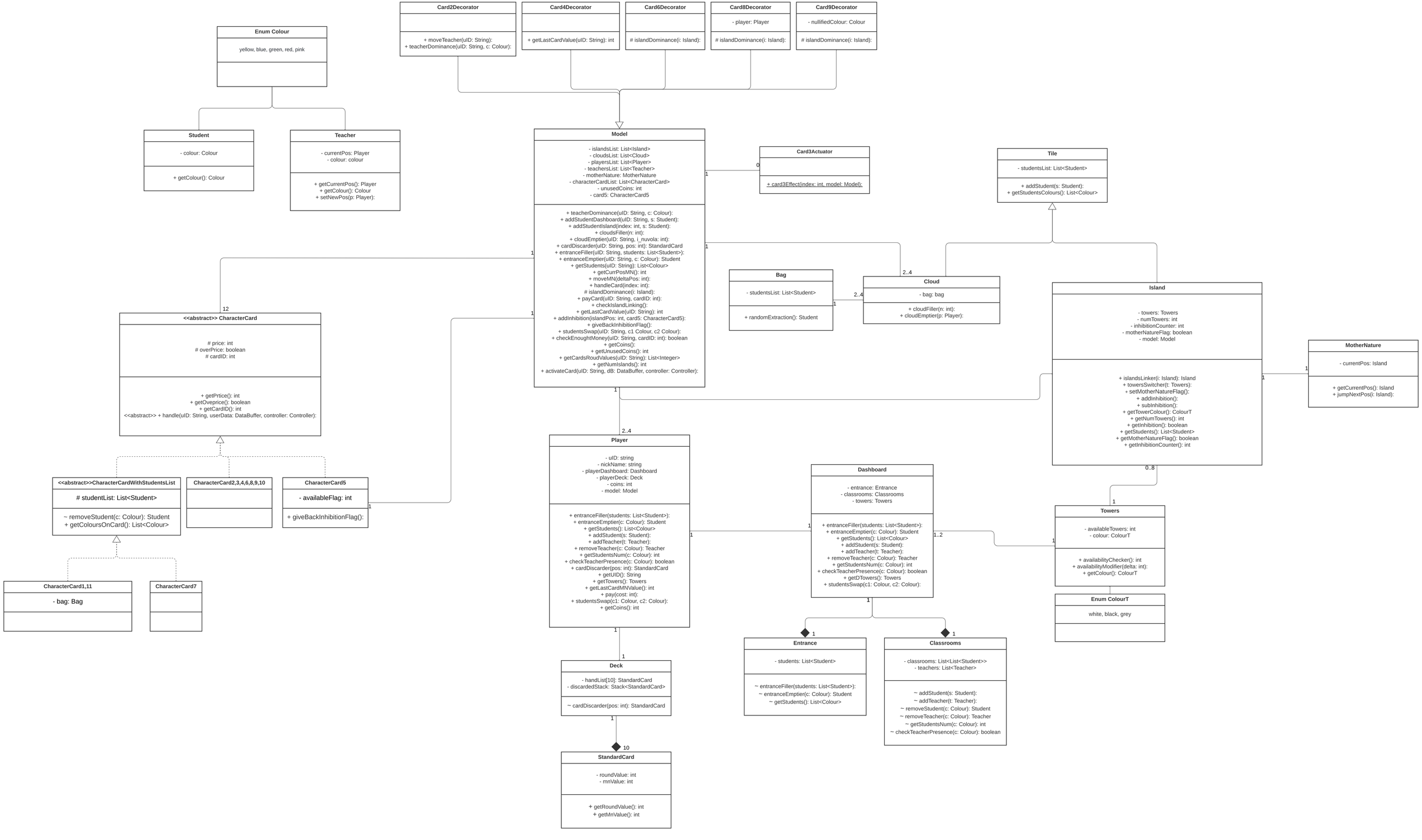
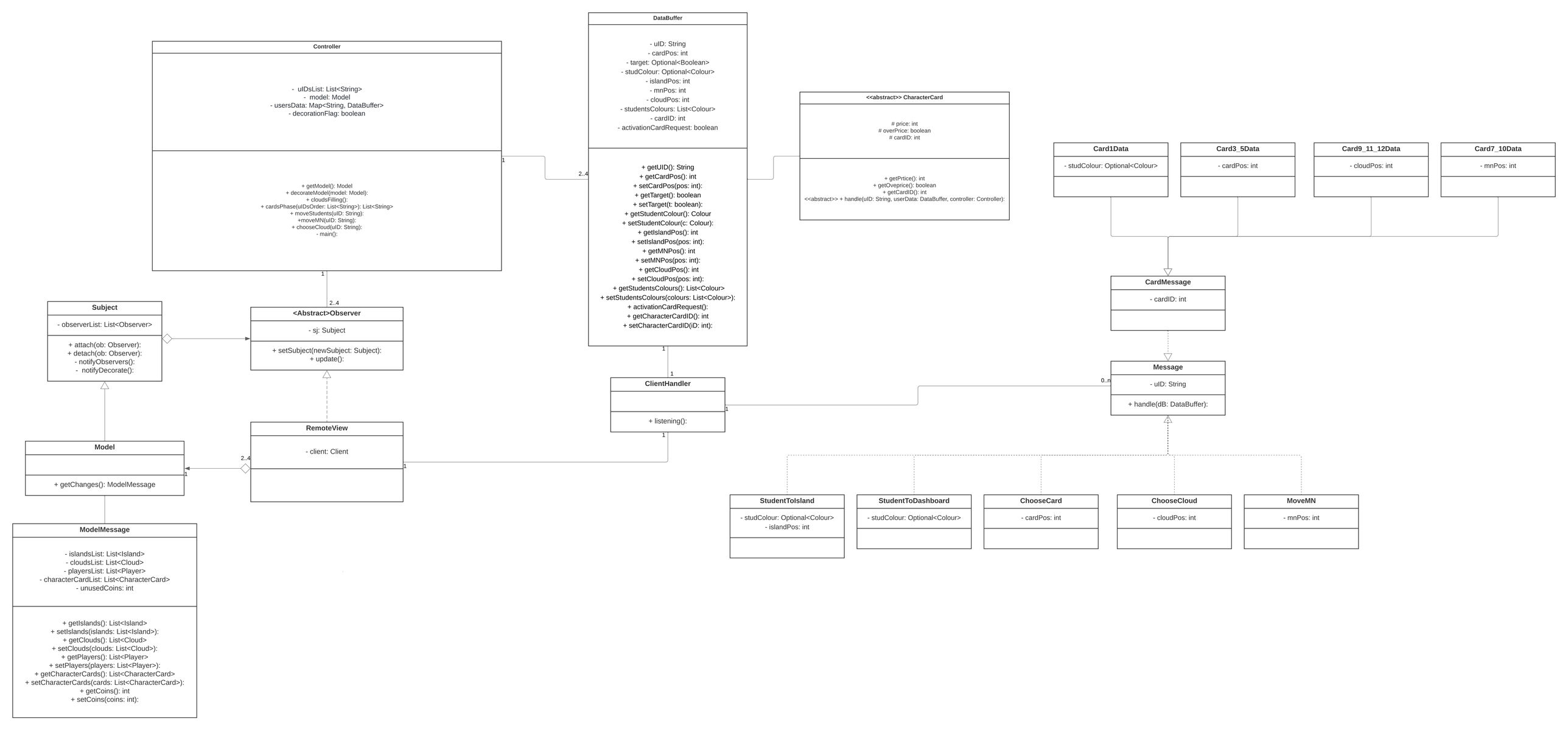

Card2Decorator
+ moveTeacher(uID: String):
+ teacherDominance(uID: String, c: Colour):
Card4Decorator
+ getLastCardValue(uID: String): int
Card6Decorator
islandDominance(i: Island):
Card8Decorator
- player: Player
islandDominance(i: Island):
Card9Decorator
- nullifiedColour: Colour
islandDominance(i: Island):
Enum Colour
yellow, blue, green, red, pink
Student
- colour: Colour
+ getColour(): Colour
Teacher
- currentPos: Player
- colour: colour
+ getCurrentPos(): Player
+ getColour(): Colour
+ setNewPos(p: Player):
Model
- islandsList: List<Island>
- cloudsList: List<Cloud>
- playersList: List<Player>
- teachersList: List<Teacher>
- motherNature: MotherNature
- characterCardList: List<CharacterCard>
- unusedCoins: int
- card5: CharacterCard5
+ teacherDominance(uID: String, c: Colour):
+ addStudentDashboard(uID: String, s: Student):
+ addStudentIsland(index: int, s: Student):
+ cloudsFiller(n: int):
+ cloudEmptier(uID: String, i, nuvola: int):
+ cardDiscarder(uID: String, pos: int): StandardCard
+ entranceFiller(uID: String, students: List<Student>):
+ entranceEmptier(uID: String, c: Colour): Student
+ getStudents(uID: String): List<Colour>
+ getCurrPosMN(): int
+ moveMN(deltaPos: int):
+ handleCard(index: int):
# islandDominance(i: Island):
+ payCard(uID: String, cardID: int):
+ checkIslandLinking():
+ getLastCardValue(uID: String): int
+ addInhibition(islandPos: int, card5: CharacterCard5):
