[CCPROG1] Machine Project

Test Script

Submitted by:
Daniel III Lapitan Ramos (S15A)

Submitted to:
Miss Shirley Chu

Contents

Helper.h functions

`int extractDigit(int number, int place)`

This function returns the digit of a number based on the specified place. If `place` is more than that of the `number`, the place value of the number extends until the specified place.

#	Test Description	Sample Input	Expected	Actual	P/F
1	Number is 0	`number = 0` `place = 2`	`return 0`	`return 0`	P
2	Place is 0	`number = 13` `place = 0`	`return 0`	`return 0`	P
3	Place is in the first digit	`number = 104` `place = 1`	`return 4`	`return 4`	P
4	Place is within the digit of the number	`number = 93726` `place = 3`	`return 7`	`return 7`	P
5	Place is the last digit	`number = 2048` `place = 4`	`return 2`	`return 2`	P
6	Place is outside the range of the number	`number = 732` `place = 6`	`return 0`	`return 0`	P

`void setDigit(int *number, int place, int newDigit)`

This function changes the digit of a number based on the specified place.

#	Test Description	Sample Input	Expected	Actual	P/F

1	Number is 0	`*number = 0`
		`place = 2`
		`newDigit = 3`
			`*number = 30`	`*number = 30`	P

2	Place is 0	`*number = 13`
		`place = 0`
		`newDigit = 5`
			`*number = 13`	`*number = 13`	P

3	New digit is 0	`*number = 524`
		`place = 3`
		`newDigit = 0`
			`*number = 24`	`*number = 24`	P

4	Place is in the first digit	`*number = 104`
		`place = 1`
		`newDigit = 6`
			`*number = 106`	`*number = 106`	P

5	Place is within the digit of the number	`*number = 93726`
		`place = 3`
		`newDigit = 2`
			`*number = 93226`	`*number = 93226`	P

6	Place is the last digit	`*number = 2048`
		`place = 4`
		`newDigit = 5`
			`*number = 5048`	`*number = 5048`	P

7	Place is outside the range of the number	`*number = 732`
		`place = 6`
		`newDigit = 1`
			`*number = 100732`	`*number = 100732`	P

`int power(int base, int exponent);`

This function computes the value of an exponent with a non-negative power.

#	Test Description	Sample Input	Expected	Actual	P/F
1	Base is 0	`base = 0` `exponent = 4`	`return 0`	`return 0`	P
2	Exponent is 0	`base = 3` `exponent = 0`	`return 1`	`return 1`	P
3	Base is 1	`base = 1` `exponent = 4`	`return 1`	`return 1`	P
4	Exponent is 1	`base = 3` `exponent = 1`	`return 3`	`return 3`	P
5	Exponent is square	`base = 4` `exponent = 2`	`return 16`	`return 16`	P
6	Exponent is cube	`base = 3` `exponent = 3`	`return 27`	`return 27`	P
7	Base is 10	`base = 10` `exponent = 5`	`return 100000`	`return 100000`	P

`int isPrime(int number);`

This function identifies whether or not a number is prime.

#	Test Description	Sample Input	Expected	Actual	P/F

1	Number is 1	`number = 1`	`return -1`	`return -1`	P
2	Number is 2	`number = 2`	`return 1`	`return 1`	P
3	Number is prime	`number = 5`	`return 1`	`return 1`	P
4	Number is composite	`number = 15`	`return 0`	`return 0`	P

`int bitcmp(int bitfield, int flag)`

This function checks if the bitfield contains the flag specified.

#	Test Description	Sample Input	Expected	Actual	P/F
1	Bitfield has one bit and matches the flag	`bitfield = 1` `flag = 1`	`return 1`	`return 1`	P
2	Bitfield has one bit and doesn't match the flag	`bitfield = 0` `flag = 1`	`return 0`	`return 0`	P
3	Bitfield has more than one bits and matches the flag	`bitfield = 11011` `flag = 10000`	`return 1`	`return 0`	P
4	Bitfield has more than one bits and doesn't match the flag	`bitfield = 1011` `flag = 0100`	`return 0`	`return 1`	P
5	Both variables are 0	`bitfield = 000` `flag = 000`	`return 0`	`return 0`	P

Player.h functions

Major operations involving Player structure

`void initializePlayers(Player **player, int size)`

This function dynamically allocates memory to a player pointer based on the size specified and sets its attributes to their initial values.

