HIGH LEVEL IMPLEMENTATION OF BB7K

Travis Dunn

Amanda Chan

C S 246 – Object Oriented Design

Professor Nomair Naeem

April 3rd 2015

Table Of Contents

Introduction

This Design Document outlines a high level overview of our project, BB7K, wherein we implemented the much loved childhood game, Monopoly.

Main

High Level Implementations of Crucial Main Functions

Program is called from command line with optional arguments (-load file or -test).
I. -load: file the game is played from the state specified in file.
II.-test: the values of each die roll can be specified by the player.

Read number of players, player names, and chars(board pieces) from cin.

roll - theBoard->movePlayer is called. Method first checks if the player is in the Timsline. Method changes the players location pointer and calls the action method on the location pointer with it's self as an argument. Board is printed after player ends their turn by typing 'next' into the command line.

trade <name> <give> <receive> - active player offers to trade <give> with <name> in return for <receive>.
theboard->trade(currentPlayer, Offer, Receive) called.

improve <property> buy/sell player->improve() improve method reads in property name and checks if player owns the property. Checks if player can afford improvement if yes calls
property->improve(buy/sell) this method bill/compensates the owner the appropriate amount. Board->print() is then called from main.

mortgage <property> - player->mortgage() called, reads in name of property, checks if it owns property. If player owns property property->mortgage() is called this method makes the property stop charging tuition and compensates the owner.

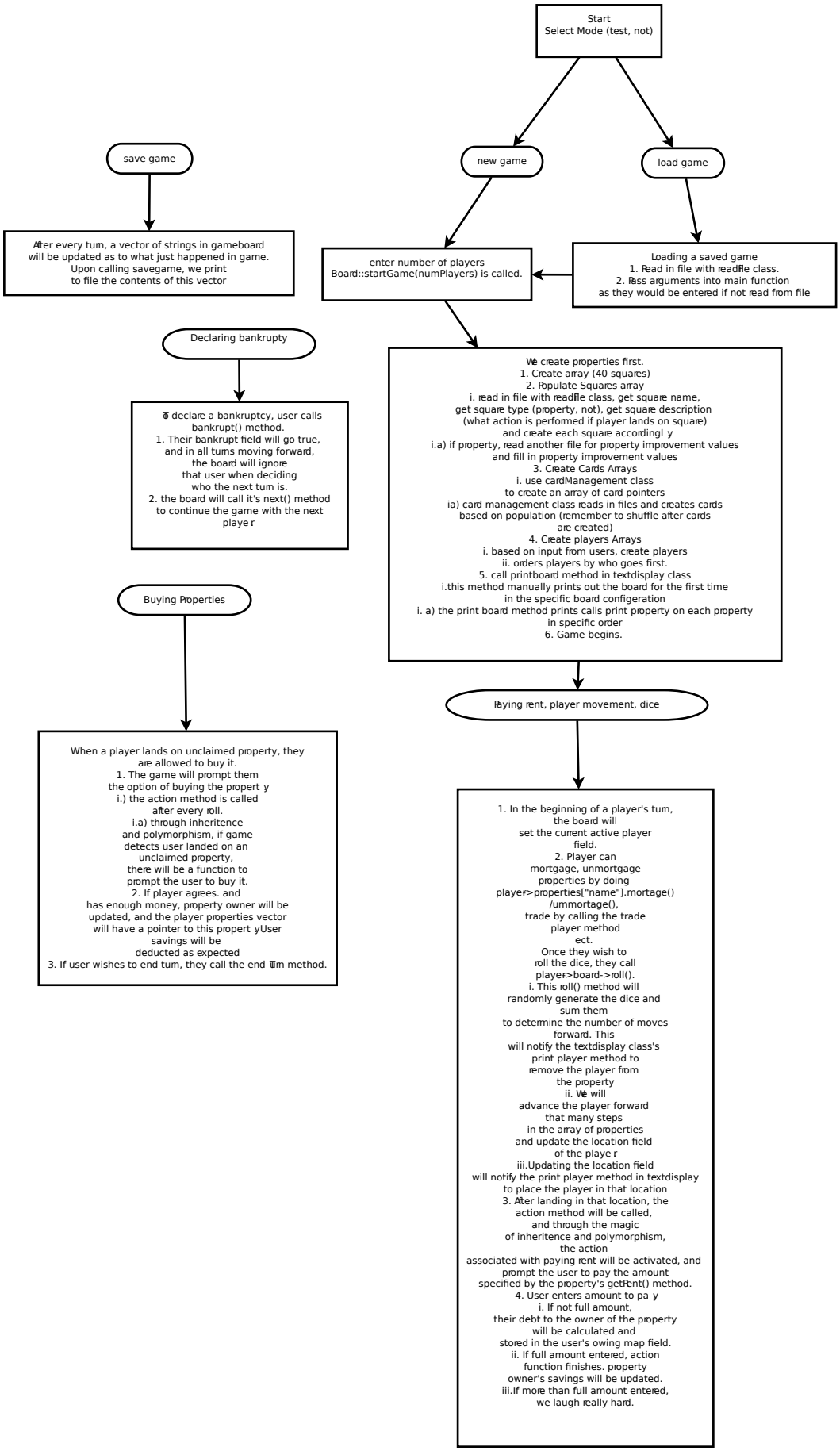
unmortgage <property> - player->unmortgage() called, checks if it owns property bills the owner the lent money plus ten percent. Makes the property start charging tuition again.

