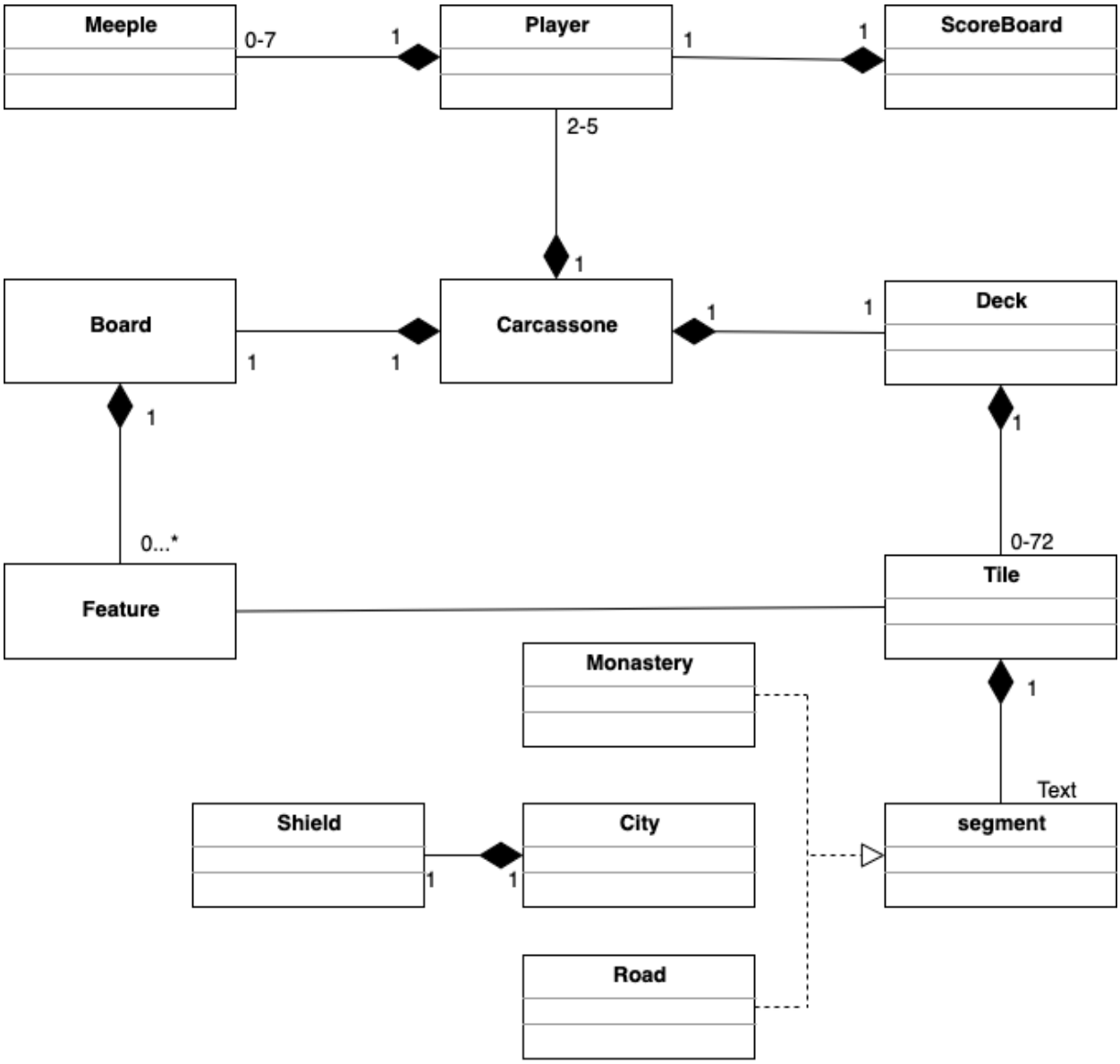