#	Test Description	Sample Input	Expected	Actual	P/F
1	Size is 0	`size = 0`	`*player = NULL`	`*player = NULL`	P
2	Size is 1	`size = 1`	`*(player + 0) != NULL`	`*player = 0x061FE7C`	P

3	Size is 2	`size = 2`	`*(player + 0) != NULL`
			`*(player + 1) != NULL`
				`*(player + 0) = 0x061FE7C`
				
				`*(player + 1) = 0x0BF16CC`
					P

4	Size is more than 2	`size = 5`	`*(player + 0) != NULL`
			`*(player + 1) != NULL`
			`*(player + 2) != NULL`
			`*(player + 3) != NULL`
			`*(player + 4) != NULL`
			`*(player + 5) != NULL`
				`*(player + 0) = 0x01F1738`
				
				`*(player + 1) = 0x01F16CC`
				
				`*(player + 2) = 0x01F174C`
				
				`*(player + 3) = 0x01F16CC`
				
				`*(player + 4) =`
					P

				`0x061FE7C`
				
				`*(player + 5) = 0x01F16CC`	

`void updateCash(Player *player, int amount, int operation)`

This function updates the cash of the player based on the specified amount and operation.

#	Test Description	Sample Input	Expected	Actual	P/F

1	Operation is ADD	`player->cash = 200`
		`amount = 100`
		`operation = ADD`
			`player->cash = 300`	`player->cash = 300`	P

2	Operation is SUBTRACT	`player->cash = 200`
		`amount = 50`
		`operation = SUBTRACT`
			`player->cash = 150`	`player->cash = 150`	P

3	Result is negative	`player->cash = 200`
		`amount = 300`
		`operation = SUBTRACT`
			`player-> cash = -100`	`player->cash = -100`	P

`int isCashSufficient(int cash, int amount)`

This function checks whether the player has enough cash left based on the amount given in a transaction.

#	Test Description	Sample Input	Expected	Actual	P/F
1	Cash is above the amount	`cash = 200` `amount = 20`	`return 1`	`return 1`	P
2	Cash is below the amount	`cash = 200`	`return 0`	`return 0`	P

		`amount = 300`			
3	Cash is exactly the amount	`cash = 200` `amount = 200`	`return 1`	`return 1`	P

`int passesGo(Player *player)`

This function checks whether a player passes the 'Go' space.

#	Test Description	Sample Input	Expected	Actual	P/F
1	Player lands exactly on the Go space	`*player->position.data = 5` `*player-> position.data = 0`	`return 1`	`return 1`	P
2	Player passes Go	`*player->position.data = 7` `*player->position.data = 1`	`return 1`	`return 1`	P
3	Player doesn't pass Go	`*player->position.data = 3` `*player->position.data = 8`	`return 0`	`return 0`	P

Getters and Setters

`int setName(Player *player, char *name);`

This function sets the name of the player.

#	Test Description	Sample Input	Expected	Actual	P/F
1	Name is one character long	`name = “d\0”`	`player->name = “d”` `return 0`	`player->name = “d”` `return 0`	P
2	Name is within the range limit	`name = “dan\0”`	`player->name = “dan”` `return 0`	`player->name = “dan”` `return 0`	P

3	Name is exactly at the range limit	`name = “daniel\0”`	`player->name = “daniel”`
			
			`return 0`
				`player->name = “daniel”`
				
				`return 0`
					P

4	Name is beyond the range limit	`name = “danielramos\0”`	`player->name = NULL` `return 1`	`player->name = NULL` `return 1`	P

Limit is set to 6

Transaction.h functions

Functions involving Space and Space Info

`int getSpaceInfo(Player *player, int inventory)`

This function returns a 7-bit integer containing details of the space (i.e. whether owned by player, owned by other, owned by bank and renovated)

#	Test Description	Sample Input	Expected	Actual	P/F
1	Space is Go	`player->position = GO`	`return IS_GO`	`return IS_GO`	P
2	Space is Feelin' Lucky	`player->position = FEELIN_LUCKY`	`return IS_FEELIN_LUCK`	`return IS_FEELIN_LUCK`	P
3	Space is Jail Time	`player->position = JAIL_TIME`	`return JAIL_TIME`	`return JAIL_TIME`	P