bankrupt - player->bankrupt() is called. Checks if player owes more money than has. If yes all assets are given to play who is owed. Players must immediately pay 10% of the principle when inheriting a mortgaged property. They may then choose to unmortgage be paying the principle if not they will have to pay the usual principle plus 10% when unmortgaging later.

assets - calls player->displayAssets(). Does not print if player is deciding how to pay tuition, and also if they are in debt.

save <name of file> - saves the state of the game

next – ends a player's turn, prints the board. Sets all appropriate fields in the Board for the next player.


Start
Select Mode (test, not)
new game
load game
save game
After every turn, a vector of strings in gameboard
will be updated as to what just happened in game.
Upon calling savegame, we print
to file the contents of this vector
enter number of players
Board::startGame(numPlayers) is called.
Loading a saved game
1. Read in file with readFile class.
2. Pass arguments into main function
as they would be entered if not read from file
Declaring bankrupty
We create properties first.
1. Create array (40 squares)
2. Populate Squares array
i. read in file with readFile class, get square name,
get square type (property, not), get square description
(what action is performed if player lands on square)
and create each square accordingl y
i.a) if property, read another file for property improvement values
and fill in property improvement values
3. Create Cards Arrays
i. use cardManagement class
to create an array of card pointers
ia) card management class reads in files and creates cards
based on population (remember to shuffle after cards
are created)
4. Create players Arrays
i. based on input from users, create players
ii. orders players by who goes first.
5. call printboard method in textdisplay class
i.this method manually prints out the board for the first time
in the specific board configeration
i. a) the print board method prints calls print property on each property
in specific order
6. Game begins.
To declare a bankruptcy, user calls
bankrupt() method.
1. Their bankrupt field will go true,
and in all turns moving forward,
the board will ignore
that user when deciding
who the next turn is.
2. the board will call it's next() method
to continue the game with the next
player
Buying Properties
Paying rent, player movement, dice
When a player lands on unclaimed property, they
are allowed to buy it.
1. The game will prompt them
the option of buying the propert y
i.) the action method is called
after every roll.
i.a) through inheritence
and polymorphism, if game
detects user landed on an
unclaimed property,
there will be a function to
prompt the user to buy it.
2. If player agrees. and
has enough money, property owner will be
updated, and the player properties vector
will have a pointer to this propert yUser
savings will be
deducted as expected
3. If user wishes to end turn, they call the end Turn method.
1. In the beginning of a player's turn,
the board will
set the current active player
field.
2. Player can
mortgage, unmortgage
properties by doing
player->properties["name"].mortage()
/ummortage(),
trade by calling the trade
player method
ect.
Once they wish to
roll the dice, they call
player->board->roll().
i. This roll() method will
randomly generate the dice and
sum them
to determine the number of moves
forward. This
will notify the textdisplay class's
print player method to
remove the player from
the property
ii. We will
advance the player forward
that many steps
in the array of properties
and update the location field
of the player
iii.Updating the location field
will notify the print player method in textdisplay
to place the player in that location
3. After landing in that location, the
action method will be called,
and through the magic
of inheritence and polymorphism,
the action
associated with paying rent will be activated, and
prompt the user to pay the amount
specified by the property's getRent() method.
4. User enters amount to pa y
i. If not full amount,
their debt to the owner of the property
will be calculated and
stored in the user's owing map field.
ii. If full amount entered, action
function finishes. property
owner's savings will be updated.
iii.If more than full amount entered,
we laugh really hard.

Basic Class Descriptions

Board: Houses interactions between squares and players. It controls trading, auctioning, creating the board, populating the squares, ect. It contains most functions all other classes call when they need to access fields of other classes they cannot see (a player trying to trade another player for a property, or moving a player to a square) and accesses those private fields to complete the desired action. Whenever a player lands on a square, the Board handles ensuring the square acts on the player accordingly. Whenever a player needs to make a property transfer, the Board handles swapping the properties from one player to another ect.

Player: Contains all information about players. Contains where the player is, information about the player, all their assets ect. Gives the player methods like buying properties, trading, bidding in an auction ect. Handles a player's individual banking information, like transfering and receiving money ect.

