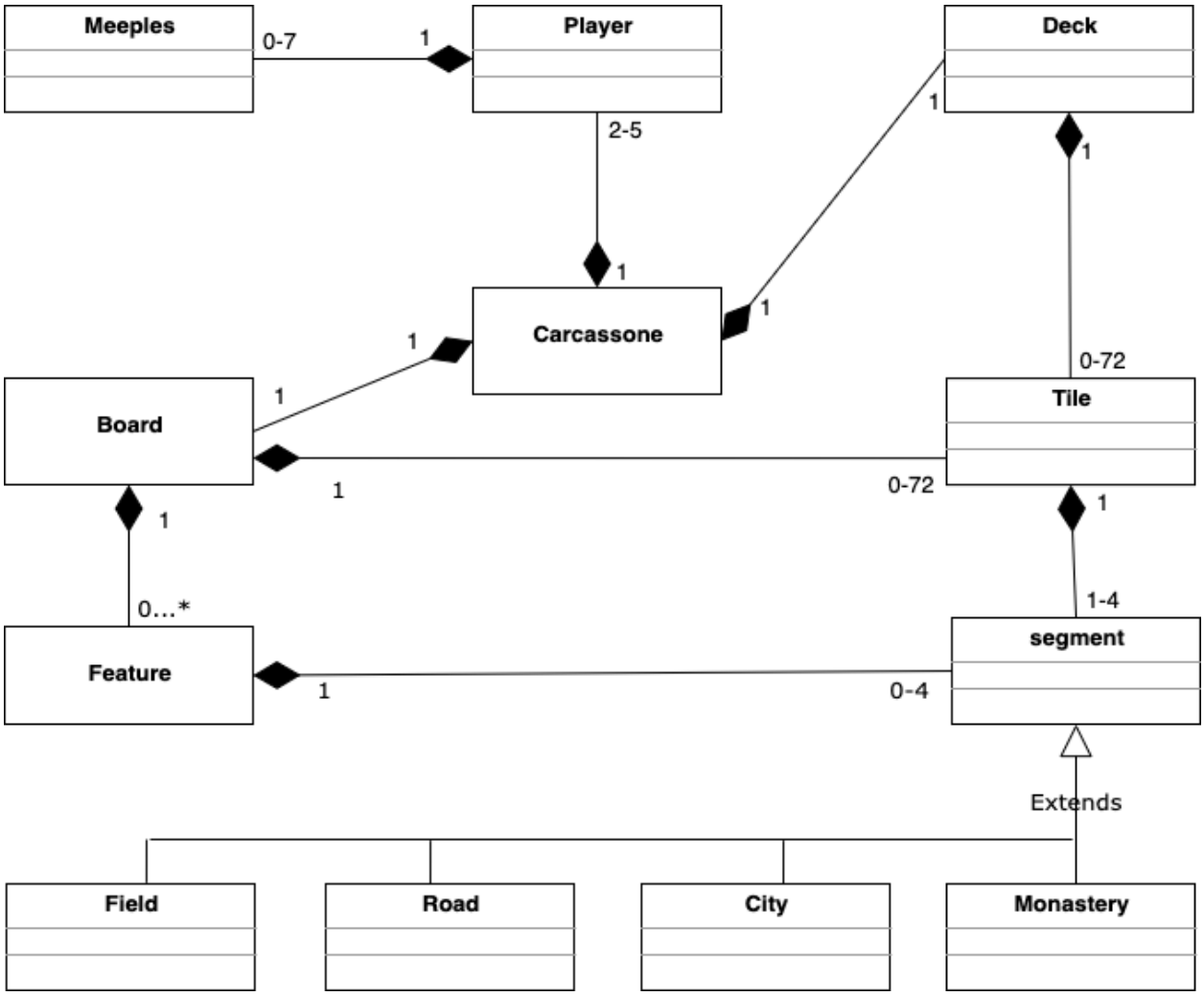
