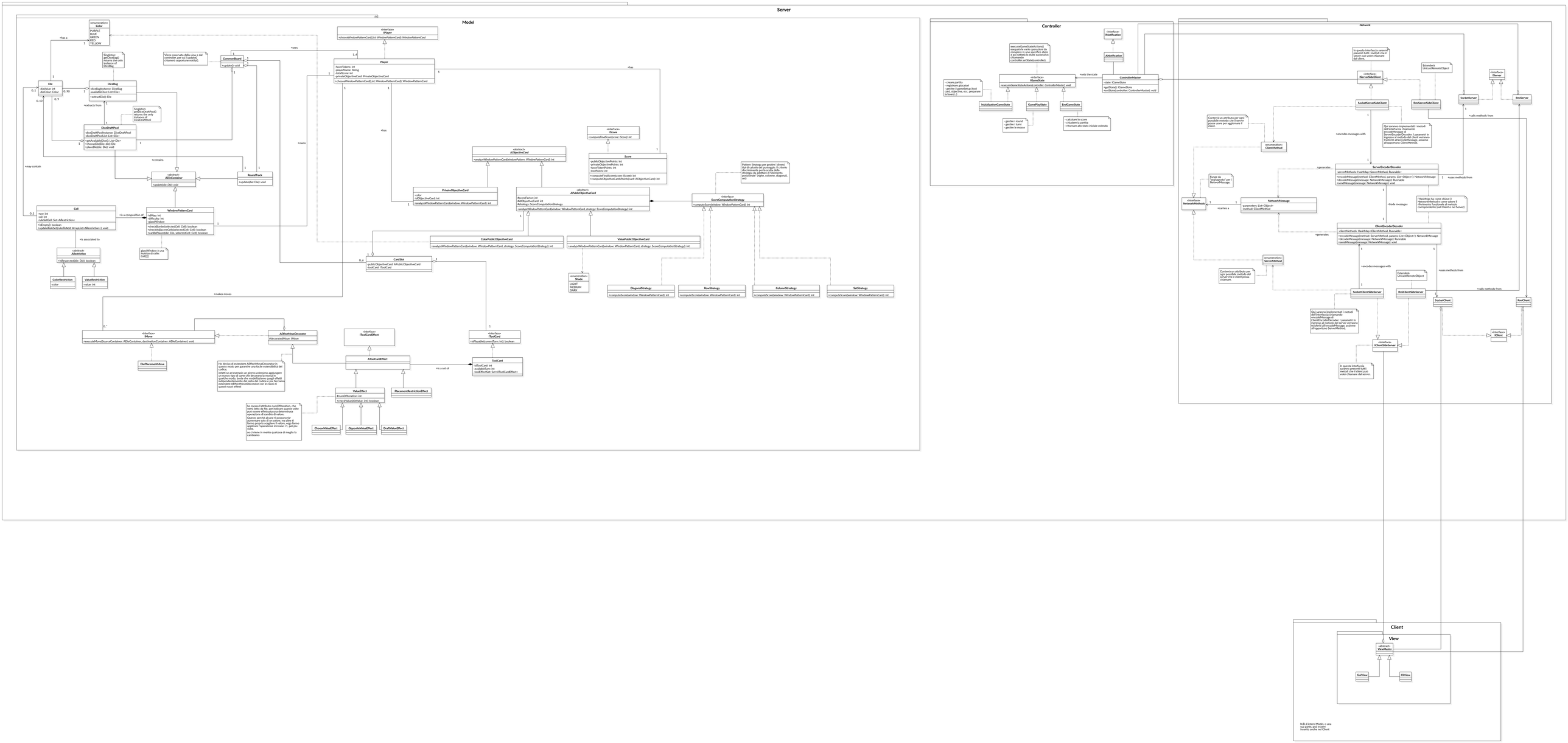
Server
Model
Controller
Network
Client
View
«enumeration»
Color
PURPLE
BLUE
GREEN
RED
YELLOW
«interface»
IPlayer
+chooseWindowPatternCard(List WindowPatternCard): WindowPatternCard
Player
#favoriTokens: int
#playerNome: String
#totalScore: int
#privateObiettivoCard: PrivateObiettivoCard
+chooseWindowPatternCard(List WindowPatternCard): WindowPatternCard
Die
-dieValue: int
-dieColor: Color
DiceBag
-diceBagInstance: DiceBag
-availableDice: List<Die>
+extractDie(): Die
DiceDraftPool
-diceDraftPool: DiceDraftPool
-diceDraftPoolList: List<Die>
+getAvailableDice(): List<Die>
+chooseDie(die: die): Die
+placeDie(dieDie: Die): void
CommonBoard
+update(): void
RoundTrack
+update(die: Die): void
AZoneContainer
+update(die: Die): void
Cell
-row: int
-col: int
-allowedCell: Set<ARestriction>
+isEmpty(): boolean
+updateAllowSetInsiduBuild: Array(List<ARestriction>) void
WindowPatternCard
-idMap: int
-difficulty: int
-glassWindow
+checkAdjacentCell(selectedCell: Cell): boolean
+checkAdjacentCellsInsideCell: Cell): boolean
+canBePlaced(die: Die, selectedCell: Cell): boolean
«interface»
IScore
+computeFinalScore(score: IScore): int
Score
-publicObjectivePoints: int
-privateObjectivePoints: int
-favorTokenPoints: int
-lostPoints: int
+computeFinalScore(score: IScore): int
+computeObjectiveCardsPoints(card: AObjectiveCard): int
«abstract»
AObjectiveCard
+analyzeWindowPatternCardwindowPattern: WindowPatternCard): int
PrivateObjectiveCard
-color
-idObjectiveCard: int
+analyzeWindowPatternCardwindow: WindowPatternCard): int
«abstract»
APublicObjectiveCard
#scoreFactor: int
#idObjectiveCard: int
#strategyScoreComputationStrategy
+analyzeWindowPatternCardwindow: WindowPatternCard, strategy: ScoreComputationStrategy): int
«interface»
IScoreComputationStrategy