+ giveBackInhibitionFlag():
+ studentsSwap(uID: String, c1 Colour, c2 Colour):
+ checkEnoughtMoney(uID: String, cardID: int): boolean
+ getCoins():
+ getUnusedCoins(): int
+ getCardsRoudValues(uID: String): List<Integer>
+ getNumIslands(): int
+ activateCard(uID: String, dB: DataBuffer, controller: Controller):
Card3Actuator
+ card3Effect(index: int, model: Model):
Tile
- studentsList: List<Student>
+ addStudent(s: Student):
+ getStudentsColours(): List<Colour>
Bag
- studentsList: List<Student>
+ randomExtraction(): Student
Cloud
- bag: bag
+ cloudFiller(n: int):
+ cloudEmptier(p: Player):
Island
- towers: Towers
- numTowers: int
- inhibitionCounter: int
- motherNatureFlag: boolean
- model: Model
+ islandsLinker(i: Island): Island
+ towersSwitcher(t: Towers):
+ setMotherNatureFlag():
+ addInhibition():
+ subInhibition():
+ getTowerColour(): ColourT
+ getNumTowers(): int
+ getInhibition(): boolean
+ getStudents(): List<Student>
+ getMotherNatureFlag(): boolean
+ getInhibitionCounter(): int
MotherNature
- currentPos: Island
+ getCurrentPos(): Island
+ jumpNextPos(i: Island):
<<abstract>> CharacterCard
price: int
overPrice: boolean
cardID: int
+ getPrtice(): int
+ getOveprice(): boolean
+ getCardID(): int
<<abstract>> + handle(uID: String, userData: DataBuffer, controller: Controller):
<<abstract>>CharacterCardWithStudentsList
studentList: List<Student>
~ removeStudent(c: Colour): Student
+ getColoursOnCard(): List<Colour>
CharacterCard2,3,4,6,8,9,10
CharacterCard5
- availableFlag: int