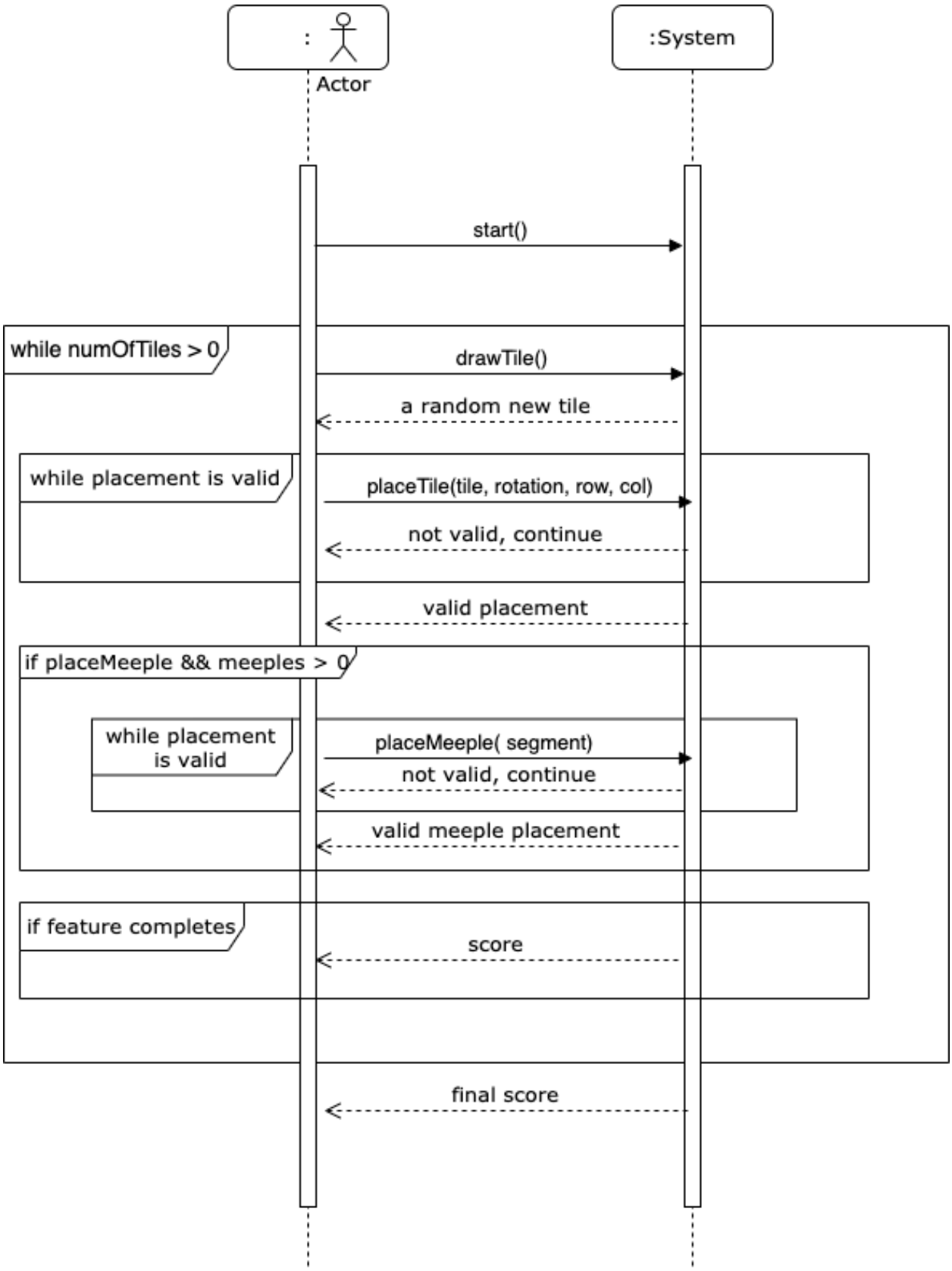