+computeScoreIn(window: WindowPatternCard): int
Pattern Strategy per gestire i diversi tipi di calcolo del punteggio. È chiaro che ci sarà bisogno di una strategia da adottare è l'elemento posizionato' (righe, colonne, diagonali, set)
ColorPublicObjectiveCard
+analyzeWindowPatternCardwindow: WindowPatternCard, strategy: ScoreComputationStrategy): int
ValuePublicObjectiveCard
+analyzeWindowPatternCardwindow: WindowPatternCard, strategy: ScoreComputationStrategy): int
CardSlot
+publicObjectiveCard: APublicObjectiveCard
-tonCard: IToolCard
«enumeration»
Shade
LIGHT
MEDIUM
DARK
DiagonalStrategy
+computeScoreIn(window: WindowPatternCard): int
RowStrategy
+computeScoreIn(window: WindowPatternCard): int
ColumnStrategy
+computeScoreIn(window: WindowPatternCard): int
SetStrategy
+computeScoreIn(window: WindowPatternCard): int
«interface»
IMove
+executeMoveSourceContainer: AZoneContainer, destinationContainer: AZoneContainer): void
AEffectMoveDecorator
#decoratedMove: IMove
DiePlacementMove
Ho deciso di estendere AEffectMoveDecorator in questo modo per garantire una facile estensibilità del codice.
Infatti se ad esempio un giorno volessimo aggiungere un nuovo tipo di carte che decorano la mossa (IMove) con un effetto (AEffectMoveDecorator) indipendentemente dal resto del codice e poi facciamo estendere AEffectMoveDecorator con le classi di questi nuovi effetti
«interface»
IToolCardEffect
«interface»
IToolCard
+isPlayable(currentTurn: int): boolean
AToolCardEffect
ToolCard
-idToolCard: int
-availableTurn: int
-toolEffectSet: Set<AToolCardEffect>
+is a set of
ValueEffect
#numOfIteration: int
+checkValueable(value: int): boolean
PlacementRestrictionEffect
No meno l'attributo numOfIteration, che servirà tutto da lite, per indicare quanto volte può essere effettuata una determinata operazione di cambio di valore.