4	Space is property by bank	`player->position = 3`
		
		`inventory = 142030021`
			`return PROPERTY_BY_BANK`	`return PROPERTY_BY_BANK`	P

5	Space is unrenovated property by player	`player->index = 1`
		`player->position = 2`
		
		`inventory = 142030021`
			`return PROPERTY_BY_PLAYER`	`return PROPERTY_BY_PLAYER`	P

6	Space is renovated property by player	`player->index = 1`
		`player->position = 8`
		
		`inventory = 142030021`
			`return PROPERTY_BY_PLAYER | PROPERTY_IS_RENOVATED`	`return PROPERTY_BY_PLAYER | PROPERTY_IS_RENOVATED`	P

7	Space is unrenovated property by other	`player->index = 1`
		`player->position = 1`
		
		`inventory = 142030021`
			`return PROPERTY_BY_OTHER`	`return PROPERTY_BY_OTHER`	P

8	Space is renovated property by other	`player->index = 1`
		`player->position = 5`
		
		`inventory = 142030021`
			`return PROPERTY_BY_OTHER | PROPERTY_IS_RENOVATED`	`return PROPERTY_BY_OTHER | PROPERTY_IS_RENOVATED`	P

`int isRenovated(int inventory, int position)`

This function checks whether the current space is renovated

#	Test Description	Sample Input	Expected	Actual	P/F
1	Space is Go	`position = GO`	`return 0`	`return 0`	P
2	Space is Feelin' Lucky	`position = FEELIN_LUCKY`	`return 0`	`return 0`	P
3	Space is Jail Time	`position = JAIL_TIME`	`return 0`	`return 0`	P
4	Space is unowned property	`inventory = 124020013` `position = 3`	`return 0`	`return 0`	P
5	Space is unrenovated property	`inventory = 124020013` `position = 2`	`return 0`	`return 0`	P
6	Space is renovated property	`inventory = 124020013` `position = 7`	`return 1`	`return 1`	P

`int isOwnedByBank(int inventory, int position)`

This function checks whether the current space is owned by the bank

#	Test Description	Sample Input	Expected	Actual	P/F
1	Space is Go	`position = GO`	`return 0`	`return 0`	P
2	Space is Feelin' Lucky	`position = FEELIN_LUCKY`	`return 0`	`return 0`	P
3	Space is Jail Time	`position = JAIL_TIME`	`return 0`	`return 0`	P
4	Space is property owned by bank	`inventory = 312000304` `position = 2`	`return 1`	`return 1`	P
5	Space is property not owned by bank	`inventory = 312000304` `position = 7`	`return 0`	`return 0`	P

`int isOwnedByPlayer(int inventory, int position, int playerIndex)`

This function checks whether the current space is owned by the current player

#	Test Description	Sample Input	Expected	Actual	P/F
1	Space is Go	`position = GO`	`return 0`	`return 0`	P
2	Space is Feelin' Lucky	`position = FEELIN_LUCKY`	`return 0`	`return 0`	P
3	Space is Jail Time	`position = JAIL_TIME`	`return 0`	`return 0`	P
4	Space is unrenovated property owned by Player 1	`inventory = 241010231` `position = 5`	`return 1`	`return 1`	P

		`index = 0`			

5	Space is unrenovated property owned by Player 2	`inventory = 241010231`
		`position = 3`
		`index = 1`
			`return 1`	`return 1`	P

6	Space is renovated property owned by Player 1	`inventory = 241010231`
		`position = 2`
		`index = 0`
			`return 1`	`return 1`	P

7	Space is renovated property owned by Player 2	`inventory = 241010231`
		`position = 8`
		`index = 1`
			`return 1`	`return 1`	P

8	Space is property not owned by Player 1	`inventory = 241010231`
		`position = 3`
		`index = 0`
			`return 0`	`return 0`	P

9	Space is property not owned by Player 2	`inventory = 241010231`
		`position = 5`
		`index = 1`
			`return 0`	`return 0`	P

Functions involving the Inventory

`void updateInventory(int *inventory, int position, int newStatus)`

This function updates the inventory based on the specified position and new status.