DeckBuilder: Builds the SLC and NEEDLES HALL decks by taking in their configeration strings and building the decks.

Card: Represents any SLC or NEEDLES HALL card. Everytime a NEEDLES HALL card or SLC card is used, the square's action method will look at the data of these cards and perform that action on the player.

Square: An abstract square class that has a virtual action method which allows the board to call the action on a subclass of the square.

NonProperty: An abstract class inheriting from square.

Property: An abstract class inheriting from square which is a superclass of all ownable properties.

(all nonproperty squares): These are implemented seperately as advised by a TA. They have different additional fields for their specific non-property square action specifications.

Academic: Inherits from Property, has additional fields to handle improving properties, and buying, mortgaging, unmortgaging these properties.

Residence: Inherits from property, has additional fields to calculate the rent differently.

Gym: Inherits from property, also has additional fields to calculate the rent differently

TCUP: A class which handles giving and returning Tim's Cups in the board. More details in “Questions Revisited” section of document.

NpData: A class structuring nonproperty data for easy access by other files. Contains an extern map containing NpData objects.

SquareData: A class structuring property square data for easy access by other parts of code. Contains an extern map containing property objects

playerData: A class structuring player data for easy access by other parts of the code. Contains an extern map containing playerData objects.

TextDisplay: Class handling printing to the screen. This class mimics assignment 4, question 2's TextDisplay class in implementation, acting as an observer of other parts of the code, like, squares and players.

Board

Description of Critical Methods

Board::Board(String mode) - main
- Before every game a new board must be made.
- The board constructor, Board::Board(String mode), is entirely responsible for the new board.
- Board::Board(String mode) constructs the squares using data available in externed maps.

Board::startGame() - cin
- Reads in player names and avatars from cin to construct Player objects and puti them in the players. Array. Order of play is the order players were entered

Board::load() - saveFile.txt
- Creates ReadFile object with saveFill string and uses it to create and new players array.
- Resulting players have avatar, money, position, rollup the rim cups and turns left in the DC tims line as specified in saveFile.
- Order of players in array is same as oder they where read in.
- Game continues with first read in player going first.

Board::roll() - cin
- calls Board::movePlayer(numSpaces) where numSpaces is the sum of two random numbers between 1 and 6.
- runs currentActivePlayer->location->action(). Action() alters the currentActivePlayer appropriately.

Board::movePlayer(int numSpaces)
- moves the currentActivePlayer a numSpaces squares forward.

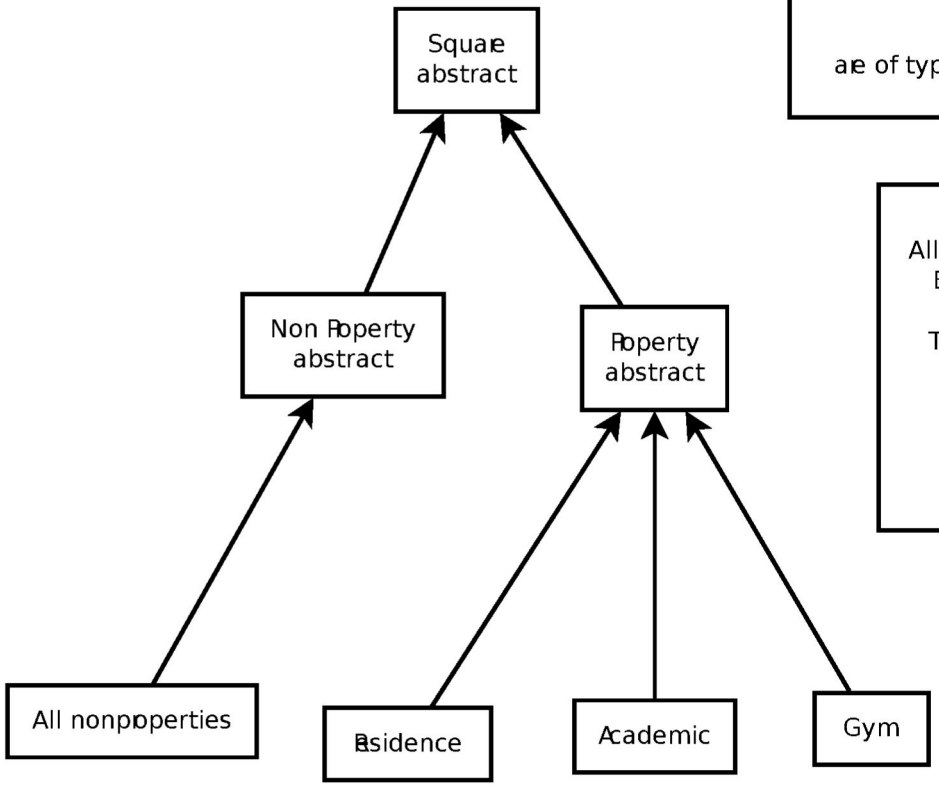
Board::trade(Player *otherPlayer, String offer, String recieve) - cin
- reads accept/reject from cin.
- if accept takes offer from currentActivePlayer and to otherPlayer the takes recieve from otherPlayer and gives it to currentActivePlayer.
- if reject do nothing.