+ giveBackInhibitionFlag():
CharacterCard1,11
- bag: Bag
CharacterCard7
Player
- uID: string
- nickName: string
- playerDashboard: Dashboard
- playerDeck: Deck
- coins: int
- model: Model
+ entranceFiller(students: List<Student>):
+ entranceEmptier(c: Colour): Student
+ getStudents(): List<Colour>
+ addStudent(s: Student):
+ addTeacher(t: Teacher):
+ removeTeacher(c: Colour): Teacher
+ getStudentsNum(c: Colour): int
+ checkTeacherPresence(c: Colour): boolean
+ cardDiscarder(pos: int): StandardCard
+ getUID(): String
+ getTowers(): Towers
+ getLastCardMNValue(): int
+ pay(cost: int):
+ studentsSwap(c1: Colour, c2: Colour):
+ getCoins(): int
Dashboard
- entrance: Entrance
- classrooms: Classrooms
- towers: Towers
+ entranceFiller(students: List<Student>);
+ entranceEmptier(c: Colour): Student
+ getStudents(): List<Colour>
+ addStudent(s: Student):
+ addTeacher(t: Teacher):
+ removeTeacher(c: Colour): Teacher
+ getStudentsNum(c: Colour): int
+ checkTeacherPresence(c: Colour): boolean
+ getDTowers(): Towers
+ studentsSwap(c1: Colour, c2: Colour):
Towers
- availableTowers: int
- colour: ColourT
+ availabilityChecker(): int
+ availabilityModifier(delta: int):
+ getColour(): ColourT
Enum ColourT
white, black, grey
Deck
- handList[10]: StandardCard
- discardedStack: Stack<StandardCard>
~ cardDiscarder(pos: int): StandardCard
StandardCard
- roundValue: int
- mnValue: int
+ getRoundValue(): int
+ getMnValue(): int
Entrance
- students: List<Student>
~ entranceFiller(students: List<Student>):
