Concordia University

Department of Computer Science and Software Engineering

Advanced Programming Practices
SOEN 6441 – Winter 2020

Software Architecture & Methodologies

Team **Java Bean**

Jatan Gohel	4007 8112
Siddhant Arora	4008 5538
Krunal Jagani	4005 6939
Harsh Vaghani	4008 4099

Guided by

Dr. Rodrigo Morales

Architectural Design Document

Introduction

Our purpose is to build the Monopoly game from the game framework we have developerd in the first build. Which follows a Model View Controller (MVC) architectural design model. We have studied and made sure we follow the extreme programming approach for the smooth development of software by implementing features like:

- **Planning:** Since from the 1st build, our goal was to create our framework as flexible and rich as possible so that it would become easier for us to do game implementation less hard. First we decided to create a list of multiple reusable modules and build our architecture of 2nd build around them.

- **Testing:** Testing is done by writing unit test cases for each modules and determined the veracity of the code we have developed. We have used Junit framework for this purpose.

- **Collective Ownership:** Distributed work among group member for different modules, saved changes into different branches and then later on merged all of them into local branch.

- **Simple Design:** Complex designs were avoided in order to make the software more user-friendly, and to also avoid faults. Smooth features were implemented with the help of Simple designs which resulted in quick, structured and clear game flow.

- **Continuous Integration:** Git was used as Revision Version Control which made all the group members up to date. Maintenance of concurrent changes, rollback sequences were maintained through local branches and master branch. Errors and conflicts detected at a time of integration were rectified quickly.

Architecture :

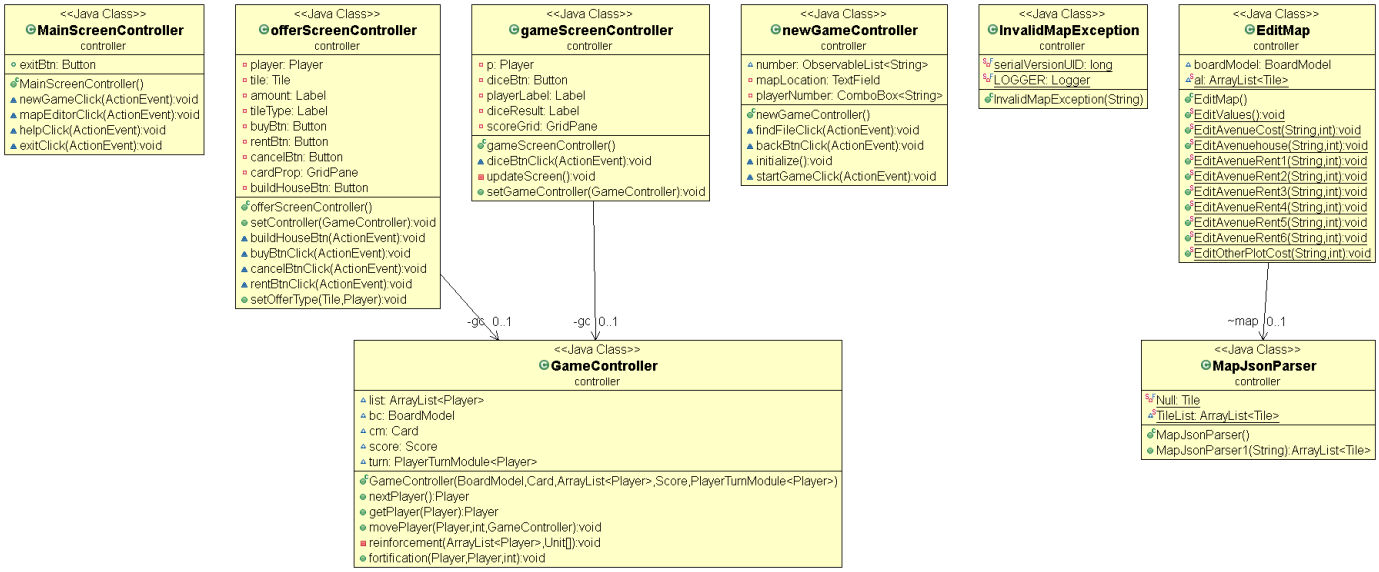
Since our first build was based on MVC architecture , we decided to continue working in the same architecture after doing some amount of refactoring to facilitate future needs of the second build.

Apart from that observer design pattern was implemented in the GameController where game controller will notify on a successful dice roll.

```
private static ArrayList<GameControlObserver> observers = new ArrayList<>();

private GameController gc;
private Player p;

public static void addObserver(GameControlObserver gameControlObserver) {
        observers.add(gameControlObserver);
}

public static void removeObserver(GameControlObserver gameControlObserver) {
        observers.remove(gameControlObserver);
}
```

Main screen View (new files added):