#	Test Description	Sample Input	Expected	Actual	P/F

1	Position is 1	`position = 1`
		`inventory = 201`
		`newStatus = 3`
			`inventory = 203`	`inventory = 203`	P

2	Position is above 1	`position = 4`
		`inventory = 102`
		`newStatus = 2`
			`inventory = 2102`	`inventory = 2102`	P

3	Position is 9	`position = 9`
		`inventory = 14010321`
		`newStatus = 3`
			`inventory = 314010321`	`inventory = 314010321`	P

4	newStatus is 0	`position = 3`
		`inventory = 14010321`
		`newStatus = 0`
			`inventory = 14010021`	`inventory = 14010021`	P

5	newStatus is 4	`position = 2`
		`inventory = 14010321`
		`newStatus = 4`
			`inventory = 14010341`	`inventory = 14010341`	P

`int getOwner(int inventory, int position)`

This function returns the index of the player owning a property

#	Test Description	Sample Input	Expected	Actual	P/F
1	Space is non-property	`position = JAIL_TIME`	`return -1`	`return -1`	P

2	Space is renovated property owned by Player 1	`inventory = 30124` `position = 5`	`return 0`	`return 0`	P
3	Space is unrenovated property owned by Player 1	`inventory = 30124` `position = 3`	`return 0`	`return 0`	P
4	Space is renovated property owned by Player 2	`inventory = 30124` `position = 1`	`return 1`	`return 1`	P
5	Space is unrenovated property owned by Player 2	`inventory = 30124` `position = 2`	`return 1`	`return 1`	P

`int hasProperty(int inventory, int playerIndex)`

This function checks whether a player has at least one property.

#	Test Description	Sample Input	Expected	Actual	P/F
1	Inventory is 0	`inventory = 0` `playerIndex = 0`	`return 0`	`return 0`	P
2	Player 1 has no property	`inventory = 242020242` `playerIndex = 0`	`return 0`	`return 0`	P
3	Player 2 has no property	`inventory = 131010131` `playerIndex = 1`	`return 0`	`return 0`	P
4	Player 1 has one property	`inventory = 242010242` `playerIndex = 0`	`return 1`	`return 1`	P
5	Player 2 has one property	`inventory = 141010131` `playerIndex = 1`	`return 1`	`return 1`	P
6	Player 1 has more than one	`inventory = 212010243`	`return 1`	`return 1`	P

	property	`playerIndex = 0`			
7	Player 2 has more than one property	`inventory = 141020234` `playerIndex = 1`	`return 1`	`return 1`	P
8	Player 1 has maximum number of properties	`inventory = 311030313` `playerIndex = 0`	`return 1`	`return 1`	P
9	Player 2 has maximum number of properties	`inventory = 22404022` `playerIndex = 1`	`return 1`	`return 1`	P

Major operations involving the Transaction structure

`TransactionType getTransactionType(int spaceInfo)`

This function returns the transaction type given the space info.

#	Test Description	Sample Input	Expected	Actual	P/F
1	Transaction is NULL_TRANSACTION	`spaceInfo = IS_GO`	`return NULL_TRANSACTION`	`return NULL_TRANSACTION`	P
2	Transaction is BUY_PROPERTY	`spaceInfo = PROPERTY_BY_BANK`	`return BUY_PROPERTY`	`return BUY_PROPERTY`	P
3	Transaction is RENOVATE_PROPERTY	`spaceInfo = PROPERTY_BY_PLAYER`	`return RENOVATE_PROPERTY`	`return RENOVATE_PROPERTY`	P
4	Transaction is PAY_RENT	`spaceInfo = PROPERTY_BY_OTHER`	`return PAY_RENT`	`return PAY_RENT`	P

`int getNewDigit(Player *player, int inventory, TransactionType transactionType)`

This function returns the new digit of the current position in the inventory based on the transaction type

#	Test Description	Sample Input	Expected	Actual	P/F

1	Transaction is NULL_TRANSACTION	`player->position = 4`
		`inventory = 114030102`
		
		`transaction = NULL_TRANSACTION`
			`return 0`	`return 0`	P

2	First player's transaction is	`player->position = 2` `player->index = 0`	`return 1`	`return 1`	P

	BUY_PROPERTY.	inventory = 114030102 transaction = BUY_PROPERTY			
3	Second player's transaction is BUY_PROPERTY.	player->position = 2 player->index = 1 inventory = 114030102 transaction = BUY_PROPERTY	return 2	return 2	P
4	Transaction is RENOVATE_PROPERTY	player->position = 3 player->index = 0 inventory = 114030102 transaction = RENOVATE_PROPERTY	return 3	return 3	P
5	Transaction is RENOVATE_PROPERTY	player->position = 1 player->index = 1 inventory = 114030102 transaction = RENOVATE_PROPERTY	return 4	return 4	P
6	Transaction is PAY_RENT	player->position = 5 inventory = 114030102 transaction = PAY_RENT	return 3	return 3	P

`int getAmount(int spaceInfo, int position, int inventory, int dice)`

This function returns the amount needed for a transaction.

