Milestone 4

Concurrent Poker Player Team

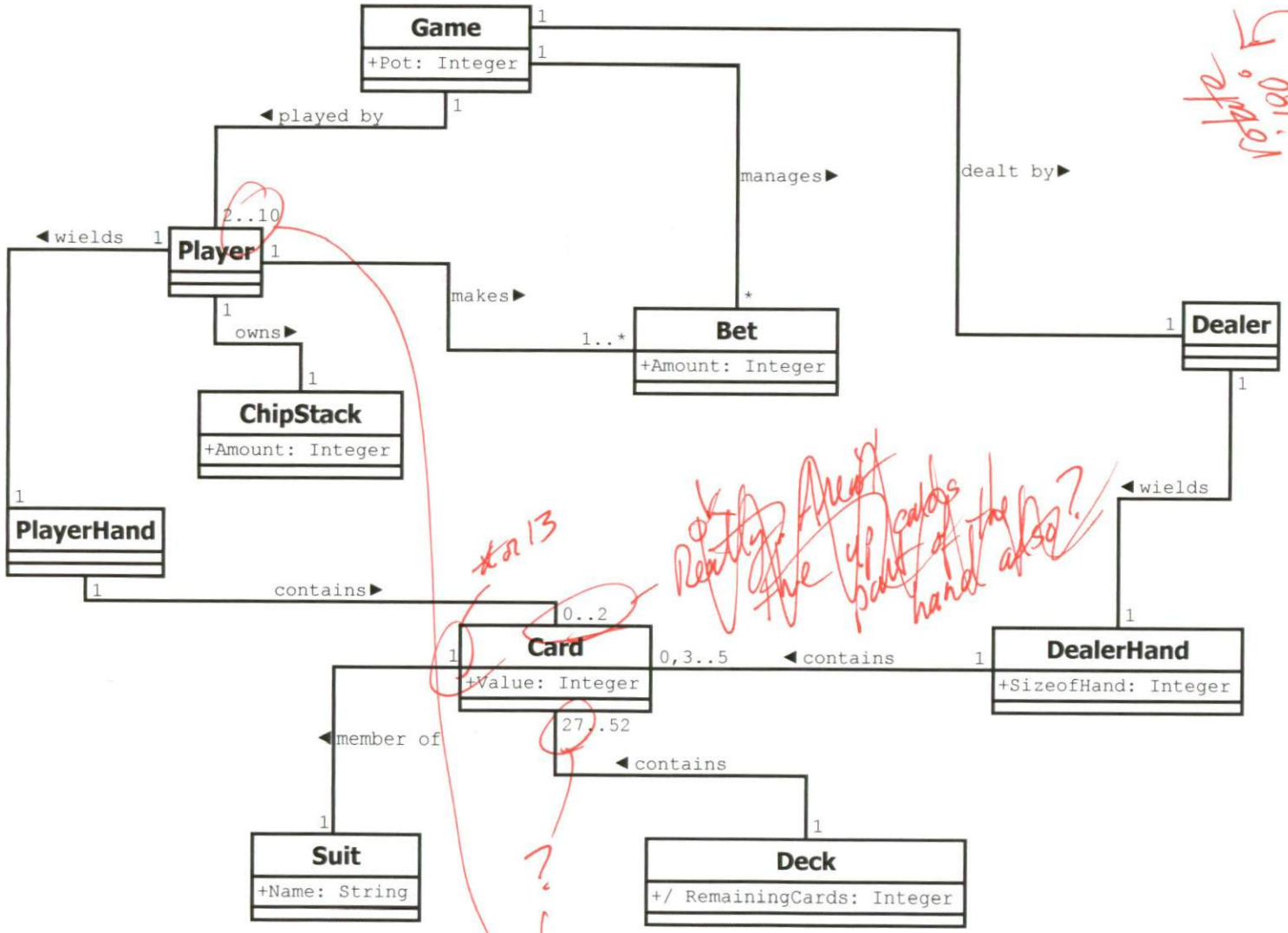
Mark Jenne, Bennie Waters, Ian Roberts, Sarah Jabon

2/1/2010

Contents

add page numbers.

Document would be more professional if diagrams were embedded images in the Word document + headings were consistently written in Word, not scaled with figures.