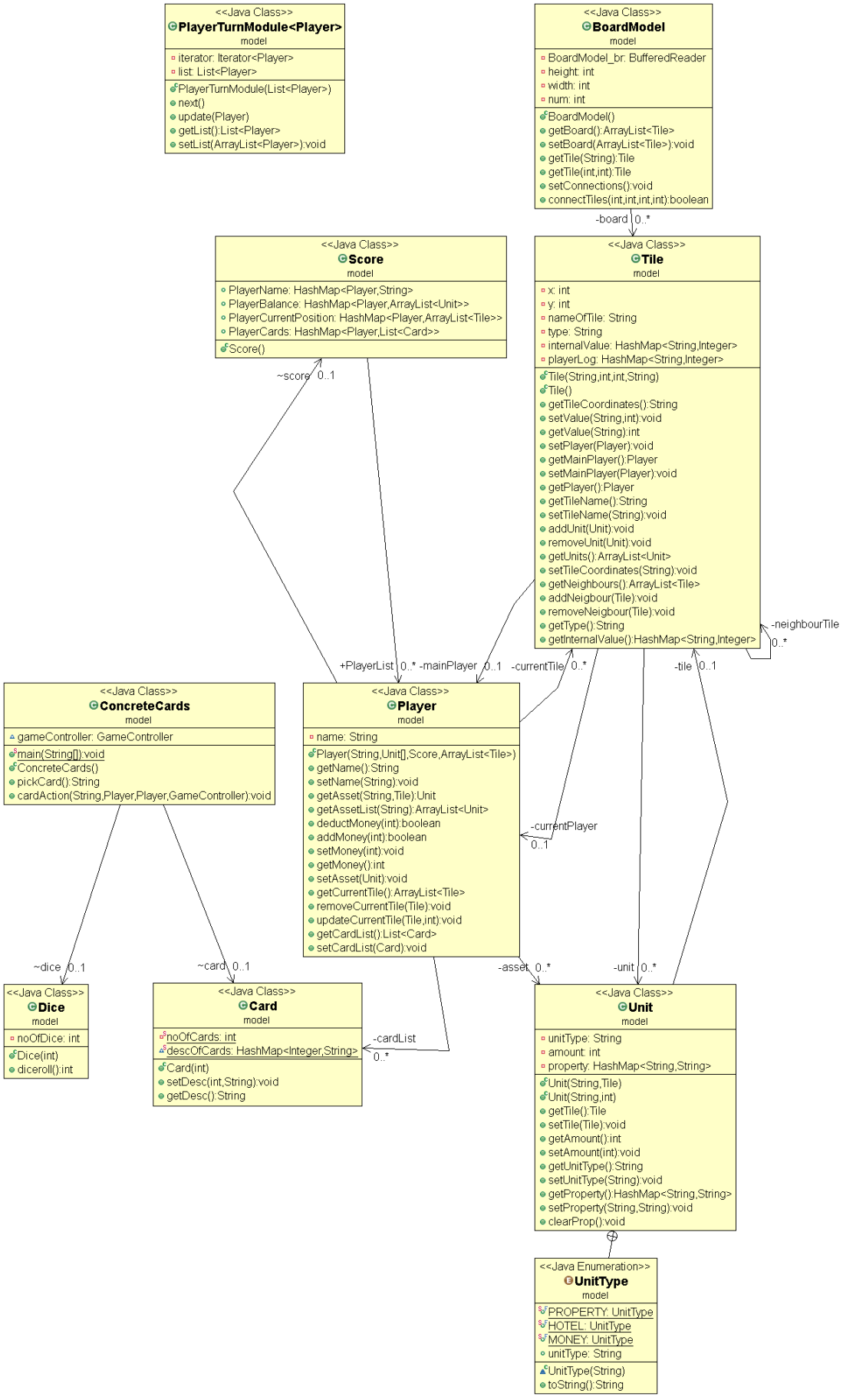
Controller (new files added) :



1. Edit map : Edit map has functionality to edit values of different tiles based on their properties. It has some security constraints. Such as end user can not change the names and type of properties because in monopoly each type of tile serves its purpose. If the property is Avenue then , an end user has the permission to edit its rents, house and cost prices.
2. Map file parser : Map file parser parses data in the form of Json format .Every object is handled individually as a matter of the fact that each plot has a different property. For instance, Avenue plots has different property than go and go has different property than railroads. So each objects had to be treated differently.
3. Invalid map file exception : It is used to generate exceptions when invalid map is passed as the parameter to the game.
4. Game Screen Controller has methods to control specific actions while interacting with the main screen. It implements observer design pattern to update screen.
5. OfferScreenController : It handles different interaction between objects such as cards, players, turn money transfer. For instance, If a player lands on a card, a card will be picked randomly and method inside it will get executed.

Models (new files added) :

- Concrete cards : Concrete implementation of the cards modules where we defined different community chest cards.