Board::Bankrupt()
- All assets are given to the creditor if the creditor is a player otherwise assets are auctioned.
- Players inheriting mortgaged property are asked if they want to pay 10% of the principle immediately or unmortgage it.
- Assigns the next player to be the activePlayer.

Board::giveMoney(int amount, Player Pointer p)
- Increases savings of p by amount, and calls p->payDebt()

Board::giveDebt(int amount, Player Pointer c, Player Pointer p)
-Increases debt of p by amount, and sets p's creditor to c, then calls p->payDebt()

Square Inheritence Flowchart



There are no squares that can be created that
are of type "Square" or types "Property" and "Non Property", so they are abstract

Reasoning for this type of inheritence:
Allows the board to have an array of just "square" pointers.
Every square in the array has some action associated
to it when a player lands on it.
This type of inheritence, through polymorphism allows
us to simply call a square's action method,
(i.e. theBoard[i]->action()) and the subclasses
of square will call the correct
action method on the player, based on what type
of square it is.

Player

Description of Critical Methods

Player::payDebt()
- defines local int payment as min(debt, savings). Calls Board->giveMoney(payment, creditor) and decrease savings and debt by payment.
- if all debt is paid creditor is assigned to null

Card Inheritence

NEEDLES HALL and SLC Cards Inherit from abstract class card. This was done because both NEEDLES HALL and SLC Cards were cards, and they apply an action on the player. These actions are defined by the configeration strings given in deckBuilder class. Abstracting away a deckBuilder class to build the deck allows us to create decks of any size with any Cards that inact any actions on a player given a configeration string. This was a design decision made for code modularization and scalability purposes.

Questions Revisited

Would the observer pattern be good pattern to use when implementing gameboard?
Due Date 1:
Yes, the following are examples where the observer pattern could be used.
We will have a textDisplay Class which will handle printing to the screen. This Textdisplay class will print out each property and all it's printable attributes. When a property changes state, it will notify this text display class to print out the change in state. This is an example of using the observer pattern, where the text display cells are being notified to print out correct values when the state of a cell has been changed.

We will have a Board class which handles interactions with the board. This board class will have a linked list of properties.
We will also implement a Property Class. Our properties will have an array that is the length of the number of players. The values in our array will be 1 or 0, which will be our 2 states. Either the player represented by the index is on the property (1) or not (0).

We can make properties be observers of adjacent properties to efficiently control player movement. When a player, named player x, needs to move n spaces, we will keep track of how many spaces they have already moved. The property player x is on after the roll will change state to not contain the player, and notify it's adjacent property to change state. This adjacent property now changed state to contain the player. If the player has not moved n spaces, it will change state again to remove the player from itself and notify the next property to contain the player. This will happen n times until player x has moved n spaces. This is another example of each property using the observer pattern to notify it's adjacent properties to change state.

Due Date 2:
The implementation we gave specified the TextDisplay class as an observer of squares and players. Squares notified the TextDisplay to print them out with the correct number of improvements, and players notified the text display to print out correctly where they are.

Instead of using a linked list of properties, we implemented an array of properties. We did this because we know the total number of properties (40) and it was cleaner to access properties by index than it was to access via going through a linked list. Player movement was controlled by moving to the correct index in the properties array.

It was unecessary to include the following array specified on due date 1:
"Our properties will have an array that is the length of the number of players. The values in our array will be 1 or 0, which will be our 2 states. Either the player represented by the index is on the property (1) or not (0)"
because our players had pointers to their location. The player itself was being observed by the TextDisplay and so we could pass in the player's location's coordinates to notify the text display to remove a player and add a player.

Question: What could you do to ensure there are never more than 4 Roll Up the Rim cups?
Due Date 1:
Keep a field called "numberRollUpRimCups" that keeps track of how many cups there are out. Do not give roll up rim cups if that number is 0.

Alternatively, you can make a singleton class called TimsCupDispenser that returns 1 or when you call it to give you a Tims cup. Any time a Tims cup is used you would return it to the dispenser.