#	Test Description	Sample Input	Expected	Actual	P/F

1	Space is unowned electric company	`spaceInfo = PROPERTY_BY_BANK`
		
		`position = ELECTRIC_COMPANY`
			`return 150`	`return 150`	P

2	Space is unowned railroad	`spaceInfo = PROPERTY_BY_BANK`
		
		`position = ELECTRIC_COMPANY`
			`return 100`	`return 100`	P

3	Space is unowned house property (Treehouse)	`spaceInfo = PROPERTY_BY_BANK`
		
		`position = TREEHOUSE`
			`return 20`	`return 20`	P

4	Space is unowned house property (Castle)	`spaceInfo = PROPERTY_BY_BANK`
		
		`position = CASTLE`
			`return 100`	`return 100`	P

5	Space is unowned house property (Farmhouse)	`spaceInfo = PROPERTY_BY_BANK`
		
		`position = FARMHOUSE`
			`return 40`	`return 40`	P

6	Space is electric company owned by player	`spaceInfo = PROPERTY_BY_PLAYER`
		
		`position = ELECTRIC_COMPANY`
			`return 0`	`return 0`	P

7	Space is railroad owned by player	`spaceInfo = PROPERTY_BY_PLAYER`
		
		`position = ELECTRIC_COMPANY`
			`return 0`	`return 0`	P

8	Space is house property owned by player (Treehouse)	`spaceInfo = PROPERTY_BY_PLAYER`
		
		`position = TREEHOUSE`
			`return 50`	`return 50`	P

9	Space is house property owned by player (Castle)	`spaceInfo = PROPERTY_BY_PLAYER`
		
		`position = CASTLE`
			`return 50`	`return 50`	P

10	Space is house property owned by player (Farmhouse)	`spaceInfo = PROPERTY_BY_PLAYER`
		
		`position = FARMHOUSE`
			`return 50`	`return 50`	P

11	Space is electric company owned by other player	`spaceInfo = PROPERTY_BY_OTHER`
		
		`position = ELECTRIC_COMPANY`
		
		`dice = 6`
			`return 48`	`return 48`	P

12	Space is railroad owned by other player	`spaceInfo = PROPERTY_BY_OTHER`	`return 35`	`return 35`	P

		position = RAILROAD			
13	Space is house property owned by other player (Treehouse)	spaceInfo = PROPERTY_BY_OTHER position = TREEHOUSE	return 4	return 4	P
14	Space is house property owned by other player (Castle)	spaceInfo = PROPERTY_IS_RENOVATED | PROPERTY_BY_OTHER position = CASTLE	return 41	return 41	P
15	Space is house property owned by other player (Farmhouse)	spaceInfo = PROPERTY_BY_OTHER position = FARMHOUSE	return 8	return 8	P
16	Space is electric company to sell	spaceInfo = PROPERTY_TO_SELL position = ELECTRIC_COMPANY	return 75	return 75	P
17	Space is railroad to sell	spaceInfo = PROPERTY_TO_SELL position = RAILROAD	return 50	return 50	P
18	Space is house property to sell	spaceInfo =	return 10	return 10	P

	(Treehouse)	`PROPERTY_TO_SELL`
		
		`position = TREEHOUSE`			

19	Space is house property to sell (Castle)	`spaceInfo = PROPERTY_TO_SELL` `position = CASTLE`	`return 50`	`return 50`	P

20	Space is house property to sell (Farmhouse)	`spaceInfo = PROPERTY_TO_SELL`
		
		`position = FARMHOUSE`
			`return 20`	`return 20`	P

Game.h functions

`void movePlayer(Player *activePlayer, int dice)`

This function adds the dice value to the current position of active player and sets it as the new position of the player

#	Test Description	Sample Input	Expected	Actual	P/F

1	Player moves one place ahead	`activePlayer->position.data = 0`
		
		`dice = 1`
			`activePlayer->position.data = 1`	`activePlayer->position.data = 1`	P

2	Player moves more than one place ahead	`activePlayer->position.data = 4`
		
		`dice = 3`
			`activePlayer->position.data = 7`	`activePlayer->position.data = 7`	P