Game
+Pot: Integer
played by
manages
dealt by
wields
Player
makes
owns
2..10
Bet
+Amount: Integer
Dealer
ChipStack
+Amount: Integer
PlayerHand
contains
0..2
Card
+Value: Integer
0,3..5
contains
DealerHand
+SizeofHand: Integer
wields
member of
27..52
contains
Suit
+Name: String
Deck
+/ RemainingCards: Integer

The Game class represents the entire poker game. The Game class has one attribute, Pot, which describes the amount of money currently in play. The Game manages Bets, because there are rules in the game that specify the amounts that a bet can be based on game states.

A Dealer wields one DealerHand. In Texas Hold 'Em Poker, the DealerHand consists of zero to five Cards depending on the round in the game. Therefore, SizeofHand is an attribute of DealerHand. SizeofHand is always zero, three, four, or five based on Texas Hold 'Em play.

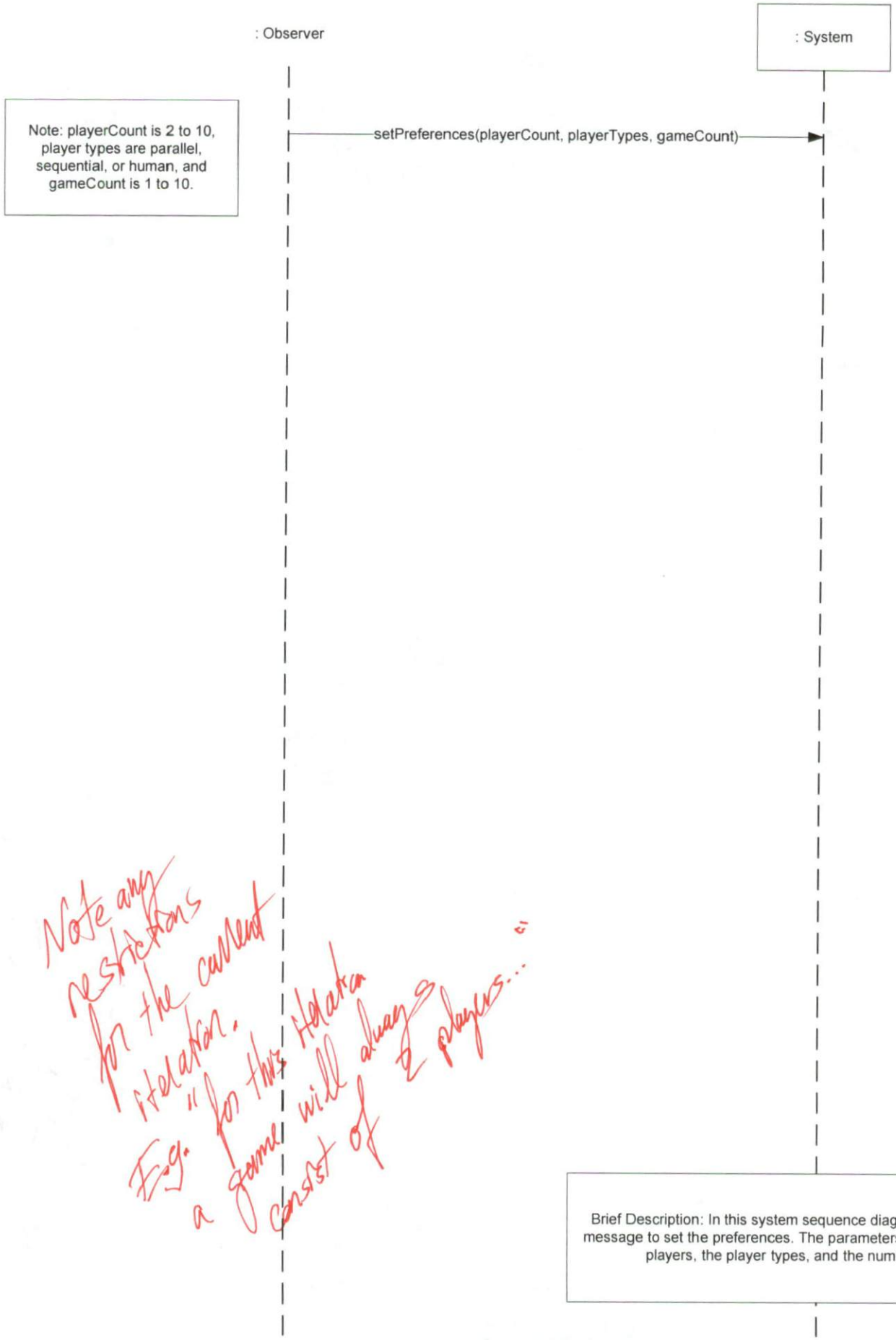
A Deck contains all the Cards that are not currently in play. The number of cards in the Deck is an attribute of Deck called RemainingCards. The Deck will always contain between 27 and 52 Cards, because the largest number of Cards that can be in play is 25 (two for each Player and five for the Dealer).