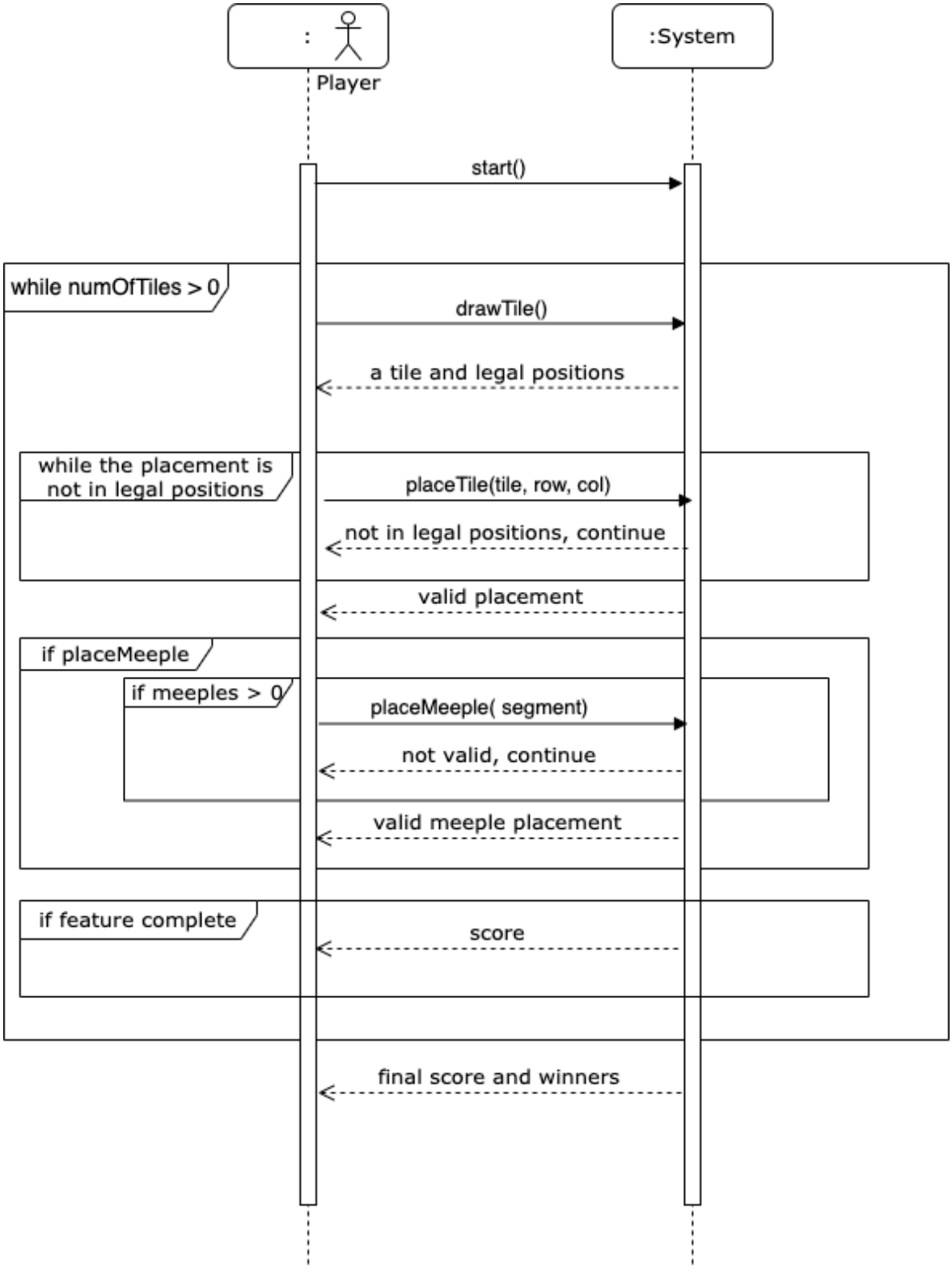