Due Date 2:
We created a class called "TCUP" which is the Tim's Cup Dispenser. This class keeps track of how many Tim's Cup's were on the board at a time. It has a private integer field named "numCups", and void public functions giveCup(Player *) and returnCup(Player *).

giveCup(Player *): Gives a cup to a player if there is less than 4 cups out on the board(i.e. increment player's numCup's field and decrements it's own numCups field) and stop giving out cups when there are equal to 4 cups on the board. (does not increment player's numCup's field and does not decrement it's own numCup's field.
returnCup(Player *): Returns a cup to the board,(i.e. increments numCups field.)

As a design decision we did not make class "TCUP" a singleton for the following reasons:
- We would have 4 instances of the singleton "TCUP", which clutters code significantly.
- There are additional functions that are required like, cleanup(), that would also need to be implemented, which reduces code readability.
- For scalability purposes we also decided the singleton design pattern was not ideal. If we needed more Tim's cups out on the board, we would have to create more instances

of Tim's cups, which is much less clean than simply incrementing and decrementing a Tim's cup int field.

However, on the flip side:

- This is at the cost of losing scalability in number of features we could add to the Tim's cups, however. Since if Tim's cups had other fields our implementation would not gracefully handle the change.

At the current stage of our implementation, however, it is not difficult to convert our implementation into a singleton design pattern, hence if given more functionality for Tim's cups, we would have used the singleton design pattern.

Question: Suppose 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?

Due Date 1:

We could implement a doubly - linked list(or array) with pointers to community chest or chance cards. We will generate the cards when the game starts based on the probability table given. We will have a pointer to the top card. If a player lands on a square that requires a community chest, or chance card, they will take the top card, take the action, and that card will notify the card next to it to become the top card. This would be an example of the observer pattern, wherein the top card changes state to not be the top card, and the card underneath it changes state to be the top card.

In the array implementation, all subsequent cards will move forward one index in the array, and the old top card becomes the top card.

In the doubly-linked list implementation, the top card observes the bottom card, and the card underneath it. Once the top card has been used, it notifies the card below it to become the top card, then it becomes the bottom card. This implementation mimics the observer pattern design pattern.

Alternatively, you can define a card class who is a friend of the player class. Each specific type of card would be a child class of the card class and have its own action member that subjects the player to some action unique to the card type. You can then make a deck of these cards by making an array of card pointers and deallocating it with a random index. The probability distribution can be manipulated by filling the array with the desired number of pointers to the same card.

Due Date 2:

We have implemented a deckBuilder class which manages building the cards. The deckBuilder class builds an array of cards. The abstract Card is a class wherein SLC and NEEDLES HALL cards inherit from. The card class has a pointer to the board class. The board class is friends with all the squares and players, hence can access all their private properties. This allows the cards to freely move the player and make any transactions to a player. The cards are built like a set of real cards, meaning, we determine how many cards there would be based on the probabilities given on the table, and construct an array of that size. Using this we can mimic

Monopoly's Community Chest and Chance cards. These cards have two functionalities; moving the player, and/or taking/giving them money. We can use the pointer to the board to allow the board to implement these functionalities. We randomly access the array every time a card is drawn.

A doubly-linked list with an observer pattern was not an ideal implementation, as it was an over-complication of the problem at hand. Implementing such a design had no real benefits.

Question: Can the decorator pattern be used to generalize improvements.
Due Date 1:
Yes improvement classes which inherit from the property class and decorate properties can be made. Each improvement would have a pointer to the next property, a name, a pointer to a function that takes in and returns an int, and a getTuition method that calls the getTuition method on the next property then applies function to it and returns the result. The function in question would be specific to each property object.

Due Date 2:
Yes we still believe we can use the decorator pattern to generalize improvements.
If there is more than one type of improvement that can be made, for example, installing windows or new floors, we could have an abstract decorator class and concrete decorator classes - separately “Floors” improvement, or “Windows” improvement – which each have their own getRent() function, which applies some transformation to the rent of the property it decorates as done in assignment 4.

Changes Between Plan Of Attack and Implementation

We did not implement a FileRead class which reads in information from a text file as specified in our old UML. We were advised by a CS246 TA against implementing such a class; putting data into text files was poor design. Every time the executable needed to be run, the text files containing the data must be in the same folder in order to run the program. Good design entails executables must stand alone. As such, we have made seperate classes inside our program that organizes data, and global maps that contains data. This implementation was decided for scalability purposes.

- Adding new data like new player types entails adding a new entry in the playerData map, which is easily implemented.
- This implementation allows any part of the code access general data.

The Board does not have a pointer to the deckBuilder or a pointer to the different decks as specified in our previous document. It makes more sense for the deck to only be relevant to the NEEDLES HALL or SLC squares. Hence, it makes more sense to have the NEEDLES HALL and SLC Squares and Card classes handle that implementation.