A Player wields one PlayerHand, which contains zero to two Cards, depending on the round of the poker game. At the beginning of the game, the Player has zero cards. Throughout the rest of the game, the Player will have two cards. A Player also owns one Stack, which has an attribute of Amount. Amount is an integer representing the amount of money with which the Player can use to bet. The Player makes Bets. Since the Player must make a Bet at the beginning of the poker game, the Player must make at least one Bet. The Bets have an attribute, Amount, which represents the amount of money the Player bet.

SSD – Use Case 1
(Start Game)

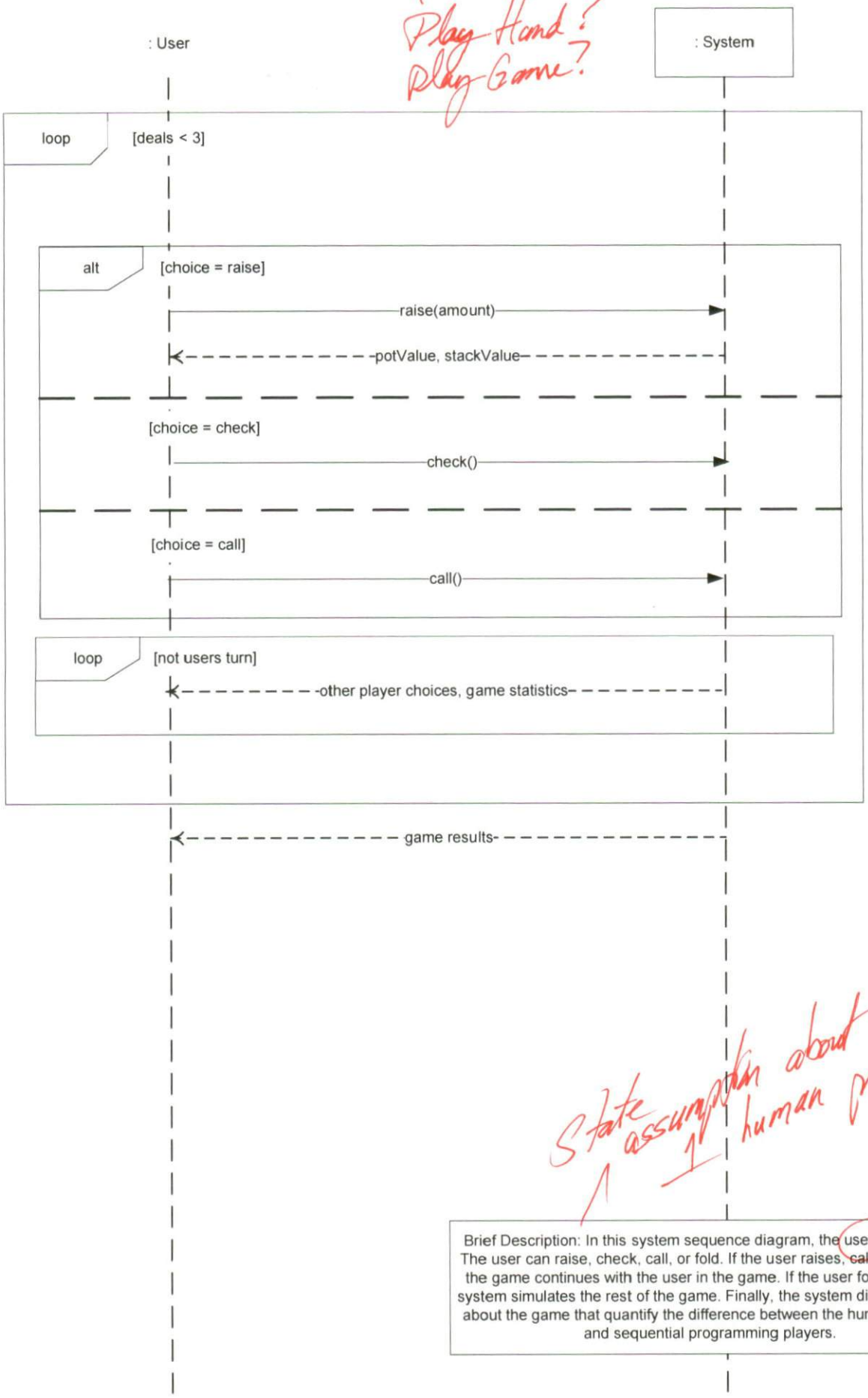
Please use a larger font size for the text descriptions. Other labels are fine.