Meeple
0-7
1
Player
1
1
ScoreBoard
2-5
1
Board
1
1
Carcassone
1
1
Deck
1
1
0...*
0-72
Feature
Tile
Monastery
1
Text
Shield
1
1
City
segment
Road


:
Player
:System
start()
while numOfTiles > 0
drawTile()
a tile and legal positions
while the placement is not in legal positions
placeTile(tile, row, col)
not in legal positions, continue
valid placement
if placeMeeple
if meeples > 0
placeMeeple(segment)
not valid, continue
valid meeple placement
if feature complete
score
final score and winners

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

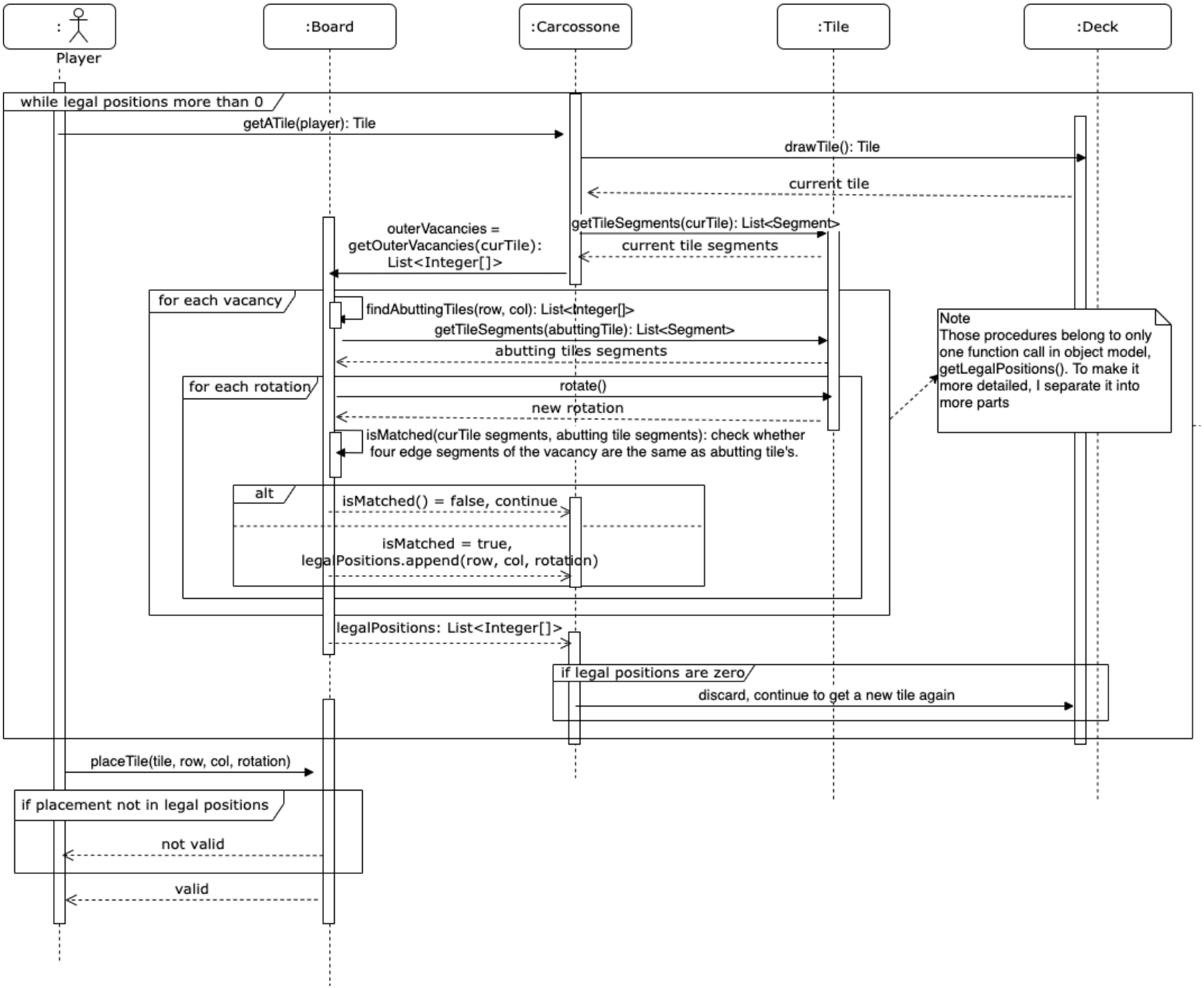
Post-condition:

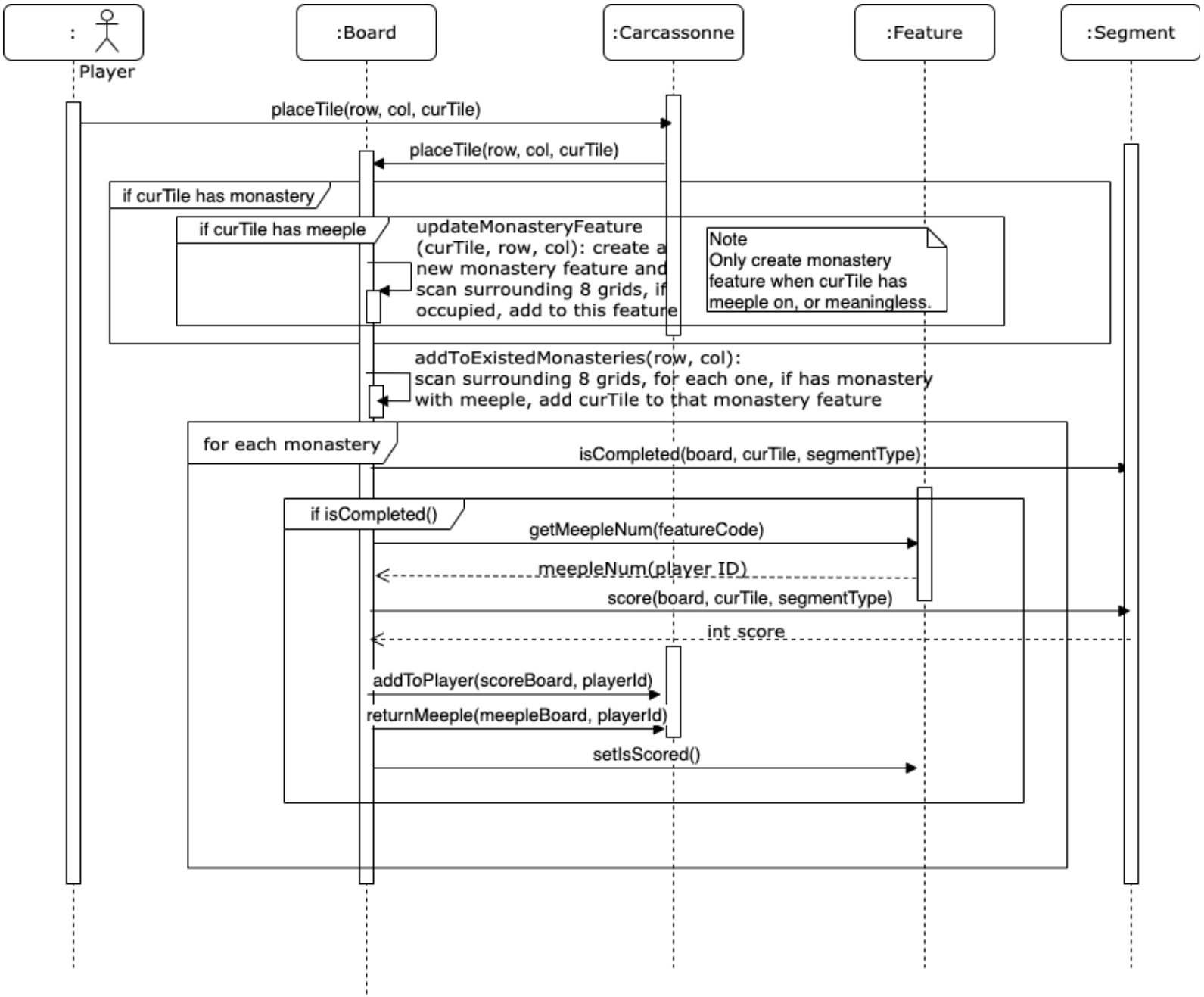
A.