Questo perché alcune fi possono far cambiare il valore di più dadi e non di uno. Siamo proprio scegliere il valore, ergo farmo applicare l'operazione increase +1, per piu volte.
se ci viene in mente qualcosa di meglio lo cambiamo
ChooseValueEffect
OppositeValueEffect
DraftValueEffect
«interface»
IInitialization
AInitialization
executeGameStateAction() eseguirà le varie operazioni da compiere in uno specifico stato e passerà lo stato successivo (chiamando controller.setState(controller).
«interface»
IGameState
+executeGameStateAction(controller: ControllerMaster): void
+sets the state
ControllerMaster
-state: IGameState
+getState(): IGameState
+setState(controller: ControllerMaster): void
creare partita
registrare giocatori
- setup (setupSetup (tool card, objective, ecc, preparare la board ..)
InitializationGameState
GamePlayState
- gestire i round
- gestire i turni
- gestire le mosse
EndGameState
- calcolare lo score
- chiudere la partita
- riportare allo stato iniziale volendo
Questo è un'interfaccia che permette tutti i metodi che il server deve usare per aggiornare il client
«interface»
IServerSideClient
Estenderà UnicastRemoteObject
«interface»
IServer
SocketServerSideClient
RmiServerSideClient
SocketServer
RmiServer
+calls methods from
Contenrà un attributo per ogni possibile metodo che il server deve usare per aggiornare il client
«enumeration»
ClientMethod
+encodes messages with
Qui saranno implementati i metodi dell'interfaccia chiamando encodeMessage di ServerEncoderDecoder. I parametri in ingresso al metodo del client verranno trasferiti all'encodeMessage, assieme all'opportuno ClientMethod.
ServerEncoderDecoder
-serverMethods: HashMap<ServerMethod, Runnable>
+executeMessage(method: ClientMethod, params: List<Object>): NetworkMessage
+decodeMessage(message: NetworkMessage): Runnable
+sendMessage(message: NetworkMessage): void
+generates
+uses methods from
Funge da "segnaposto" per i NetworkMessage.
«interface»
NetworkMethod
+carries a
NetworkMessage
-parameters: List<Object>
-method: ClientMethod
+trade messages
Ho deciso di estendere AEffectMoveDecorator in questo modo per garantire una facile estensibilità del codice.
Infatti se ad esempio un giorno volessimo aggiungere un nuovo tipo di carte che decorano la mossa (IMove) con un effetto (AEffectMoveDecorator) indipendentemente dal resto del codice e poi facciamo estendere AEffectMoveDecorator con le classi di questi nuovi effetti
Ho deciso di estendere AEffectMoveDecorator in questo modo per garantire una facile estensibilità del codice.
Infatti se ad esempio un giorno volessimo aggiungere un nuovo tipo di carte che decorano la mossa (IMove) con un effetto (AEffectMoveDecorator) indipendentemente dal resto del codice e poi facciamo estendere AEffectMoveDecorator con le classi di questi nuovi effetti
Ho deciso di estendere AEffectMoveDecorator in questo modo per garantire una facile estensibilità del codice.
Infatti se ad esempio un giorno volessimo aggiungere un nuovo tipo di carte che decorano la mossa (IMove) con un effetto (AEffectMoveDecorator) indipendentemente dal resto del codice e poi facciamo estendere AEffectMoveDecorator con le classi di questi nuovi effetti
«enumeration»
ClientMethod
Contenrà un attributo per ogni possibile metodo del client che il client possa chiamare
ClientEncoderDecoder
-clientMethods: HashMap<ClientMethod, Runnable>
+executeMessage(method: ServerMethod, params: List<Object>): NetworkMessage
+decodeMessage(message: NetworkMessage): Runnable
+sendMessage(message: NetworkMessage): void
+encodes messages with
+uses methods from
Estenderà UnicastRemoteObject
SocketClientSideServer
RmiClientSideServer
SocketClient
RmiClient
+calls methods from
Qui saranno implementati i metodi dell'interfaccia chiamando encodeMessage di ClientEncoderDecoder. I parametri in ingresso al metodo del server verranno trasferiti all'encodeMessage, assieme all'opportuno ServerMethod.
«interface»
IClientSideServer
«interface»
IClient
In questa interfaccia saranno presenti tutti i metodi che il client deve poter chiamare dal server.
NB: L'intero Model, o una sua parte, può essere inserito anche nel Client