SSD – Use Case 2
(Set Preferences)



SSD – Use Case 3
(Take Turn)

Play Hand?
Play Game?



Note: The user can choose to fold at any time. In this case, the system finishes the simulation without the user. Then, the system displays the game results.

State assumption about human player

Brief Description: In this system sequence diagram, the user take a turn. The user can raise, check, call, or fold. If the user raises, calls, or checks, the game continues with the user in the game. If the user folds, then the system simulates the rest of the game. Finally, the system displays results about the game that quantify the difference between the human, parallel, and sequential programming players.

S-V

Contract CO1: startGame

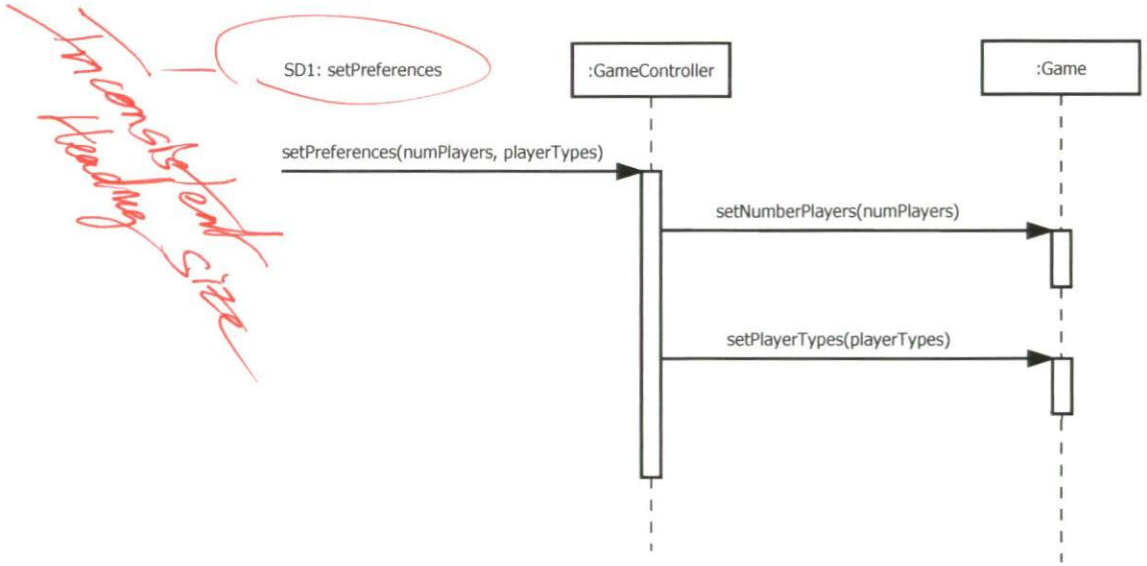
Operation:	startGame()
Cross References:	Use Cases: Start Game
Preconditions:	The system is set to create a game with two players – one sequential and one parallel.
Postconditions:	An instance of Game, *game*, was created An instance of GameStatistics, *gameStats*, was created Attributes of *gameStats* were set based on simulation