Meeples
0-7
1
Player
Deck
2-5
1
1
1
Carcassone
1
1
0-72
Board
Tile
1
1
0-72
1
1
1-4
0…*
Feature
segment
1
0-4
Extends
Field
Road
City
Monastery


:
Actor
:System
start()
while numOfTiles > 0
drawTile()
a random new tile
while placement is valid
placeTile(tile, rotation, row, col)
not valid, continue
valid placement
if placeMeeple && meeples > 0
while placement is valid
placeMeeple(segment)
not valid, continue
valid meeple placement
if feature completes
score
final score

Behavioral Contract

Operation: Playing a tile without a meeple

Pre-condition:

1. It is the current player's turn to play a tile
2. The current player gets a new tile from the deck
3.
    A. There is at least one valid space to place the tile (by saying valid space, it means this place is a blank space adjacent to the edges of current tiles and the current tile feature could match them all)
    B. There is no valid space to place the current tile

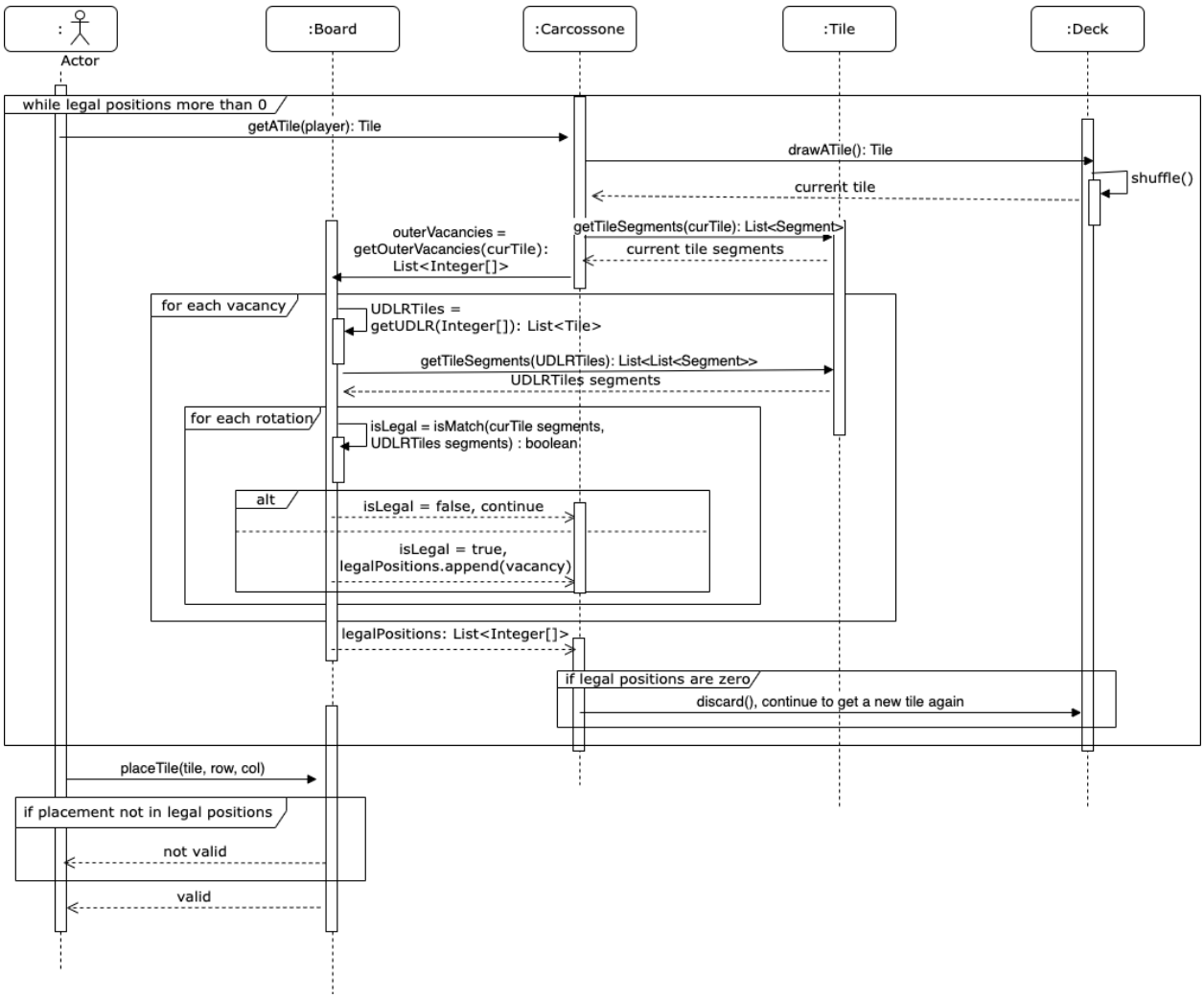
Post-condition:

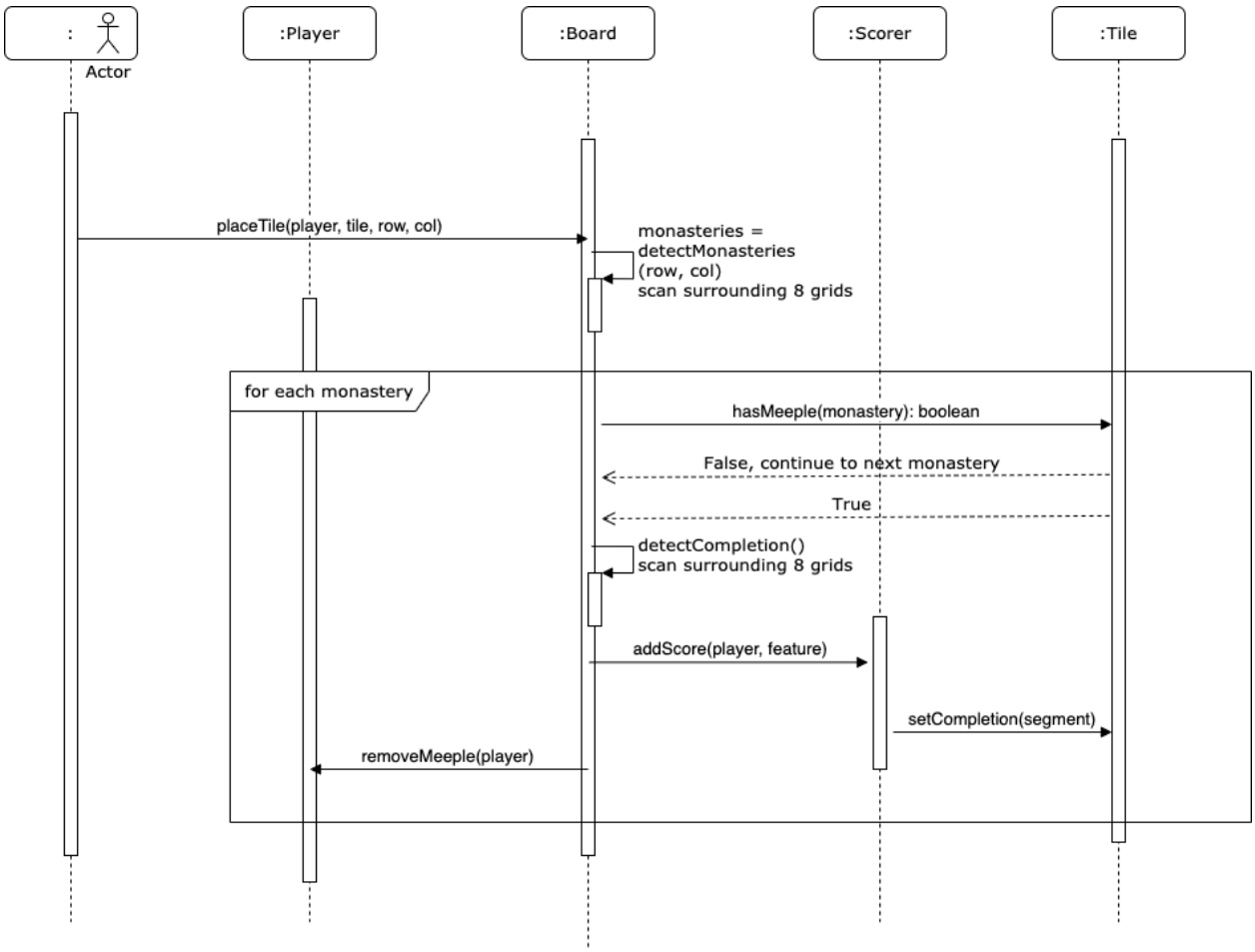
A.