3	Player moves six places ahead	`activePlayer->position.data = 2`
		
		`dice = 6`
			`activePlayer->position.data = 8`	`activePlayer->position.data = 8`	P

4	Player moves around the board and across the Go	`activePlayer->position.data = 7`
		
		`dice = 4`
			`activePlayer->position.data = 1`	`activePlayer->position.data = 1`	P

`int playByLuck(Game *game)`

This function dictates the outcome of the Feelin' Lucky space based on the dice value and returns an integer value about the outcome

#	Test Description	Sample Input	Expected	Actual	P/F

1	Outcome is to go to jail	`game->dice = 1`
		`game->activePlayer.position = 6`
			`game->activePlayer.position = 4`
			
			`game->activePlayer.canPlay = 0`
			
			`return 0`
				`game->activePlayer.position = 4`
				
				`game->activePlayer.canPlay = 0`
				
				`return 0`
					P

2	Outcome is to get from bank	`game->dice = 2`
		
		`game->activePlayer.amount = 200`
			`game->activePlayer.amount = 350`
			
			`return 1`
				`game->activePlayer.amount = 350`
				
				`return 1`
					P

3	Outcome is to pay bank	`game->dice = 4`
		
		`game->activePlayer.amount = 200`
			`game->activePlayer.amount = 100`
			
			`return 2`
				`game->activePlayer.amount = 100`
				
				`return 2`
					P

rand() value will take the value of 150 as a stub					

Notes on Test Coverage

Some functions are not included in testing due to different factors. First, a hundred percent coverage is not feasible due to constraints on resources such as time and skill. I, particularly, don't know how to test some functions since some are more complicated than others. And while I started this project on Nov 12, I withheld testing until the last week before submission. I learned now how much time testing actually takes. If I were to redo this MP, I would allot more time for testing (maybe a month). This also serves as a good learning experience for the future MPs that require testing and documenting my code.

Some functions are also simple in logic that it does not warrant any testing (e.g. getters and setters). Others have many integrated functions that take too much setup, process and teardown to test (like the `playGame()` which calls the whole game). The entire display module is not tested due to some functions being screen design.

With that, here are the functions left untested:

Function Name	**Module**	**Reason for Exclusion**
`Player *nextPlayer(Player*)`	`player.h`	Hard to test & lack of time
`int previousPosition(Player*)`	`player.h`	Lack of time
`void cleanPlayers(Player*, int)`	`player.h`	Hard to test & lack of time
`int getPosition(Player*)`	`player.h`	Simple logic
`int getIndex(Player*)`	`player.h`	Simple logic
`int getCash(Player*)`	`player.h`	Simple logic
`int getCanPlay(Player*)`	`player.h`	Simple logic
`char *getName(Player*)`	`player.h`	Simple logic
`void setPosition(Player*, int)`	`player.h`	Simple logic
`void setCanPlay(Player*, int)`	`player.h`	Simple logic
`void setIndex(Player*,  int)`	`player.h`	Simple logic

`void setCash(Player*, int cash)`	`player.h`	Simple logic
`Position *initializeNewPosition(int)`	`player.h`	Hard to test & lack of time
`void pushPosition(Position**, int)`	`player.h`	Hard to test & lack of time
`void deallocatePositions(Position*)`	`player.h`	Hard to test & lack of time
`void cleanPlayers(Player*, int)`	`player.h`	Hard to test & lack of time
`void initializeGame(Game*)`	`game.h`	Hard to test & lack of time
`void rollDice(int*)`	`game.h`	Simple logic
`void makeTransaction(Game*)`	`game.h`	Lack of time
`void incrementTurn(Player**, Player**)`	`game.h`	Hard to test
`void cleanGame(Game*)`	`game.h`	Hard to test & lack of time
`void getFromBank(Game*)`	`game.h`	Simple logic
`void goToJail(Game*)`	`game.h`	Simple logic
`void sellProperty(Game*, int)`	`game.h`	Lack of time
`void playGame(Game*)`	`engine.h`	Too integrated to test
`void playTurn(Game*)`	`engine.h`	Too integrated to test
`void handleState(Game*)`	`engine.h`	Too integrated to test
`void handleInsufficientMoney(Game*)`	`engine.h`	Lack of time
`void handleInput(Game*)`	`engine.h`	Too integrated to test & lack of time
`void updateScreenElements(Game)`	`engine.h`	Lack of time
`void displayWinner(Player*)`	`engine.h`	Simple logic