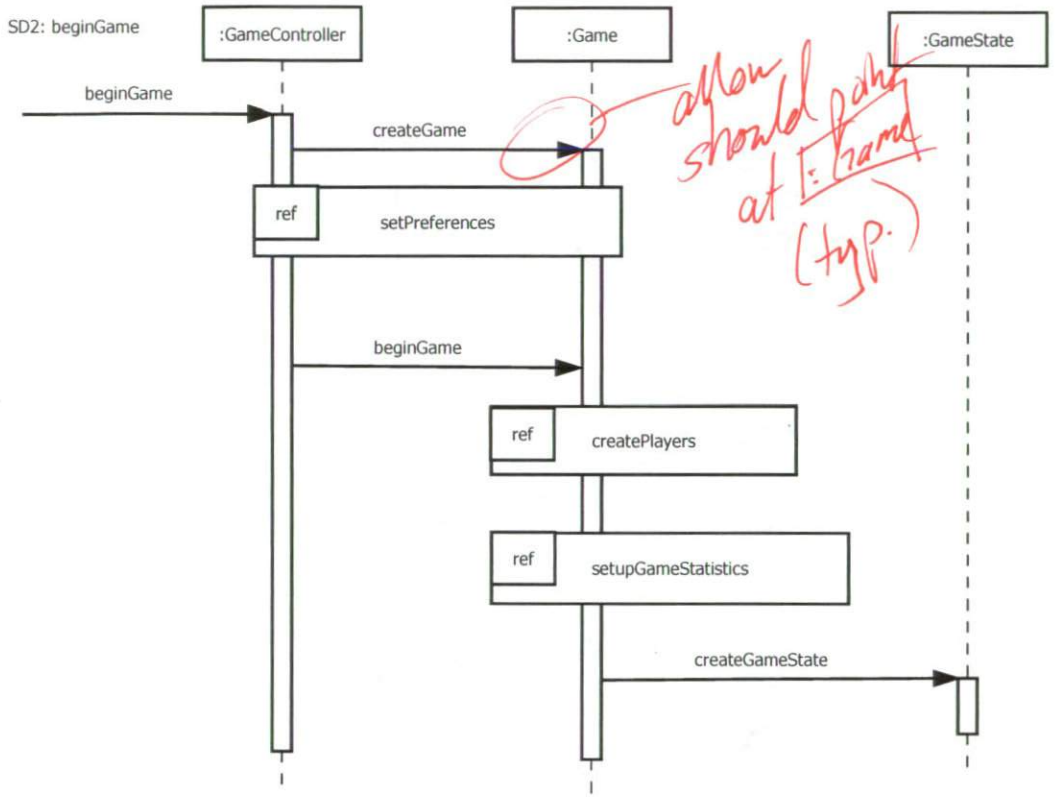
Contract CO2: setPreferences

Operation:	setPreferences(playerCount, playerTypes{List}, gameCount)
Cross References:	Use Cases: Set Preferences
Preconditions:	The application has initialized properly Parameter menus are functioning properly An instance of Game, *game*, exists
Postconditions:	*game.playerCount* was set to playerCount *game.gameCount* was set to gameCount *player*, an instance of HumanPlayer, was created Instances of other Players were created corresponding to each playerType

Contract CO3: raise

Operation:	raise(amount)
Cross References:	Use Cases: Take Turn
Preconditions:	The game is in play It is the user's turn in the game An instance of Game, *game*, exists *player[i]* is currentPlayer *player[i]* is an instance of HumanPlayer *player[i+1]* is nextPlayer
Postconditions:	*game.pot* was increased by amount *player[i].stack* was decreased by amount *player[i]*'s turn ended *player[i+1]*'s turn began


SD1: setPreferences
:GameController
:Game
setPreferences(numPlayers, playerTypes)
setNumberPlayers(numPlayers)
setPlayerTypes(playerTypes)


SD2: beginGame
:GameController
:Game
:GameState
beginGame
createGame
ref
setPreferences
beginGame
ref
createPlayers
ref
setupGameStatistics
createGameState