Friend Relationships

Class Player

Board: The board has access to all private fields of players for the following reasons:

- it is responsible for mutating player objects to securely accomplish trading properties and money between players.
- The board is responsible for changing the player's location pointer when the player is moved.
- The board is responsible for adding new players, many fields (tims cups, savings, ect) need to be set during this process.

TCUP: is a friend of the player's numCup's field. It is responsible for incrementing and decrementing a player's number of cup's field. We did not make the Tim's Cup's a singleton class for reasons discussed in the "Questions Revisited" section of this document.

Class Square

Board: The board has access to all private fields of all the squares.

- The board is responsible for creating all it's squares (it's has an owns a relationship with square). In the creation of squares, many fields need to be set, such as it's name, type, ect.
- In game, manipulation of a square's fields may be necessary by the board, such as the property's owner.

Player: The Player class is responsible for editing an academic property's number of improvement, if they own the property. Hence it is friends with an academic property's "numImprovements" field.

Design Patterns Used

Observer:
TextDisplay is an observer of both Square and Player. Anytime a square changes state (improvements are added), it notifies the TextDisplay class to update it on the board. Anytime a player moves, it notifies the TextDisplay class to update where the player is on the board.

Singleton:
The board is a singleton, since there is ever only one board in play at one time.

Considered Implementing:
Visitor Design Pattern on action method on Square:
We could have acheived activating different actions on the player based on what type of square the player was on by using the visitor design pattern, however we did not, as there is an increase in the amount of code written with the visitor design pattern for not much gain. Our Square inheritence hierchy allows us to effectively call different action methods on the player based on which square they are on. There is really only one state a player (the current active player) can be in when they land on a square, so there is not much gain from implementing the visitor design pattern. However, if the player had many different attributes that would effect landing on different squares differently, the visitor design pattern would be ideal.

Cons of this decision:

- We really overloaded our Board Class, as it handles all the private transactions between squares and players.

Singleton Design Pattern on Tim's Cups:
Reasoning against this is in "Questions Revisited" section of this document.

Ownership, Associations

Board: Has owns-a relationships with Squares, Players, TextDisplay. When the board is destroyed, it destroys all it's squares, players, and textdisplay.

Needles Hall, SLC Squares: owns-a deck of cards. When SLC and NEEDLES HALL properties are destroyed, it destroys their respective deck of cards used to determine what action to inact on the player.

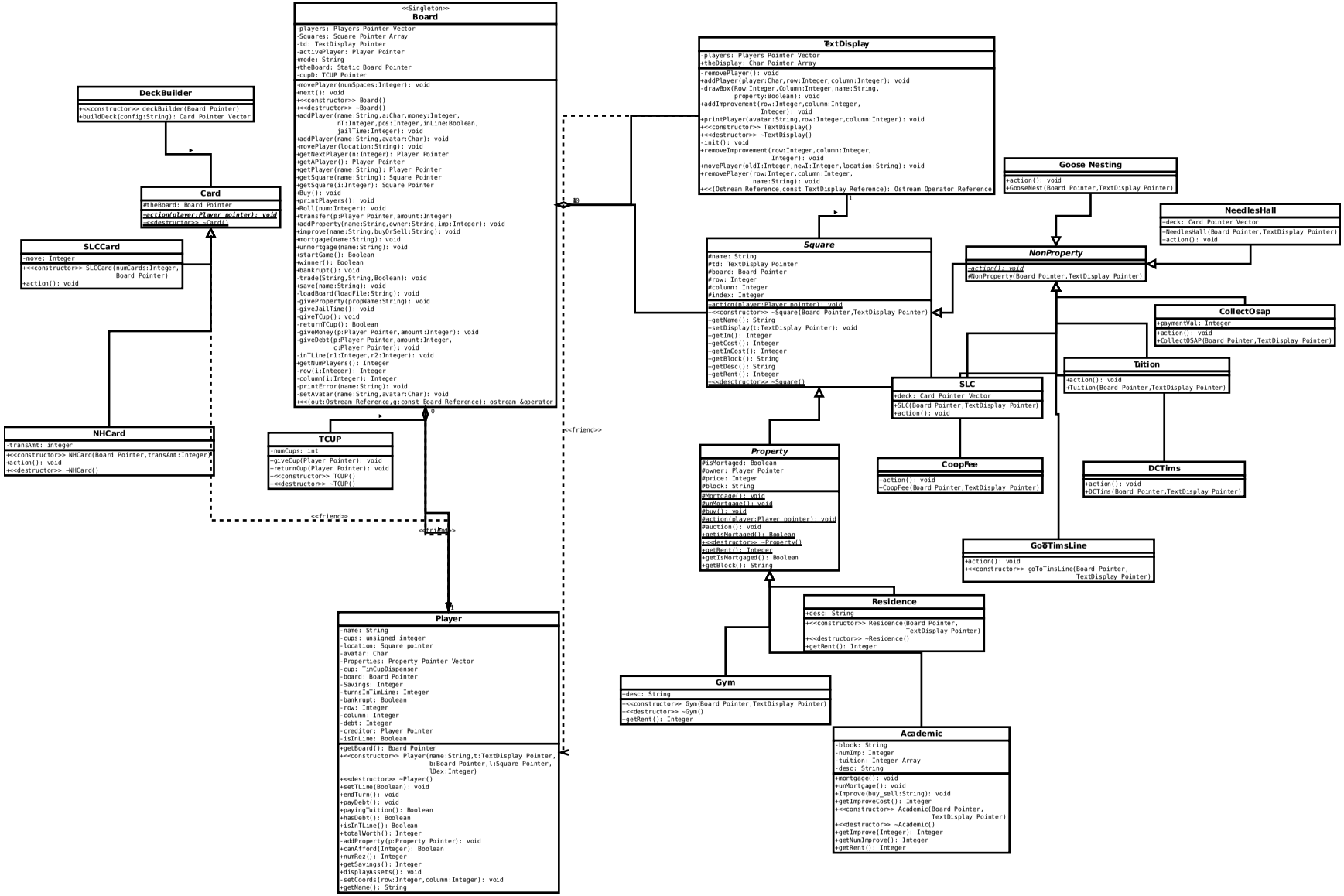
Players: has-a relationship with properties. A player has a vector of pointers to properties. However if the player is removed from the game due to bankrupcty, their properties should not disappear from the board. Hence, a has-a relationship.