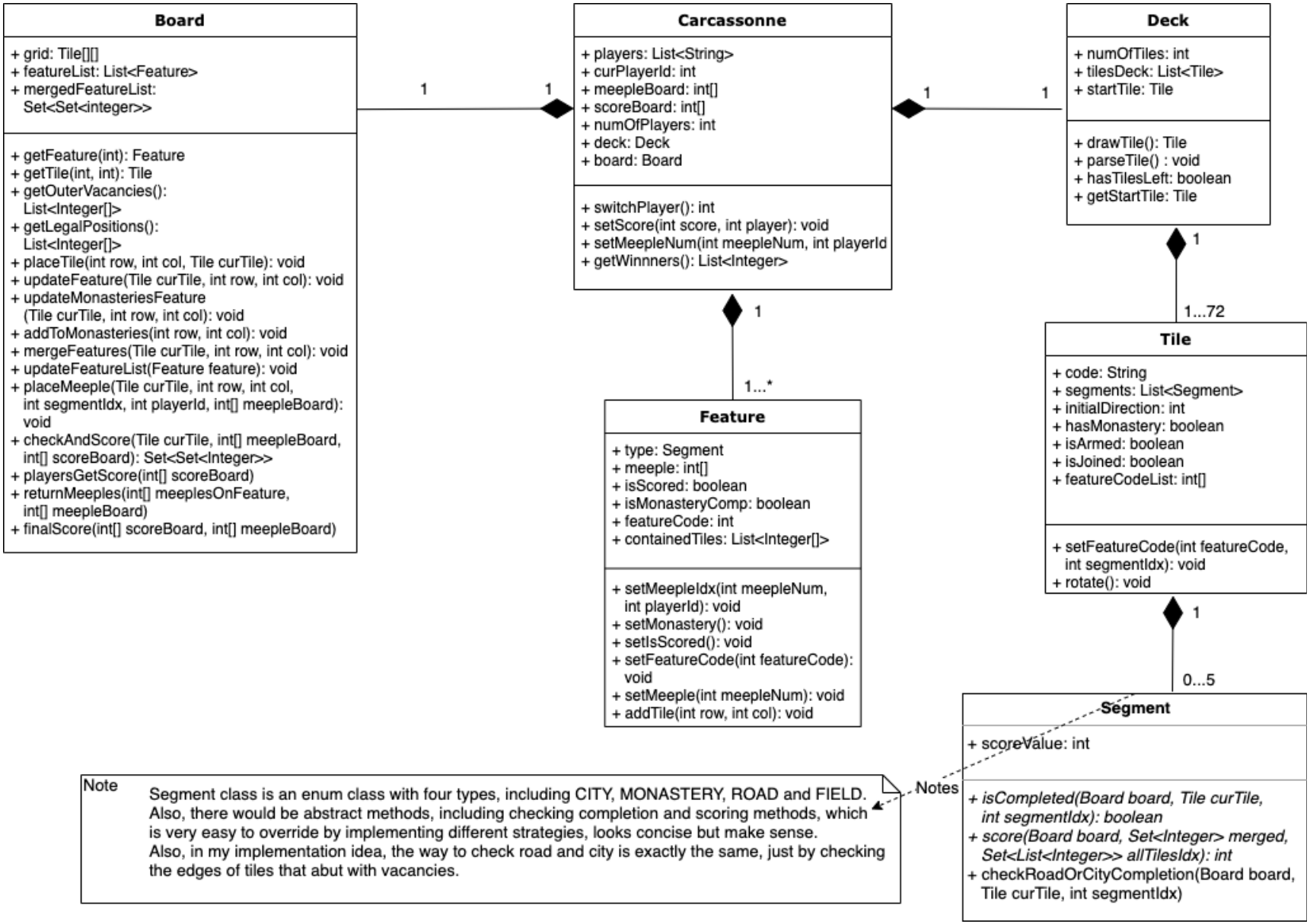
1. the current player chooses one of valid space to put the current tile and the grid updates, feature lists may increase
2. the tiles in deck decreases one

B.