Contract CO4: check

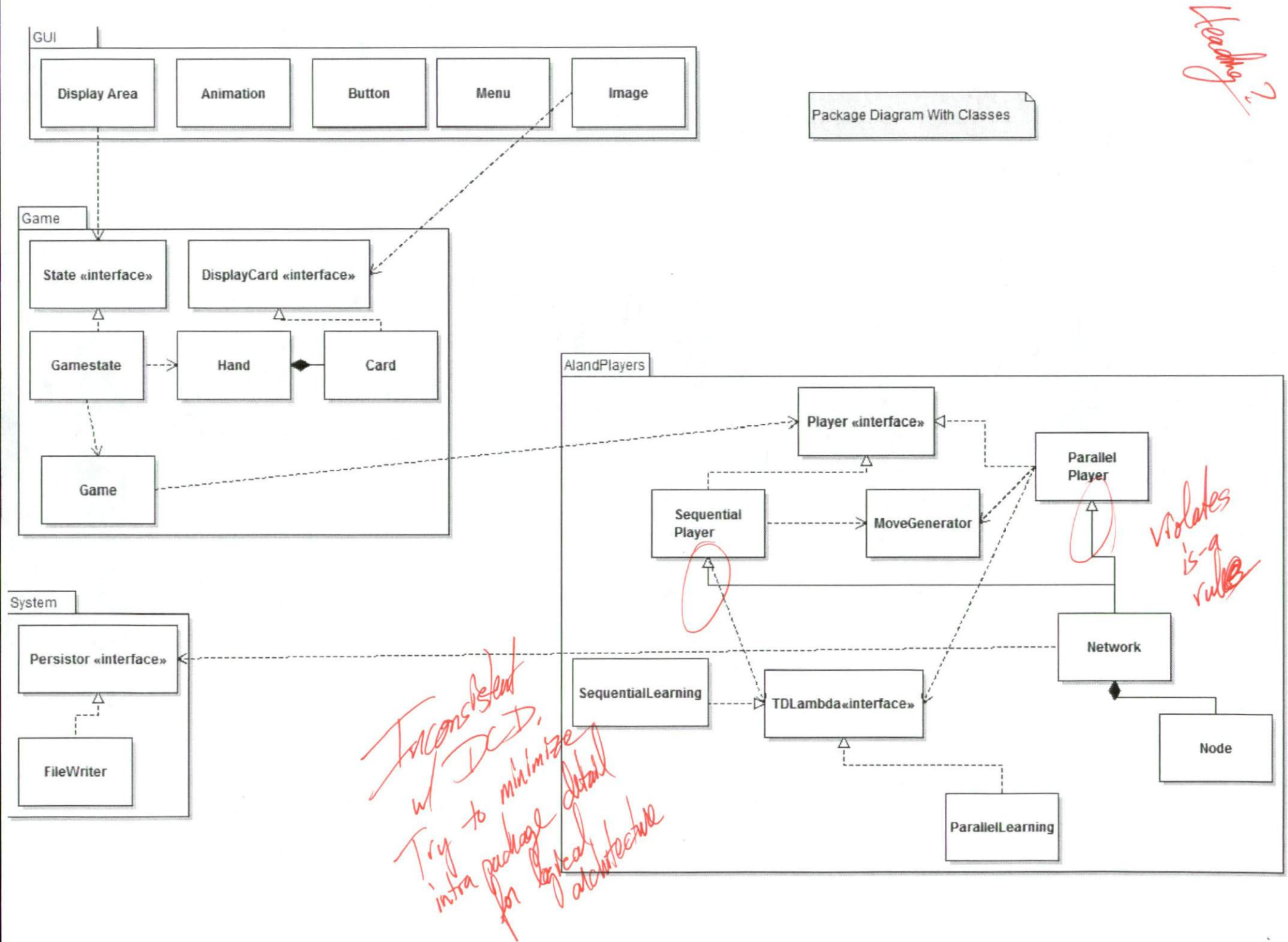
Operation:	check()
Cross References:	Use Cases: Take Turn
Preconditions:	The game is in play
	players{list} are playing game
	player[i] is currentPlayer
	player[i] is an instance of HumanPlayer
	player[i+1] is nextPlayer
Postconditions:	*game.pot* was increased by the amount of the last raise
	player[i].stack was decreased by amount of the last raise
	player[i]'s turn ended
	player[i+1]'s turn began