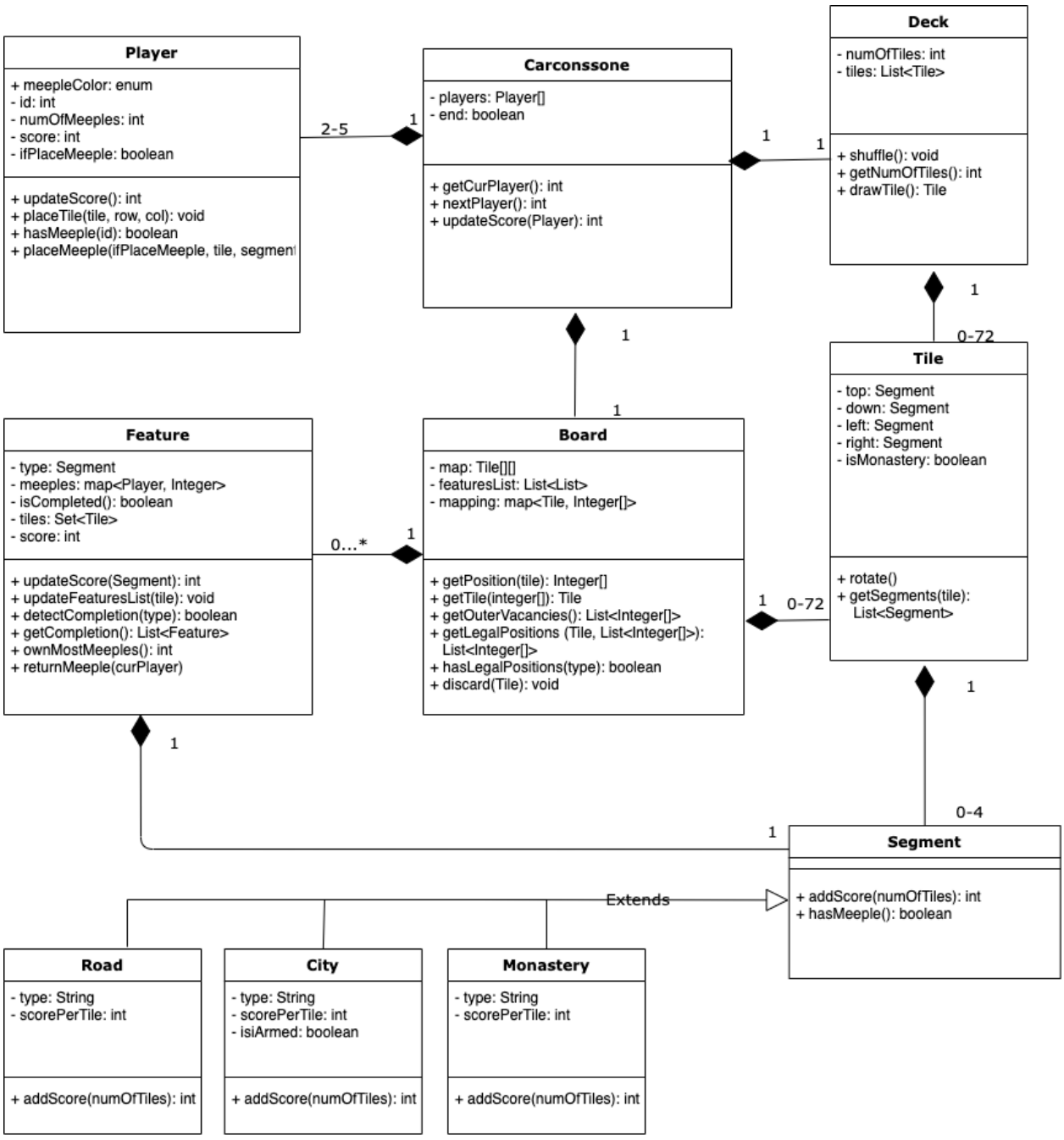
1. the current player chooses one of valid space to put the current tile and the grid updates, feature lists may increase
2. the tiles in deck decreases one

B.

1. The processor discards current tile and the deck decreases one tile
2. The current player gets another new tile


:
Actor
:Board
:Carcossone
:Tile
:Deck
while legal positions more than 0
getATile(player): Tile
drawATile(): Tile
shuffle()
current tile
outerVacancies =
getOuterVacancies(curTile):
List<Integer[]>
getTileSegments(curTile): List
current tile segments
for each vacancy
UDLRTiles =
getUDLR(Integer[]): List<Tile>
getTileSegments(UDLRTiles): List<List >
UDLRTiles segments
for each rotation
isLegal = isMatch(curTile segments,
UDLRTiles segments) : boolean
alt
isLegal = false, continue
isLegal = true,
legalPositions.append(vacancy)
legalPositions: List<Integer[]>
if legal positions are zero
discard(), continue to get a new tile again
placeTile(tile, row, col)
if placement not in legal positions
not valid
valid


:
Actor
:Player
:Board
:Scorer
:Tile
placeTile(player, tile, row, col)
monasteries =
detectMonasteries
(row, col)
scan surrounding 8 grids
for each monastery
hasMeeple(monastery): boolean
False, continue to next monastery
True
detectCompletion()
scan surrounding 8 grids
addScore(player, feature)
setCompletion(segment)
removeMeeple(player)