<<Java Class>>
PlayerTurnModule<Player>
model
iterator: Iterator<Player>
list: List<Player>
PlayerTurnModule(List<Player>)
next()
update(Player)
getList():List<Player>
setList(ArrayList<Player>):void
<<Java Class>>
BoardModel
model
BoardModel_br: BufferedReader
height: int
width: int
num: int
BoardModel()
getBoard():ArrayList<Tile>
setBoard(ArrayList<Tile>):void
getTile(String):Tile
getTile(int,int):Tile
setConnections():void
connectTiles(int,int,int,int):boolean
-board 0..*
<<Java Class>>
Score
model
PlayerName: HashMap<Player,String>
PlayerBalance: HashMap<Player,ArrayList<Unit>>
PlayerCurrentPosition: HashMap<Player,ArrayList<Tile>>
PlayerCards: HashMap<Player,List<Card>>
Score()
~score 0..1
<<Java Class>>
Tile
model
x: int
y: int
nameOfTile: String
type: String
internalValue: HashMap<String,Integer>
playerLog: HashMap<String,Integer>
Tile(String,int,int,String)
Tile()
getTileCoordinates():String
setValue(String,int):void
getValue(String):int
setPlayer(Player):void
getMainPlayer():Player
setMainPlayer(Player):void
getPlayer():Player
getTileName():String
setTileName(String):void
addUnit(Unit):void
removeUnit(Unit):void
getUnits():ArrayList<Unit>
setTileCoordinates(String):void
getNeighbours():ArrayList<Tile>
addNeigbour(Tile):void
removeNeigbour(Tile):void
getType():String
getInternalValue():HashMap<String,Integer>
-neighbourTile
0..*
+PlayerList 0..*
-mainPlayer 0..1
-currentTile 0..*
-tile 0..1
<<Java Class>>
ConcreteCards
model
gameController: GameController
main(String[]):void
ConcreteCards()
pickCard():String
cardAction(String,Player,Player,GameController):void
<<Java Class>>
Player
model
name: String
Player(String,Unit[],Score,ArrayList<Tile>)
getName():String
setName(String):void
getAsset(String,Tile):Unit
getAssetList(String):ArrayList<Unit>
deductMoney(int):boolean
addMoney(int):boolean
setMoney(int):void
getMoney():int
setAsset(Unit):void
getCurrentTile():ArrayList<Tile>
removeCurrentTile(Tile):void
updateCurrentTile(Tile,int):void
getCardList():List<Card>
setCardList(Card):void
-currentPlayer
0..1
~dice 0..1
~card 0..1
-asset 0..*
-unit 0..*
<<Java Class>>
Dice
model
noOfDice: int
Dice(int)
diceroll():int
<<Java Class>>
Card
model
noOfCards: int
descOfCards: HashMap<Integer,String>
Card(int)
setDesc(int,String):void
getDesc():String
-cardList
0..*
<<Java Class>>
Unit
model
unitType: String
amount: int
property: HashMap<String,String>
Unit(String,Tile)
Unit(String,int)
getTile():Tile
setTile(Tile):void
getAmount():int
setAmount(int):void
getUnitType():String
setUnitType(String):void
getProperty():HashMap<String,String>
setProperty(String,String):void
clearProp():void
<<Java Enumeration>>
UnitType
model
PROPERTY: UnitType
HOTEL: UnitType
MONEY: UnitType
unitType: String
UnitType(String)
toString():String

Resources (new files added) :

- Valid Json map file : Valid file with correct data information

  For instance here is an object of valid tile:

```
{
  "a_name": "Mediterranean Avenue",
  "b_type": "property",
  "c_x": "9",
  "d_y": "0",
  "e_cost": "60",
  "f_rent": "[2,10,30,90,160,250]",
  "g_house": "50"
},
```

- Invalid Json map file : Invalid map with few dummy data to check the validity of the map file

```
{
  "a_name": "Mediterranean Avenue",
  "b_type": "123",
  "c_x": "9.8",
  "d_y": "0",
  "e_cost": "60",
  "f_rent": "[2,10,30,90,160,250,500,600,100,1000]",
  "g_house": "50"
},
```

Refactoring :

Since, It was difficult to anticipate all the functional requirement of the second build in the game framework we developed in the first build, We had to perform some refactoring to make those module usable for the upcoming build. Since most of the modules were reused in their initial format , the refactoring task was not much time consuming. Some of the basic refactoring we did, includes Addition of getters and setters, modification in the method / variable visibility , changes in the constructor, changes in the method working.

Here are few the instances where we had to do refactoring :

1. Getter and setters addition in Tile class for typeOftile variable.The map file we have taken into reference has a variable called Property which distinguishes between different types of tile. We had created typeOftile variable but we had not encapsulated the variable inside the tile class. So, we added getters and setters for the same variable and passed the object reference to the mapfile parser class in order to parse property variable of the tile.

2. Static reference modification : Some locally declared variables were reused in the second build but we could not pass the reference of these variables in the current class so we had to declare those variables either globally using static keyword or using public modifier so that they can be reused by different classes of the same package or different package.

3. We are no longer passing entire list to the playerturn module. The reason is we have derived banker as an object from the player module itself. So, sending entire player list would include Banker object as well and Banker object does not get its turn in actual game playing. Banker has been fixed at the index 0.

4. In board framework we have derived army type as a Unit in Unit module. Since, Monopoly game does not have Army we have refactored it to Property of the plot.

References

- http://www.oracle.com/technetwork/articles/javase/index-142890.html

- https://mdn.mozillademos.org/files/16042/model-view-controller-light-blue