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


:
Player
:Board
:Carcossone
:Tile
:Deck
while legal positions more than 0
getATile(player): Tile
drawTile(): Tile
current tile
outerVacancies =
getOuterVacancies(curTile):
List<Integer[]>
getTileSegments(curTile): List
current tile segments
for each vacancy
findAbuttingTiles(row, col): List<Integer[]>
getTileSegments(abuttingTile): List
abutting tiles segments
for each rotation
rotate()
new rotation
isMatched(curTile segments, abutting tile segments): check whether
four edge segments of the vacancy are the same as abutting tile's.
alt
isMatched() = false, continue
isMatched = true,
legalPositions.append(row, col, rotation)
legalPositions: List<Integer[]>
Note
Those procedures belong to only one function call in object model, getLegalPositions(). To make it more detailed, I separate it into more parts
if legal positions are zero
discard, continue to get a new tile again
placeTile(tile, row, col, rotation)
if placement not in legal positions
not valid
valid


:
Player
:Board
:Carcassonne
:Feature
:Segment
placeTile(row, col, curTile)
placeTile(row, col, curTile)
if curTile has monastery
if curTile has meeple
updateMonasteryFeature
(curTile, row, col): create a
new monastery feature and
scan surrounding 8 grids, if
occupied, add to this feature
Note
Only create monastery
feature when curTile has
meeple on, or meaningless.
addToExistedMonasteries(row, col):
scan surrounding 8 grids, for each one, if has monastery
with meeple, add curTile to that monastery feature
for each monastery
isCompleted(board, curTile, segmentType)
if isCompleted()
getMeepleNum(featureCode)
meepleNum(player ID)
score(board, curTile, segmentType)
int score
addToPlayer(scoreBoard, playerId)
returnMeeple(meepleBoard, playerId)
setIsScored()