~ entranceEmptier(c: Colour): Student
~ getStudents(): List<Colour>
Classrooms
- classrooms: List<List<Student>>
- teachers: List<Teacher>
~ addStudent(s: Student):
~ addTeacher(t: Teacher):
~ removeStudent(c: Colour): Student
~ removeTeacher(c: Colour): Teacher
~ getStudentsNum(c: Colour): int
~ checkTeacherPresence(c: Colour): boolean


Controller
- uIDsList: List<String>
- model: Model
- usersData: Map<String, DataBuffer>
- decorationFlag: boolean
+ getModel(): Model
+ decorateModel(model: Model):
+ cloudsFilling():
+ cardsPhase(uIDsOrder: List<String>): List<String>
+ moveStudents(uID: String):
+moveMN(uID: String):
+ chooseCloud(uID: String):
- main():
DataBuffer
- uID: String
- cardPos: int
- target: Optional<Boolean>
- studColour: Optional<Colour>
- islandPos: int
- mnPos: int
- cloudPos: int
- studentsColours: List<Colour>
- cardID: int
- activationCardRequest: boolean
+ getUID(): String
+ getCardPos(): int
+ setCardPos(pos: int):
+ getTarget(): boolean
+ setTarget(t: boolean):
+ getStudentColour(): Colour
+ setStudentColour(c: Colour):
+ getIslandPos(): int
+ setIslandPos(pos: int):
+ getMNPos(): int
+ setMNPos(pos: int):
+ getCloudPos(): int
+ setCloudPos(pos: int):
+ getStudentsColours(): List<Colour>
+ setStudentsColours(colours: List<Colour>):
+ activationCardRequest():
+ getCharacterCardID(): int
+ setCharacterCardID(iD: int):
<<abstract>> CharacterCard
price: int
overPrice: boolean
cardID: int
+ getPrtice(): int
+ getOveprice(): boolean
+ getCardID(): int
