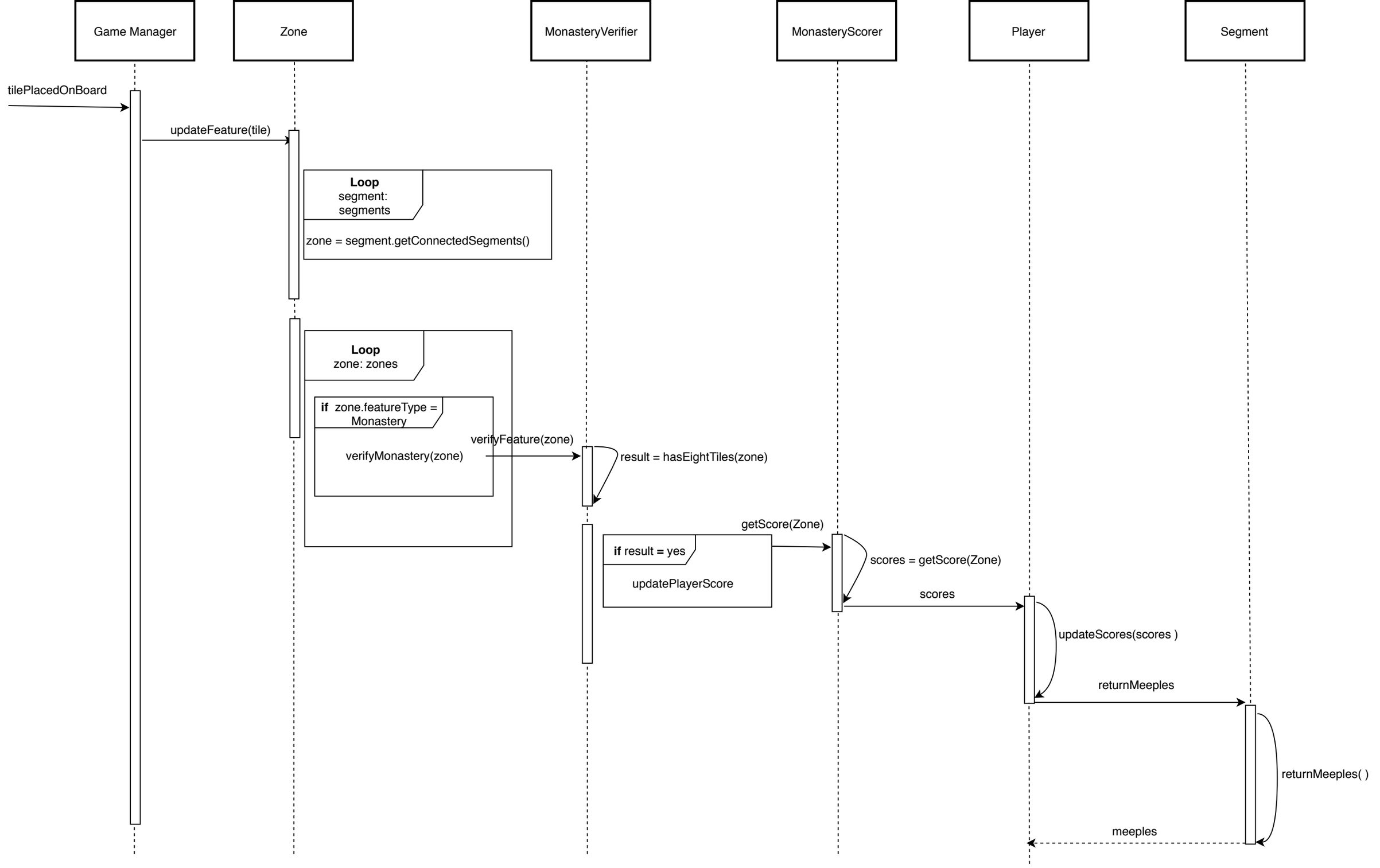
Contract CO5: getScore

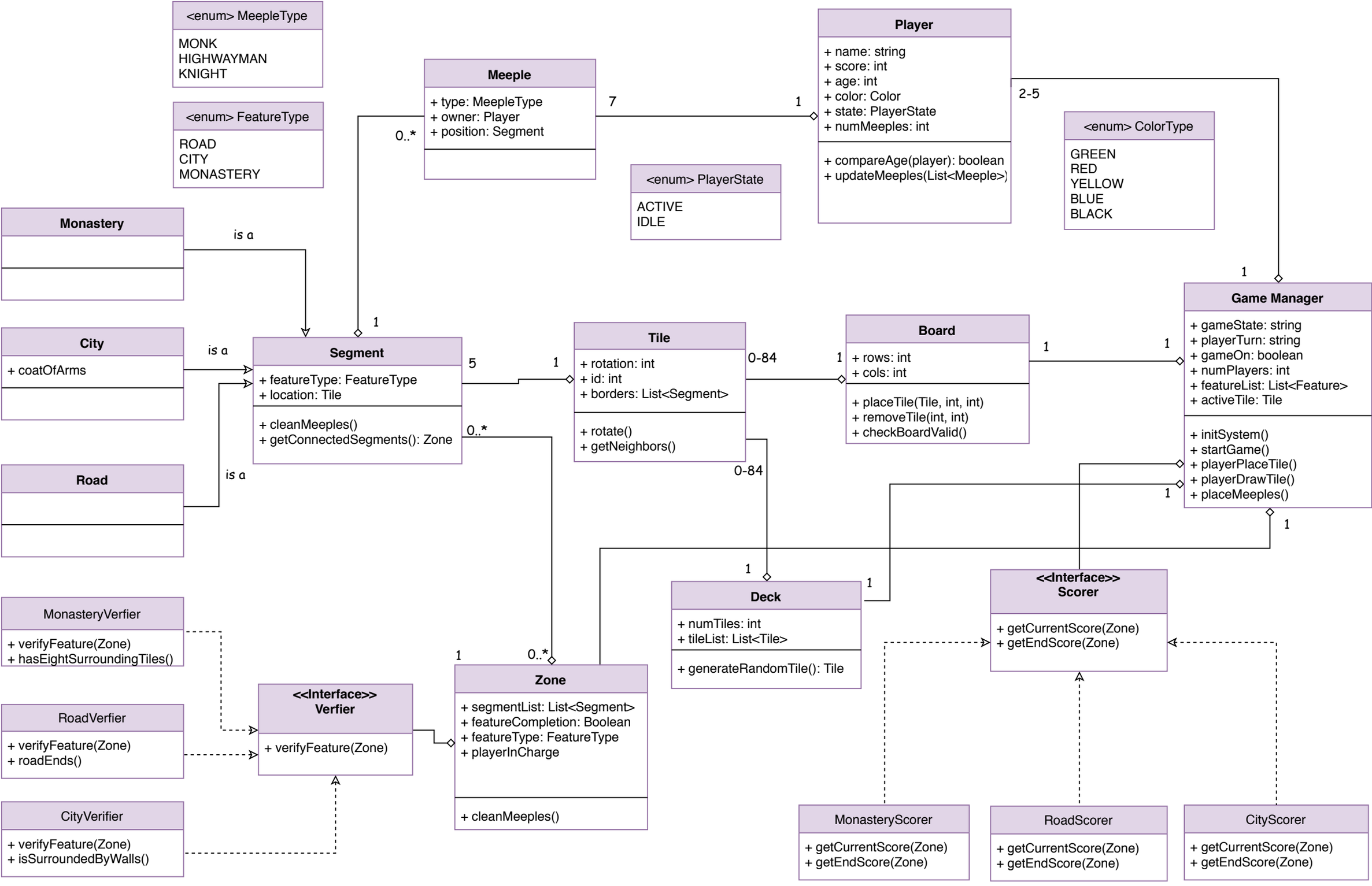
Operation:	getScore( )
Cross References:	Use Cases: In Game
Preconditions	1. GameOn is True 2. Placement of tile is valid 3. Placement of meeple is valid
Postconditions	1. Score for this move is computed

Contract CO6: getResult

Operation:	getResult( )
Cross References:	Use Cases: After Game
Preconditions	1. GameOn is false 2. Deck is empty
Postconditions	1. Winner is obtained


Game Manager
Board
placeTile(tile, loc, rot)
Loop
Valid = no
placeTile(tile, loc, rot)
checkLocOccupied(loc)
Occupied: yes/no
if Occuplied = no
placeTile(tile, location, rotation)
Loop
border: borders
checkBorderValid(neighborBorder)
Valid: yes/no
if Valid = no
removeTile(tile, location)


Game Manager
Zone
MonasteryVerifier
MonasteryScorer
Player
Segment
tilePlacedOnBoard
updateFeature(tile)
Loop
segment:
segments
zone = segment.getConnectedSegments()
Loop
zone: zones
if zone.featureType =
Monastery
verifyMonastery(zone)
verifyFeature(zone)
result = hasEightTiles(zone)
getScore(Zone)
if result = yes
updatePlayerScore
scores = getScore(Zone)
scores
updateScores(scores)
returnMeeples
returnMeeples( )
meeples