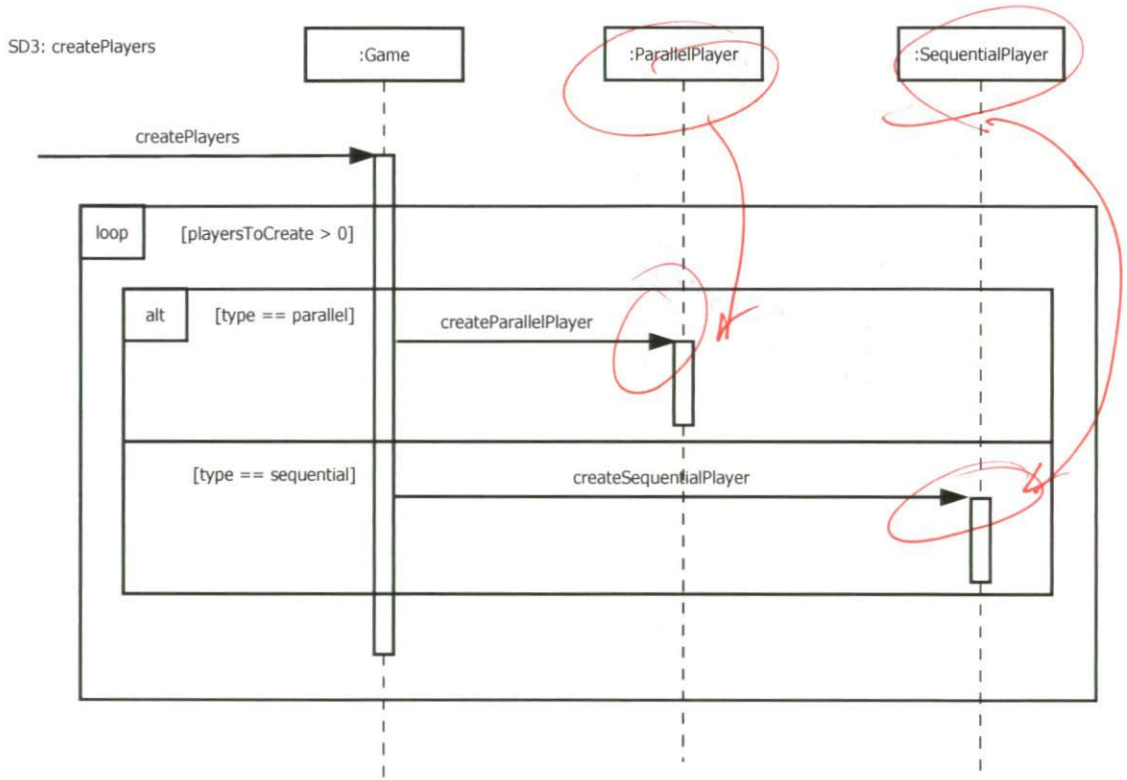
Contract CO5: check

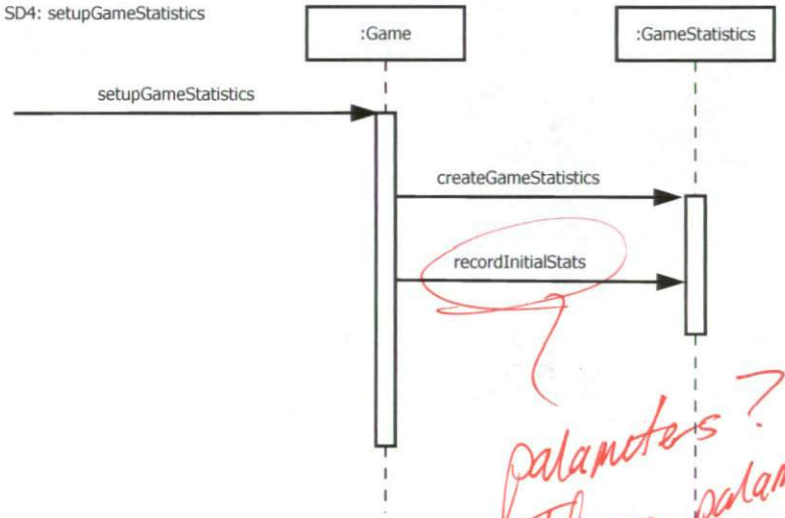
Operation:	call()
Cross References:	Use Cases: Take Turn
Preconditions:	The game is in play
	players{list} are playing game
	player[i] is currentPlayer
	player[i] is an instance of HumanPlayer
	player[i+1] is nextPlayer
Postconditions:	*player[i]*'s turn ended
	player[i+1]'s turn began