Player
+ meepleColor: enum
- id: int
- numOfMeeples: int
- score: int
- ifPlaceMeeple: boolean
+ updateScore(): int
+ placeTile(tile, row, col): void
+ hasMeeple(id): boolean
+ placeMeeple(ifPlaceMeeple, tile, segment
2-5
1
Carconssone
- players: Player[]
- end: boolean
+ getCurPlayer(): int
+ nextPlayer(): int
+ updateScore(Player): int
1
1
Deck
- numOfTiles: int
- tiles: List<Tile>
+ shuffle(): void
+ getNumOfTiles(): int
+ drawTile(): Tile
1
0-72
Tile
- top: Segment
- down: Segment
- left: Segment
- right: Segment
- isMonastery: boolean
+ rotate()
+ getSegments(tile): List
1
1
Feature
- type: Segment
- meeples: map<Player, Integer>
- isCompleted(): boolean
- tiles: Set<Tile>
- score: int
+ updateScore(Segment): int
+ updateFeaturesList(tile): void
+ detectCompletion(type): boolean
+ getCompletion(): List<Feature>
+ ownMostMeeples(): int
+ returnMeeple(curPlayer)
0...*
1
Board
- map: Tile[][]
- featuresList: List<List>
- mapping: map<Tile, Integer[]>
+ getPosition(tile): Integer[]
+ getTile(integer[]): Tile
+ getOuterVacancies(): List<Integer[]>
+ getLegalPositions (Tile, List<Integer[]>): List<Integer[]>
+ hasLegalPositions(type): boolean
+ discard(Tile): void
1
0-72
1
1
0-4
Segment
+ addScore(numOfTiles): int
+ hasMeeple(): boolean
Extends
Road
- type: String
- scorePerTile: int
+ addScore(numOfTiles): int
City
- type: String
- scorePerTile: int
- isiArmed: boolean
+ addScore(numOfTiles): int
Monastery
- type: String
- scorePerTile: int
+ addScore(numOfTiles): int

Rationale.

From my object model, you can see I designed Player, Carcassone(controller), Board, Deck, Tile, Segment, Feature classes, Segment has three different subclasses: Road, Monastery and City, because they have invariants and variants, so basically I plan to write Segment into an abstract class with three different expansions. The key point of this game design is the interactions among those classes and core idea to validate. I used Player to denote meeple to reduce an meeple class, and one Carcassone has one board, one deck, one deck has multiple tiles, one board has multiple features, one tile has up to 4 segments, that segment means the edge characteristic, and each feature only contains the same segment to group them and check completion.

When it is the turn for a player, the Carcassone would request a random tile from deck, get the segments of current tile, then the board would generate outer vacancies as potential legal position, then for each vacancy, the Board try to match the segments of current tile with each vacancy's surrounding tiles(up, down, left, right), with four rotations, and finally return back all of legal positions. Then it is very easy for the game to validate whether the placement of tile is legal or illegal, like if the player puts the tile at the position out of legal positions, then alert invalid and try again, also, the player could choose to rotate, which increase the interest of the game. Why would I decide to generate all of legal positions before the player make the placement? This is because each time you have to consider if there is no legal positions at all, so actually each time when there is a new tile, the board should check this corner case, or it would be a dead circulation, so it should be a smart way to generate all of the legal positions while checking this corner case, right?

Another point is my Feature class, initially I was confused about how to get score function according to feature with clean(less repetition) classes, then I decide to put a score field and updateScore() method into it, since each feature has only one type of segment(for example, a city, this doesn't mean that all of tiles on one feature has only one kind of segment, I just define the feature by segment), then I could write different attributes to different segment, ang use polymorphism to get different score rules, since segment should not have a score function, so I decide to give it a scorePerTile field, and each time there is a new segment added to the feature, the score of current feature would update, if the feature is completed, just get the score and pass it to the player who owns most meeples.
Also, data structures for representing different field are crucial. I used a Tile[][] to denote the board grid, and there should be a map to reflect the coordinate and tile as well. I used List<> to represent features, legal positions and outer vacancies and so on because they are not length fixed, and I used a map to represent meeple counts and player's relationship, which could easily get the player(s) who has most meeples.

I designed so many delegations, which is strongly necessarily, I believe.