Design document

Final Design Explanation

In our final design, the program is still build around the “big three” pivotal objects -- Board, Cell, and Player. Attempting to make things easy to track and modify, we have choose to channel all actions through a class called Behavior; in this sense, the Behavior communicates between the “big three” and performs the most basic functionality. However, the events in cells are actually compound events (combination of multiple basic functionality). Hence, the Behavior class is also related to the concrete decorators of Event (such as Tuition and TimLine). Next up, we move on the player strategy/logic; so we created this Strategy class that prompts human players for decisions. The Strategy class works in the following mechanism: whenever a player is asked for decision (from Events or Cell or Board ...), the player will call the getStrategy() method which takes in Player’s decision through stdin and output the the respective class(one that asked for decision).

To this point, after getting done the five most critical Objects to game playing, we need only to complete the remaining peripheral/supportive Objects like -- TextDisplay, Dice, property and so on.

Deviation from original design

Initially, the defined roles for between the “big three”, Strategy, and Behavior are not so clear cut. Some of the functions are still not uniformed in the sense of operation mechanism (board controls the playround() whereas for decisions, Behavior will be called). Now after converging all possible actions to Behavior, we have set up the mechanism as following:

Strategy

Behavior

Board Cell Player

Each time there needs an action, the program moves upward from the “big three” to Behavior, which will again call Strategy for command. Then upon receiving the command, changes will affect the “big three” through Behavior. So in this sense, behavior is the center-piece of this mechanism.

Explanation of design pattern used

Decorator pattern: between Event and its concrete decorators, like buyProperty and TimLine.

Observer pattern: between Behavior and Cell, between Behavior and Board, between Behavior and Player.

Singleton: Board, Dice, XDisplay, Behavior,

Q & A (Due date 1)

Question 1

After reading this subsection, would be Observer Pattern be a good pattern to use when implementing a game board? Why or why not?

Original answer:

Because at all times there could exist only one board that contains all the actions, we think the board should be implemented at a singleton associating with two fundamental objects, Player and Cell. However observer pattern exists between these three objects. A board can notify Players (one-way) and Players can notify/be notified by a Cell.

New answer:

We still stick to our original idea about implementing the observer pattern. As it turns out, a observer relationship between the "big three"(Cell, Board, Player) and behavior makes the changes to these three objects very easy to keep track of and edit.

Question 2

Suppose that we wanted to model SLC and Needles Hall more closely to Chance and Community Chest cards. Is there a suitable design pattern you could use? How would you use it?

Original answer:

We have implemented the event as a decorator that houses concrete decorators such as collectRent(for collecting tuition) or drawCards(for SLC and Needles Hall). And between Player and Event, there is a observer pattern in place such that when a event happens, it affects the corresponding players and in so, player is an observer to the ongoing event.

New answer:

Right now, all changes to the players (cash or assets) are through Behavior. Hence to model SLC and NH more closely to the actual game, and now Player is an Observer to Behavior and various events merely put in place multiple Behaviors together to achieve the respective results. In this sense, we feel the modeling is neater and also more closely resemble the Chance and Community Chest cards than our previous design. This is largely because now we can affect players not just

their money, but also give/take away their properties through Behavior (which does this through concrete events).

Question 3

What could you do to ensure there are never more than 4 Roll Up the Rim cups?

Original answer:

Giving or retrieving the cups are all done through the behavior object, and the behavior class is implemented as a singleton. Thus to ensure that at all times during a game there are at most four cups, we need only to check a private field inside the behavior class before calling any giving-cup function.

New answer:

Same design, same implementation as our original ideas.

Question 4

Research the Strategy Design Pattern. Consider both the Strategy and Bridge design patterns, would either be useful in implementing computer players with different levels of difficulty/intelligence?

Original answer:

For implementation with varying degree of intelligence, we decided to employ the strategy pattern, so when even a event occurs there will be a call to one of the sub-classes of strategy (abstract super-class) for decision. And each of these sub-class contains its own algorithm for decision making (buy buy buy, or focus cash balance). Hence it resembles the strategy design pattern.

A bridge design pattern is also possible, but would be much more tedious to implement. The bridge pattern essentially separates the interface from the implementation. So in this case, we could treat each event as a interface and write the respective implementation (or possibly how different levels of intelligence would react). This way, though workable, the program become much more coupled.

New answer:

largely similar to our original idea, the decision making process get relayed to the Strategy Object through Behavior. And from there we could implement AI of different difficulties (employing different tactics). So that a smarter computer will call smartComputer for decision whereas an ordinary computer get its decisions from mediocreComputer. Notably, the human players' decisions also comes from Strategy which takes in the command from standard input.