Board
+ grid: Tile[][]
+ featureList: List<Feature>
+ mergedFeatureList: Set<Set<integer>>
+ getFeature(int): Feature
+ getTile(int, int): Tile
+ getOuterVacancies(): List<Integer[]>
+ getLegalPositions(): List<Integer[]>
+ placeTile(int row, int col, Tile curTile): void
+ updateFeature(Tile curTile, int row, int col): void
+ updateMonasteriesFeature (Tile curTile, int row, int col): void
+ addToMonasteries(int row, int col): void
+ mergeFeatures(Tile curTile, int row, int col): void
+ updateFeatureList(Feature feature): void
+ placeMeeple(Tile curTile, int row, int col, int segmentIdx, int playerId, int[] meepleBoard): void
+ checkAndScore(Tile curTile, int[] meepleBoard, int[] scoreBoard): Set<Set<Integer>>
+ playersGetScore(int[] scoreBoard)
+ returnMeeples(int[] meeplesOnFeature, int[] meepleBoard)
+ finalScore(int[] scoreBoard, int[] meepleBoard)
1
1
Carcassonne
+ players: List<String>
+ curPlayerId: int
+ meepleBoard: int[]
+ scoreBoard: int[]
+ numOfPlayers: int
+ deck: Deck
+ board: Board
+ switchPlayer(): int
+ setScore(int score, int player): void
+ setMeepleNum(int meepleNum, int playerId
+ getWinnners(): List<Integer>
1
1
Deck
+ numOfTiles: int
+ tilesDeck: List<Tile>
+ startTile: Tile
+ drawTile(): Tile
+ parseTile() : void
+ hasTilesLeft: boolean
+ getStartTile: Tile
1
1...72
Tile
+ code: String
+ segments: List
+ initialDirection: int
+ hasMonastery: boolean
+ isArmed: boolean
+ isJoined: boolean
+ featureCodeList: int[]
+ setFeatureCode(int featureCode, int segmentIdx): void
+ rotate(): void
1
0...5
Segment
+ scoreValue: int
+ isCompleted(Board board, Tile curTile, int segmentIdx): boolean
+ score(Board board, Set<Integer> merged, Set<List<Integer>> allTilesIdx): int
+ checkRoadOrCityCompletion(Board board, Tile curTile, int segmentIdx)
1
1...*
Feature
+ type: Segment
+ meeple: int[]
+ isScored: boolean
+ isMonasteryComp: boolean
+ featureCode: int
+ containedTiles: List<Integer[]>
+ setMeepleIdx(int meepleNum, int playerId): void
+ setMonastery(): void
+ setIsScored(): void
+ setFeatureCode(int featureCode): void
+ setMeeple(int meepleNum): void
+ addTile(int row, int col): void
Notes
Note
Segment class is an enum class with four types, including CITY, MONASTERY, ROAD and FIELD.
Also, there would be abstract methods, including checking completion and scoring methods, which is very easy to override by implementing different strategies, looks concise but make sense.
Also, in my implementation idea, the way to check road and city is exactly the same, just by checking the edges of tiles that abut with vacancies.

Rationale (Updated)

My design goals for this homework are fulfilled functional correctness of Carcassonne game, strong flexibility of my implementation and the proper reusability on different platform. Also, to make it more efficiency, I specially use data structures that not occupy too much memory.

In the design principles, I mainly focus on the information hiding, code reuse, polymorphism and extensibility. I make almost all the fields and method to be private or package private, implement many getter and setter method, also mind the defensive copy to guarantee the encapsulation idea and the unchangeability of original data. I integrate so much code into helper methods to avoid much duplicated code, and also use abstract methods to realize the code reuse. For some classes, I designed interface, like the ImageOperation class, to use polymorphism when the concrete instance initialized. I have an Enum class for four type segments and apply check completion method and score method in an abstract way, which is much easier to override —— apply different algorithms and strategies, so it is also very convenient to extend the characteristic, like adding more different type of segments.

As you can see, I designed a new way to implementing the strategy pattern, by using the Enum. To be specific, Segment class is an enum class with four types, including CITY, MONASTERY, ROAD and FIELD. Also, there would be abstract methods, including checking completion and scoring methods, whichis very easy to override by implementing different strategies, looks concise but make sense. Also, in my implementation idea, the way to check road and city is exactly the same, just by checking the edges of tiles that abut with vacancies. One of the key advantages of Strategy pattern is its extensibility, like introducing a new Strategy is as easy as writing a new class and implementing Strategy interface. But with Enum, you can create a different Enum instance without creating a different class, which means a smaller number of classes and the same benefit of strategy pattern as using abstract class or interface.

Response.

Changes:

1. Add shield to cities in domain model
2. Add score board to players in domain model
3. Delete the concept of interfaces and abstract classes, for example, delete the keyword "extends" in domain model
4. Clarify how the game enables different scoring algorithms in object model.
5. Realize the code reuse by combing the city and road checking, also design abstract check completion and score function to implement different algorithms.
6. Players in Carcassonne class indicate that it is being treated as a model of the internal user rather than a model of the external one.
7. Show the feature list on the board in a correct way.
8. Clarify how the user specify the rotation and show how to check vacancies with abutting tile's adjacency edges segments consistency in interaction diagram 1.
9. Update second interaction diagram by showing more detailed logics combined with my real implementation. Though there are some method in my real implementation, that's because I split significant methods into small pieces for easy understanding.
10. Update rationale with more specific design principles.