Property, NonProperty: is-a relationship with Square, as they inherit from square.

Academic, Residence, Gym: is-a relationship with Property, as they inherit from property.

All non-property squares: is-a relationship with NonProperty, as they inherit from NonProperty.

NEEDLES HALL, SLC Cards: is-a relationship with Card, as they inherit from Card.


DeckBuilder
+<<constructor>> deckBuilder(Board Pointer)
+buildDeck(config:String): Card Pointer Vector
Card
#theBoard: Board Pointer
#action(player:Player pointer): void
+<<destructor>> ~Card()
SLCCard
-move: Integer
+<<constructor>> SLCCard(numCards:Integer, Board Pointer)
+action(): void
NHCard
-transAmt: integer
+<<constructor>> NHCard(Board Pointer,transAmt:Integer)
+action(): void
+<<destructor>> ~NHCard()
<<Singleton>>
Board
-players: Players Pointer Vector
-Squares: Square Pointer Array
-td: TextDisplay Pointer
-activePlayer: Player Pointer
+mode: String
+theBoard: Static Board Pointer
-cupD: TCUP Pointer
-movePlayer(numSpaces:Integer): void
+next(): void
+<<constructor>> Board()
+<<destructor>> ~Board()
+addPlayer(name:String,a:Char,money:Integer, nT:Integer,pos:Integer,inLine:Boolean, jailTime:Integer): void
+addPlayer(name:String,avatar:Char): void
-movePlayer(location:String): void
+getNextPlayer(n:Integer): Player Pointer
+getAPlayer(): Player Pointer
+getPlayer(name:String): Player Pointer
+getSquare(name:String): Square Pointer
+getSquare(i:Integer): Square Pointer
+Buy(): void
+printPlayers(): void
+Roll(num:Integer): void
+transfer(p:Player Pointer,amount:Integer)
+addProperty(name:String,owner:String,imp:Integer): void
+improve(name:String,buyOrSell:String): void
+mortgage(name:String): void
+unmortgage(name:String): void
+startGame(): Boolean
+winner(): Boolean
+bankrupt(): void
-trade(String,String,Boolean): void
+save(name:String): void
-loadBoard(loadFile:String): void
-giveProperty(propName:String): void
-giveJailTime(): void
-giveTCup(): void
-returnTCup(): Boolean
-giveMoney(p:Player Pointer,amount:Integer): void
-giveDebt(p:Player Pointer,amount:Integer, c:Player Pointer): void
-inTLine(r1:Integer,r2:Integer): void
+getNumPlayers(): Integer
-row(i:Integer): Integer
-column(i:Integer): Integer
-printError(name:String): void
-setAvatar(name:String,avatar:Char): void
+<<(out:Ostream Reference,g:const Board Reference): ostream &operator
TCUP
-numCups: int
+giveCup(Player Pointer): void
+returnCup(Player Pointer): void
+<<constructor>> TCUP()
+<<destructor>> ~TCUP()
<<friend>>
<<friend>>
<<friend>>
Player
-name: String
-cups: unsigned integer
-location: Square pointer
-avatar: Char
-Properties: Property Pointer Vector
-cup: TimCupDispenser
-board: Board Pointer
-Savings: Integer
-turnsInTimLine: Integer
-bankrupt: Boolean
-row: Integer
-column: Integer
-debt: Integer
-creditor: Player Pointer
-isInLine: Boolean
+getBoard(): Board Pointer
+<<constructor>> Player(name:String,t:TextDisplay Pointer, b:Board Pointer,l:Square Pointer, Idex:Integer)
+<<destructor>> ~Player()
+setTLine(Boolean): void
+endTurn(): void
+payDebt(): void
+payingTuition(): Boolean
+hasDebt(): Boolean
+isInTLine(): Boolean
+totalWorth(): Integer
-addProperty(p:Property Pointer): void
+canAfford(Integer): Boolean
+numRez(): Integer
+getSavings(): Integer
+displayAssets(): void
-setCoords(row:Integer,column:Integer): void
+getName(): String
TextDisplay
-players: Players Pointer Vector
+theDisplay: Char Pointer Array
-removePlayer(): void