Question 5

Is the Decorator Pattern a good pattern to use when implementing Improvements? Why or why not?

Original answer:

Yes, the Decorator Pattern is a proper choice. Because the player could either choose to improve the property or sell off the improvement for cash. Hence the improvement class will need to be able to call both of these actions. In essence, we are decorating the property with available improvements.

New answer:

We implemented a vector to store the different level of rents, and also stores the level of implement as a private field in Property. We implemented this way, because only rent varies as improvement level increases. In future, if different improvement levels trigger different events, we could easily make use of the existing concrete event objects and implement a decorator pattern to Property.

Q & A (Due date 2)

Question 1

What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

Answer 1

Communication is key. Since the work is distributed between two people, hence it is very important that the team members are not simply working on their own and piece their work together in the end. Team requires working together and not just working in proximity.

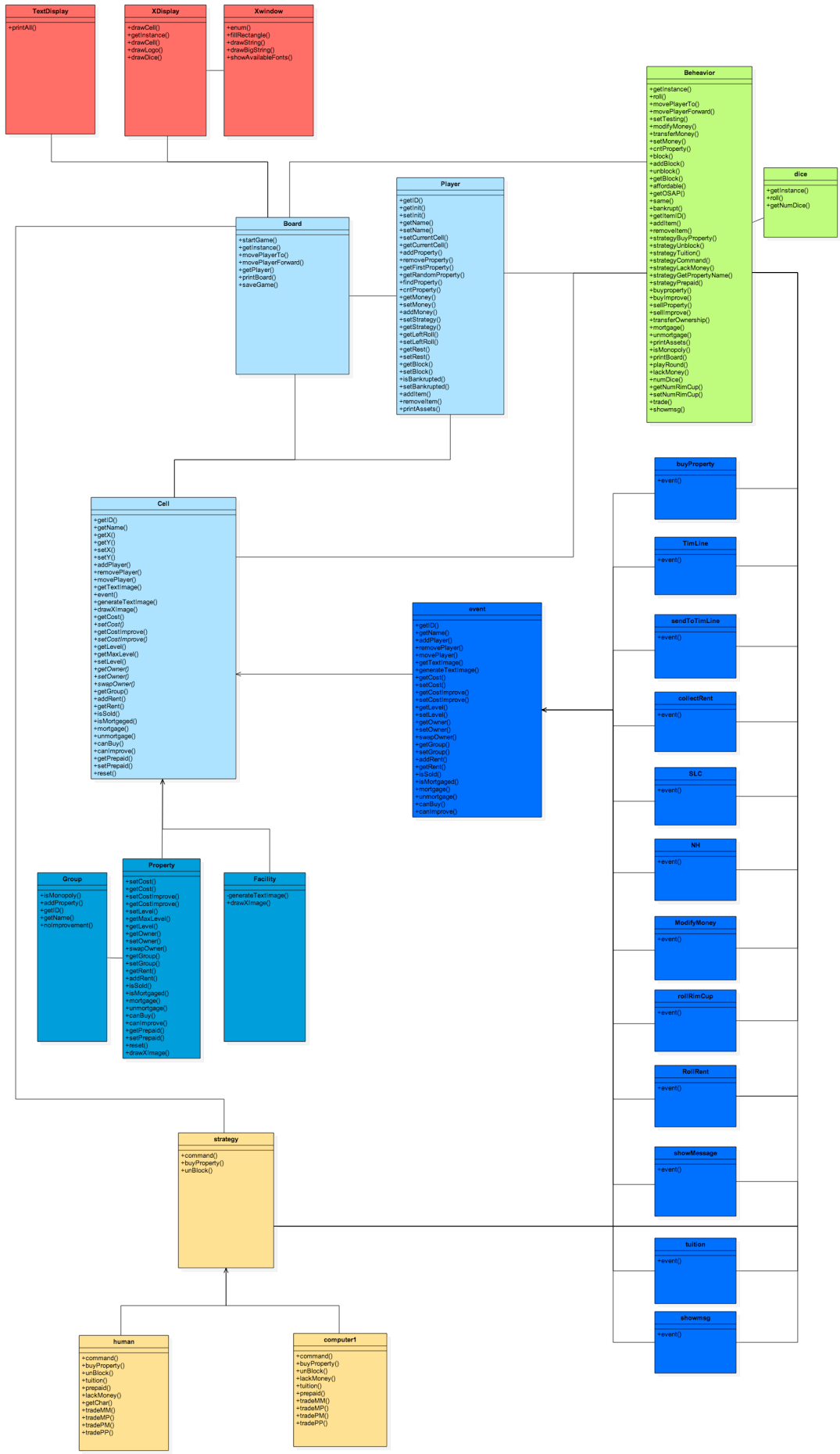
Also for the coding part, we learned that the design matters a lot (so think about the project before starting). A clever design will definitely saves plenty of repetitive work and reduces complexity.