<enum> MeepleType
MONK
HIGHWAYMAN
KNIGHT
<enum> FeatureType
ROAD
CITY
MONASTERY
Meeple
+ type: MeepleType
+ owner: Player
+ position: Segment
7
1
0..*
Player
+ name: string
+ score: int
+ age: int
+ color: Color
+ state: PlayerState
+ numMeeples: int
+ compareAge(player): boolean
+ updateMeeples(List<Meeple>)
2-5
<enum> ColorType
GREEN
RED
YELLOW
BLUE
BLACK
<enum> PlayerState
ACTIVE
IDLE
Monastery
is a
City
+ coatOfArms
is a
Road
is a
Segment
+ featureType: FeatureType
+ location: Tile
+ cleanMeeples()
+ getConnectedSegments(): Zone
1
5
1
0..*
Tile
+ rotation: int
+ id: int
+ borders: List
+ rotate()
+ getNeighbors()
0-84
0-84
Board
+ rows: int
+ cols: int
+ placeTile(Tile, int, int)
+ removeTile(int, int)
+ checkBoardValid()
1
1
Game Manager
+ gameState: string
+ playerTurn: string
+ gameOn: boolean
+ numPlayers: int
+ featureList: List<Feature>
+ activeTile: Tile
+ initSystem()
+ startGame()
+ playerPlaceTile()
+ playerDrawTile()
+ placeMeeples()
1
1
1
Deck
+ numTiles: int
+ tileList: List<Tile>
+ generateRandomTile(): Tile
1
1
<<Interface>>
Scorer
+ getCurrentScore(Zone)
+ getEndScore(Zone)
MonasteryVerfier
+ verifyFeature(Zone)
+ hasEightSurroundingTiles()
RoadVerfier
+ verifyFeature(Zone)
+ roadEnds()
CityVerifier
+ verifyFeature(Zone)
+ isSurroundedByWalls()
<<Interface>>
Verfier
+ verifyFeature(Zone)
1
0..*
Zone
+ segmentList: List
+ featureCompletion: Boolean
+ featureType: FeatureType
+ playerInCharge
+ cleanMeeples()
MonasteryScorer
+ getCurrentScore(Zone)
+ getEndScore(Zone)
RoadScorer
+ getCurrentScore(Zone)
+ getEndScore(Zone)
CityScorer
+ getCurrentScore(Zone)
+ getEndScore(Zone)

Justify design decisions

My design decisions include:

1. GameManager Class
First, I think it is necessary to have a game manager to take care of the life cycle of the game. It manages the game process from a high-level including checking how many players are in the game and keeping track of the state of the game, and active tile. It also functions as the interface of GUI and the internal game components.

2. Player Class
For the game to play, player class is created to represent actual players and keep track of their name, age, score, and others.

3. Board Class
Next, I think we should have a board class that corresponds to the actual game map where the game is played on. The map consists of tiles and is also responsible for checking if the tile placement is valid or not every time the player places it.

4. Tile Class
Tile class is drawn from the deck and placed by the player. It can rotate and has a location on the map. A tile consists of several segments, which can be renamed as borders.

5. Deck Class
Deck class generates a random tile for each player in each round of the game.

6. Segment Class
Segment class is designed as an abstract class that can be inherited by different concrete types of landscapes, which include roads, cities, monasteries. All these subtypes of landscapes share some invariant components but also some variant parts, say, the coat of arm in the city. Therefore, template method design pattern is chosen because there is a strong relationship observed between the abstract segment type and all those concrete segment types. In addition, it is also responsible for clearing meeples that are on this particular segment.

7. Zone Class
Zone class is composed of segments. Zone can be completed as a feature or as just left incomplete. Zone can be composed by connecting segments of the same type together.

8. Meeple Class
Meeple class is a simple class that belongs to the player. It is used to determine which player the feature belongs to when it is complete.

9. Verifier Class
In addition to all classes that correspond to concrete game components, we need to have a verifier class that specifically checks if a certain type of feature is completed. Since there are different mechanisms for different types of landscapes, which is similar to the concept of applying different strategies that serve similar purposes, strategy design pattern is used here. Therefore, a common interface is defined.

10. Scorer Class
This class is similar to the verifier class except it is specifically used to compute scores for different types of features. Each scorer implements two methods. One is used for computing scores before the game ends while the other one is used for computing scores when the game ends.