<<abstract>> + handle(uID: String, userData: DataBuffer, controller: Controller):
Card1Data
- studColour: Optional<Colour>
Card3_5Data
- cardPos: int
Card9_11_12Data
- cloudPos: int
Card7_10Data
- mnPos: int
CardMessage
- cardID: int
Message
- uID: String
+ handle(dB: DataBuffer):
Subject
- observerList: List<Observer>
+ attach(ob: Observer):
+ detach(ob: Observer):
- notifyObservers():
- notifyDecorate():
<Abstract>Observer
- sj: Subject
+ setSubject(newSubject: Subject):
+ update():
ClientHandler
+ listening():
RemoteView
- client: Client
Model
+ getChanges(): ModelMessage
ModelMessage
- islandsList: List<Island>
- cloudsList: List<Cloud>
- playersList: List<Player>
- characterCardList: List<CharacterCard>
- unusedCoins: int
+ getIslands(): List<Island>
+ setIslands(islands: List<Island>):
+ getClouds(): List<Cloud>
+ setClouds(clouds: List<Cloud>):
+ getPlayers(): List<Player>
+ setPlayers(players: List<Player>):
+ getCharacterCards(): List<CharacterCard>
+ setCharacterCards(cards: List<CharacterCard>):
+ getCoins(): int
+ setCoins(coins: int):
StudentToIsland
- studColour: Optional<Colour>
- islandPos: int
StudentToDashboard
- studColour: Optional<Colour>
ChooseCard
- cardPos: int
ChooseCloud
- cloudPos: int
MoveMN
- mnPos: int
1
2..4
0..n

Controller e Rete

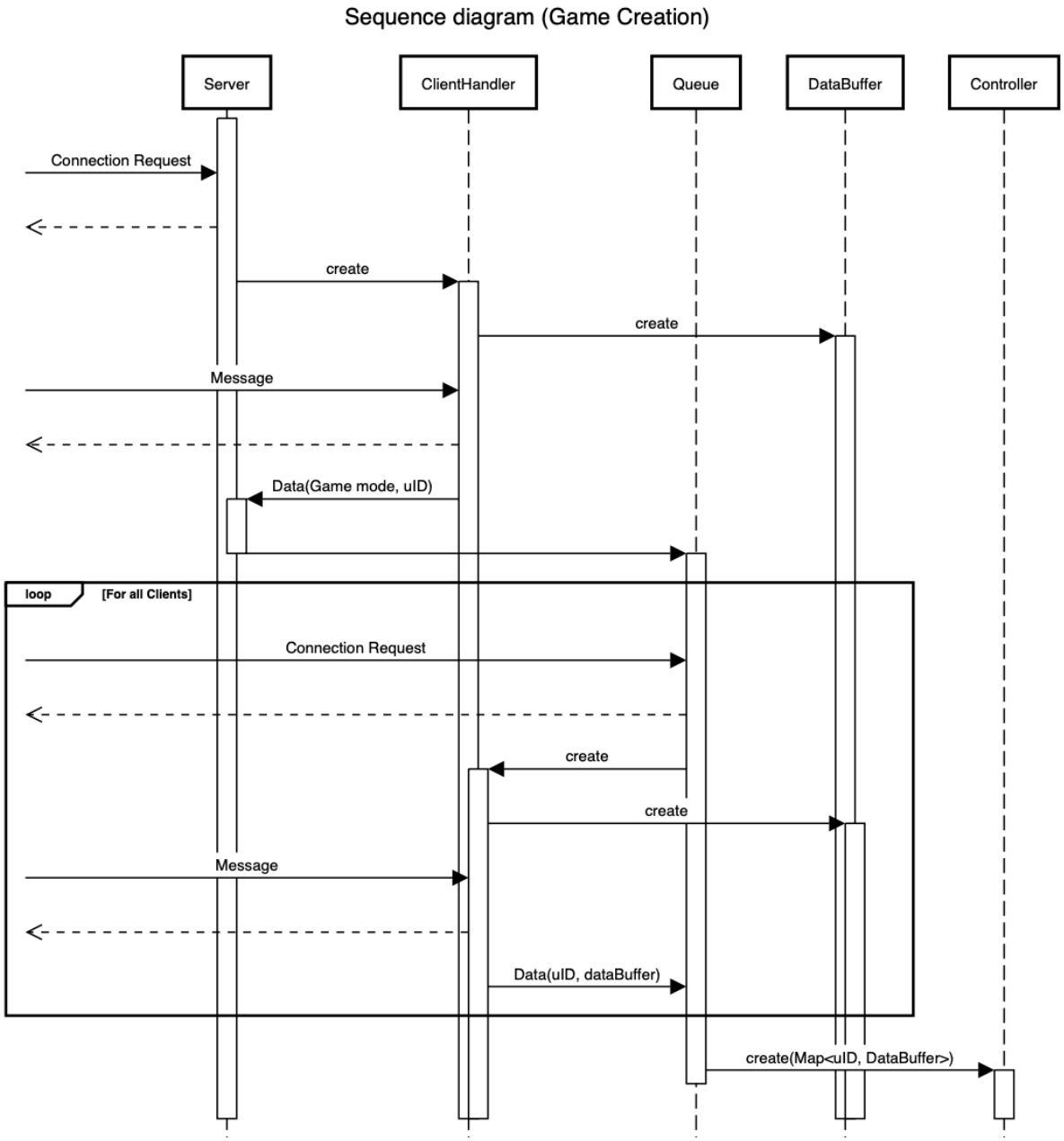
<Francesco Maria Tranquillo>, <Denis Previtali>, <Andrea Zatti>
Gruppo <GC37>

3 maggio 2022

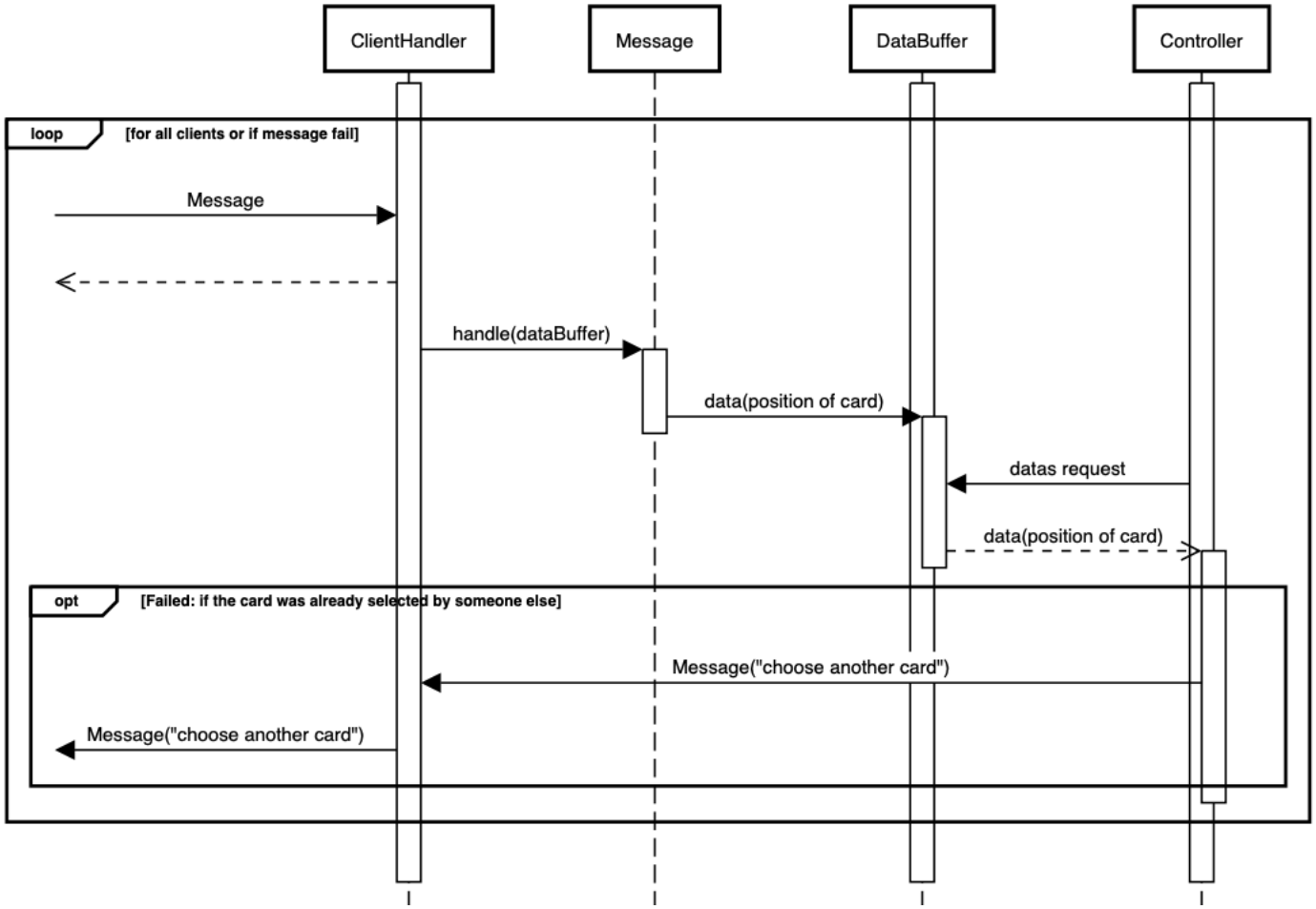
Descrizione del diagramma UML del controller.