Question 2

What would you have done differently if you had the chance to start over?

Answer 2

If we could start again, we will spend more time designing the details. As it turns out, it gets increasingly difficult as the amount of details that we need to remember surges.


TextDisplay
+printAll()
XDisplay
+drawCell()
+getInstance()
+drawCell()
+drawLogo()
+drawDice()
Xwindow
+enum()
+fillRectangle()
+drawString()
+drawBigString()
+showAvailableFonts()
Beheavior
+getInstance()
+roll()
+movePlayerTo()
+movePlayerForward()
+setTesting()
+modifyMoney()
+transferMoney()
+setMoney()
+cntProperty()
+block()
+addBlock()
+unblock()
+getBlock()
+affordable()
+getDSAPI()
+same()
+bankrupt()
+getItemID()
+addItem()
+removeItem()
+strategyBuyProperty()
+strategyUnblock()
+strategyTuition()
+strategyCommand()
+strategyLackMoney()
+strategyGetPropertyName()
+strategyPrepaid()
+buyproperty()
+buyImprove()
+sellProperty()
+sellImprove()
+transferOwnership()
+mortgage()
+unmortgage()
+printAssets()
+isMonopoly()
+printBoard()
+playRound()
+lackMoney()
+numDice()
+getNumRimCup()
+setNumRimCup()
+trade()
+showmsg()
dice
+getInstance()
+roll()
+getNumDice()
Player
+getID()
+getInit()
+setInit()
+getName()
+setName()
+setCurrentCell()
+getCurrentCell()
+addProperty()
+removeProperty()
+getFirstProperty()
+getRandomProperty()
+findProperty()
+cntProperty()
+getMoney()
+setMoney()
+addMoney()
+setStrategy()
+getStrategy()
+getLeftRoll()
+setLeftRoll()
+getRest()
+setRest()
+getBlock()
+setBlock()
+isBankrupted()
+setBankrupted()
+addItem()
+removeItem()
+printAssets()
Board
+startGame()
+getInstance()
+movePlayerTo()
+movePlayerForward()
+getPlayer()
+printBoard()
+saveGame()
Cell
+getID()
+getName()
+getX()
+getY()
+setX()
+setY()
+addPlayer()
+removePlayer()
+movePlayer()
+getTextImage()
+event()
+generateTextImage()
+drawXImage()
+getCost()
+setCost()
+getCostImprove()
+setCostImprove()
+getLevel()
+getMaxLevel()
+setLevel()
+getOwner()
+setOwner()
+swapOwner()
+getGroup()
+addRent()
+getRent()
+isSold()
+isMortgaged()
+mortgage()
+unmortgage()
+canBuy()
+canImprove()
+getPrepaid()
+setPrepaid()
+reset()
event
+getID()
+getName()
+addPlayer()
+removePlayer()
+movePlayer()
+getTextImage()
+generateTextImage()
+getCost()
+setCost()
+getCostImprove()
+setCostImprove()
+getLevel()
+setLevel()
+getOwner()
+setOwner()
+swapOwner()
+getGroup()
+setGroup()
+addRent()
+getRent()
+isSold()
+isMortgaged()
+mortgage()
+unmortgage()
+canBuy()
+canImprove()
buyProperty
+event()
TimLine
+event()
sendToTimLine
+event()
collectRent
+event()
SLC
+event()
NH
+event()
ModifyMoney
+event()
rollRimCup
+event()
RollRent
+event()
showMessage
+event()
tuition
+event()
showmsg
+event()
Group
+isMonopoly()
+addProperty()
+getID()
+getName()
+noImprovement()
Property
+setCost()
+getCost()
+setCostImprove()
+getCostImprove()
+setLevel()
+getMaxLevel()
+getLevel()
+getOwner()
+setOwner()
+swapOwner()
+getGroup()
+setGroup()
+getRent()
+addRent()
+isSold()
+isMortgaged()
+mortgage()
+unmortgage()
+canBuy()
+canImprove()
+getPrepaid()
+setPrepaid()
+reset()
+drawXImage()
Facility
-generateTextImage()
+drawXImage()
strategy
+command()
+buyProperty()
+unBlock()
human
+command()
+buyProperty()
+unBlock()
+tuition()
+prepaid()
+lackMoney()
+getChar()
+tradeMM()
+tradeMP()
+tradePM()
+tradePP()
computer1
+command()
+buyProperty()
+unBlock()
+lackMoney()
+tuition()
+prepaid()
+tradeMM()
+tradeMP()
+tradePM()
+tradePP()