+addPlayer(player:Char,row:Integer,column:Integer): void
-drawBox(Row:Integer,Column:Integer,name:String, property:Boolean): void
+addImprovement(row:Integer,column:Integer, Integer): void
+printPlayer(avatar:String,row:Integer,column:Integer): void
+<<constructor>> TextDisplay()
+<<destructor>> ~TextDisplay()
-init(): void
+removeImprovement(row:Integer,column:Integer, Integer): void
+movePlayer(oldI:Integer,newI:Integer,location:String): void
+removePlayer(row:Integer,column:Integer, name:String): void
+<<(Ostream Reference,const TextDisplay Reference): Ostream Operator Reference
Square
#name: String
#td: TextDisplay Pointer
#board: Board Pointer
#row: Integer
#column: Integer
#index: Integer
#action(player:Player pointer): void
+<<constructor>> ~Square(Board Pointer,TextDisplay Pointer)
+getName(): String
+setDisplay(t:TextDisplay Pointer): void
+getIm(): Integer
+getCost(): Integer
+getImCost(): Integer
+getBlock(): String
+getDesc(): String
+getRent(): Integer
+<<destructor>> ~Square()
Goose Nesting
+action(): void
+GooseNest(Board Pointer,TextDisplay Pointer)
NonProperty
#action(): void
#NonProperty(Board Pointer,TextDisplay Pointer)
NeedlesHall
+deck: Card Pointer Vector
+NeedlesHall(Board Pointer,TextDisplay Pointer)
+action(): void
CollectOsap
+paymentVal: Integer
+action(): void
+CollectOSAP(Board Pointer,TextDisplay Pointer)
Tuition
+action(): void
+Tuition(Board Pointer,TextDisplay Pointer)
SLC
+deck: Card Pointer Vector
+SLC(Board Pointer,TextDisplay Pointer)
+action(): void
CoopFee
+action(): void
+CoopFee(Board Pointer,TextDisplay Pointer)
DCTims
+action(): void
+DCTims(Board Pointer,TextDisplay Pointer)
GoToTimsLine
+action(): void
+<<constructor>> goToTimsLine(Board Pointer, TextDisplay Pointer)
Property
#isMortaged: Boolean
#owner: Player Pointer
#price: Integer
#block: String
#Mortgage(): void
#unMortgage(): void
#buy(): void
#action(player:Player pointer): void
#auction(): void
+getIsMortaged(): Boolean
+<<destructor>> ~Property()
+getRent(): Integer
+getIsMortgaged(): Boolean
+getBlock(): String
Residence
+desc: String
+<<constructor>> Residence(Board Pointer, TextDisplay Pointer)
+<<destructor>> ~Residence()
+getRent(): Integer
Gym
+desc: String
+<<constructor>> Gym(Board Pointer,TextDisplay Pointer)
+<<destructor>> ~Gym()
+getRent(): Integer
Academic
-block: String
-numImp: Integer
-tuition: Integer Array
-desc: String
+mortgage(): void
+unMortgage(): void
+Improve(buy_sell:String): void
+getImproveCost(): Integer
+<<constructor>> Academic(Board Pointer, TextDisplay Pointer)
+<<destructor>> ~Academic()
+getImprove(Integer): Integer
+getNumImprove(): Integer
+getRent(): Integer

Conclusion

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?

We learned it's important both partners have a common goal in mind. In a project that is so time crunched, it is important to delegate tasks according to which partner does what the best. We also learned designing the project early and then implementing it reduced many implementation bugs.
It's easier if most of the work is done while the partners are in the same room. It's a good idea to take breaks, otherwise productivity plummets.

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

Travis claims he would not have taken the course. We should have thought more clearly about the transaction system, as many bugs arose from lack of thought in that area. We began coding in the beginning of the second week, we should have started coding earlier.