Il controller è composto da una funzione main (mentre per ogni ClientHandler verrà dedicato un thread a parte) che viene eseguita da un thread, nel seguente metodo main abbiamo una serie di chiamate agli altri metodi in modo da rispettare l'ordine dell'andamento della partita. In questi metodi grazie all'attributo privato di tipo mappa (che associa un ClientHandler al suo DataBuffer) attendiamo che ogni ClientHandler riempia il suo DataBuffer per poter procedere con l'esecuzione del metodo. I metodi getModel e decorateModel sono utilizzati dalle character cards.
Il DataBuffer è una classe che fa da intermediario tra ClientHandler e controller, il quale viene passato come attributo al metodo dei messaggi "handle()" che si occuperà di riempire tramite i metodi setter il DataBuffer il quale poi li fornirà al controller non appena verranno richiesti tramite i getter. I getter se vengono chiamati prima che il dato richiesto sia stato fornito dal ClientHandler si mettono in attesa; quindi alla fine dei setter ci sarà una notifyAll. Anche le character cards si collegano al DataBuffer per estrarne i dati necessari.
I messaggi in arrivo dai ClientHandler sulla rete saranno quindi quelli che riempiranno il DataBuffer, mentre l'unico messaggio in uscita dal server sarà quello che contiene le modifiche del model.
Per inviare le modifiche del model facciamo uso del pattern observer(che successivamente implementeremo con dei listener), il model erediterà quindi da una sovraclasse chiamata subject la quale predispone i metodi necessari per realizzare il pattern. Inoltre sul server abbiamo anche una classe chiamata RemoteView che eredita da una classe astratta Observer e si collega alla classe

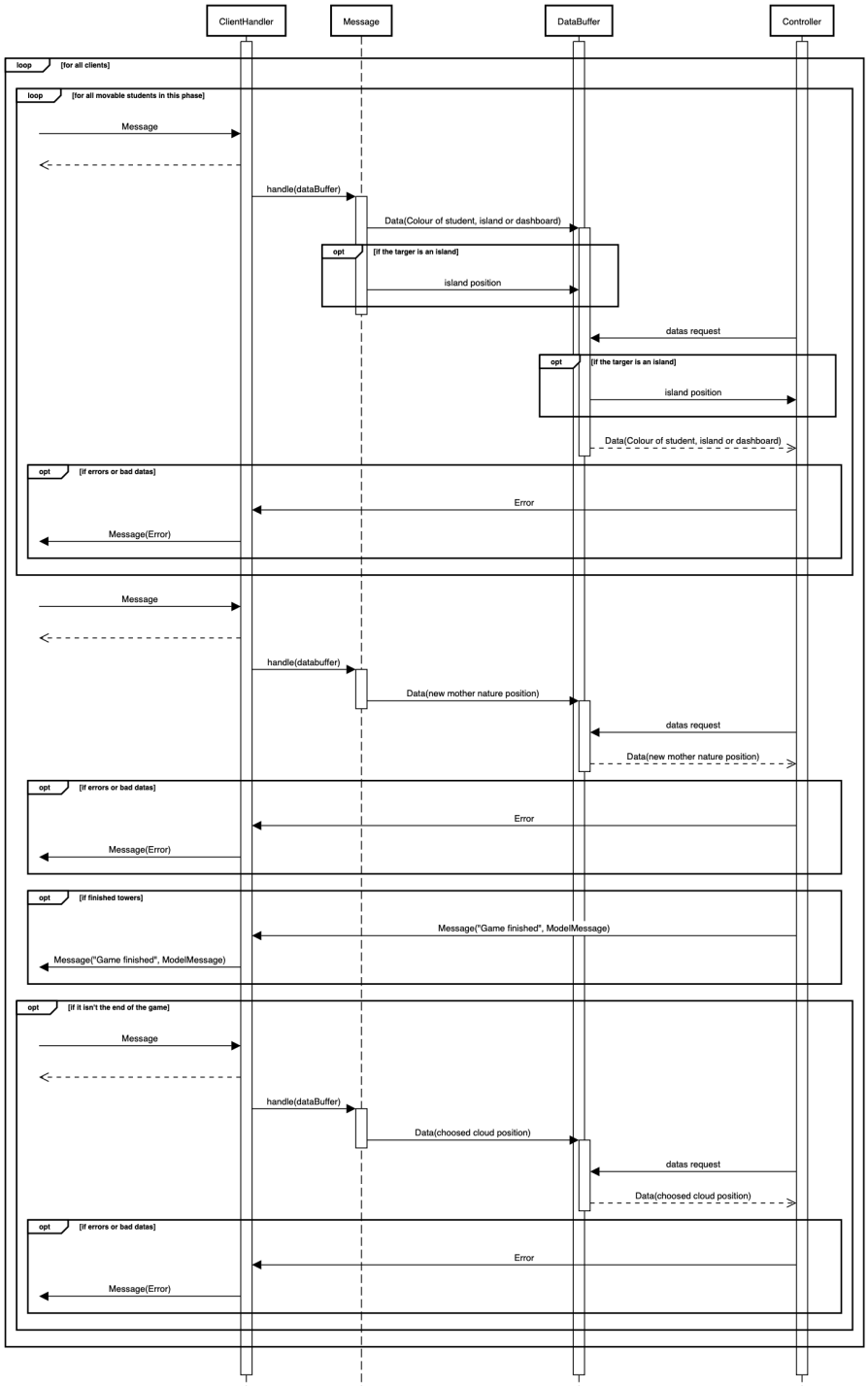
ClientHandler, questo per poter comunicare ai client i cambiamenti del model tramite un messaggio che ne contiene solo gli oggetti variati(ModelMessage). Di seguito lasciamo i sequence diagrams dei messaggi scambiati sulla rete.


Sequence diagram (Game Creation)
Server
ClientHandler
Queue
DataBuffer
Controller
Connection Request
create
create
Message
Data(Game mode, uID)
loop
[For all Clients]
Connection Request
create
create
Message
Data(uID, dataBuffer)
create(Map<uID, DataBuffer>)

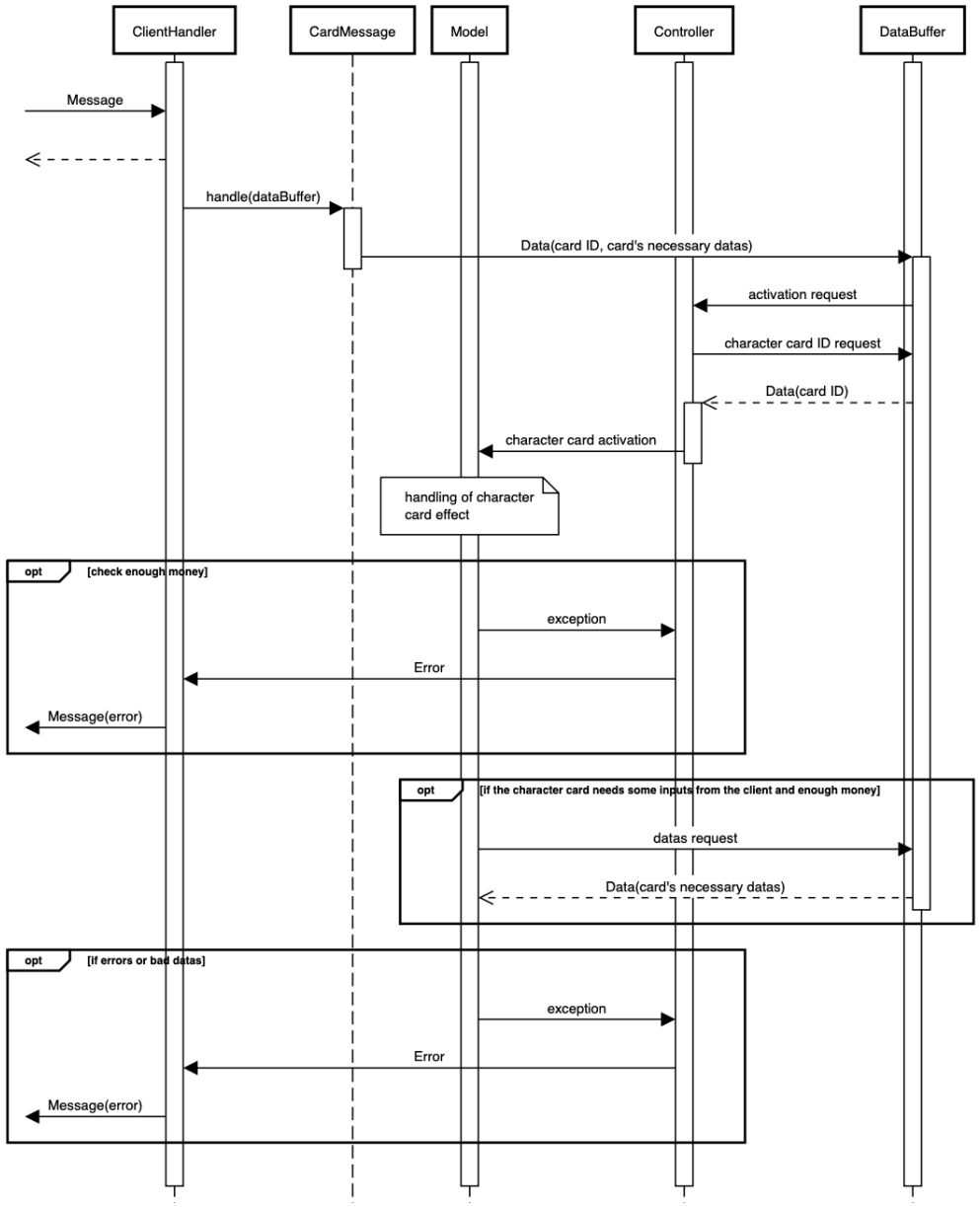
Sequence diagram(card phase)

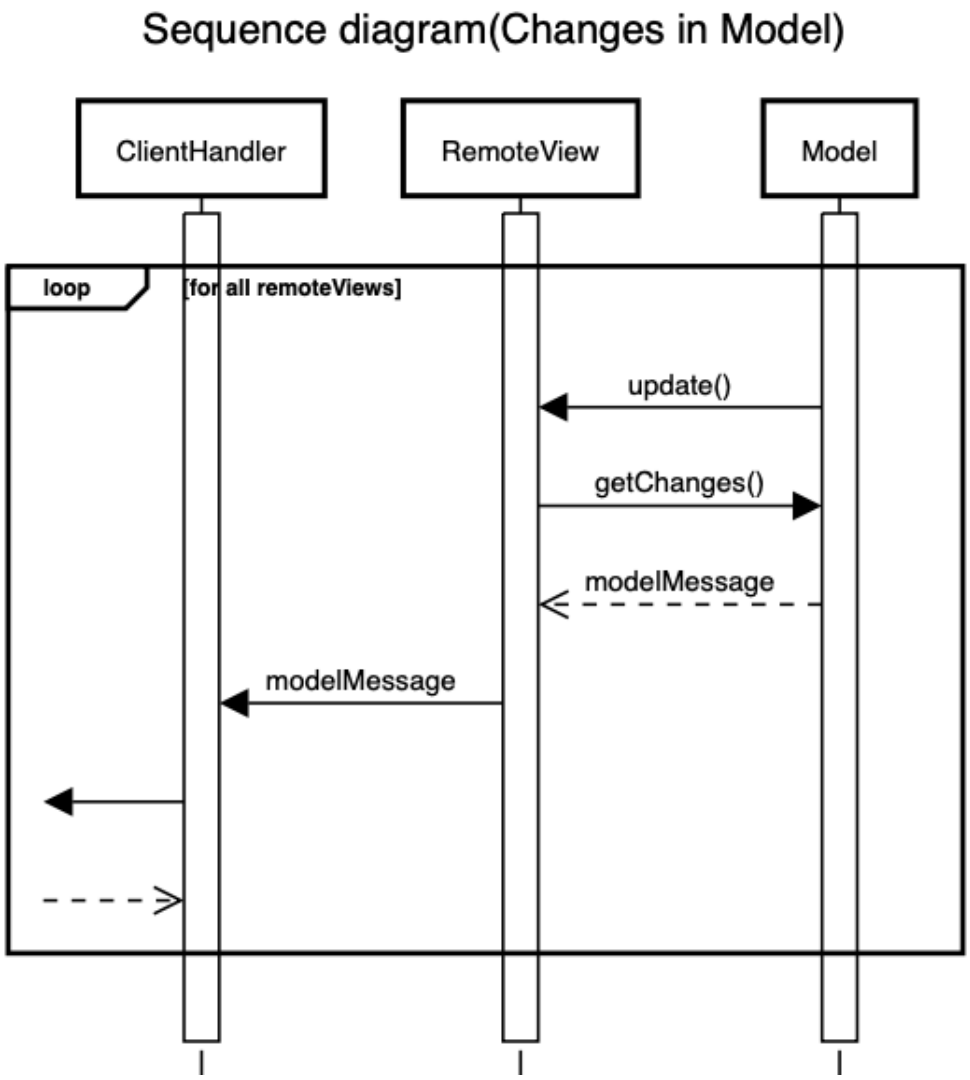


Sequence diagram(Action Phase)



Sequence diagram(character card activation)




Sequence diagram(Changes in Model)
ClientHandler
RemoteView
Model
loop
[for all remoteViews]
update()
getChanges()
modelMessage
modelMessage