Contract CO6: fold

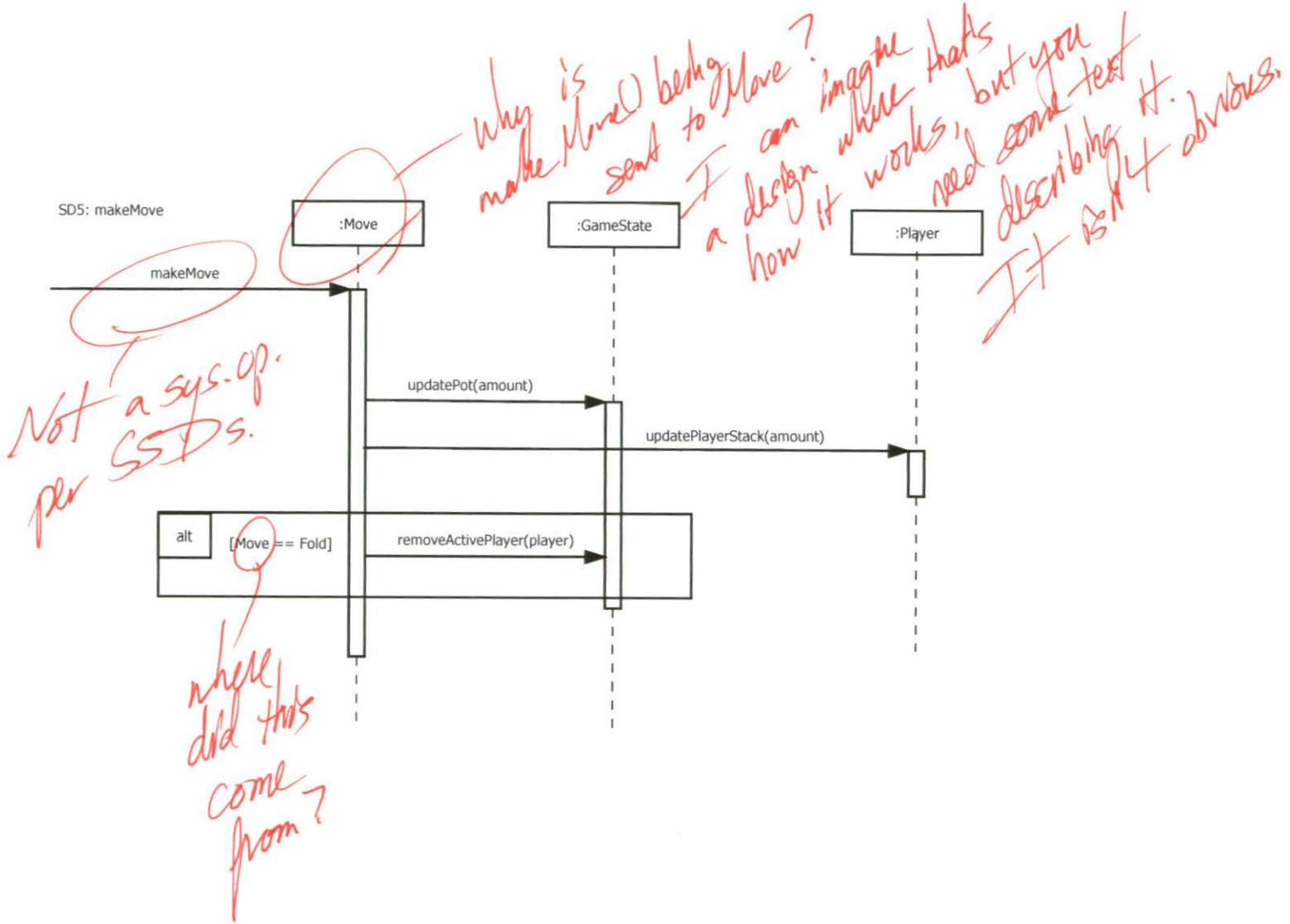
Operation:	fold()
Cross References:	Use Cases: Take Turn
Preconditions:	The game is in play
	players{list} are playing game
	player[i] is currentPlayer
	player[i] is an instance of HumanPlayer
	player[i+1] is nextPlayer
Postconditions:	*player[i]*'s turn ended
	player[i+1]'s turn began
	player[i] was removed from *players{list}*

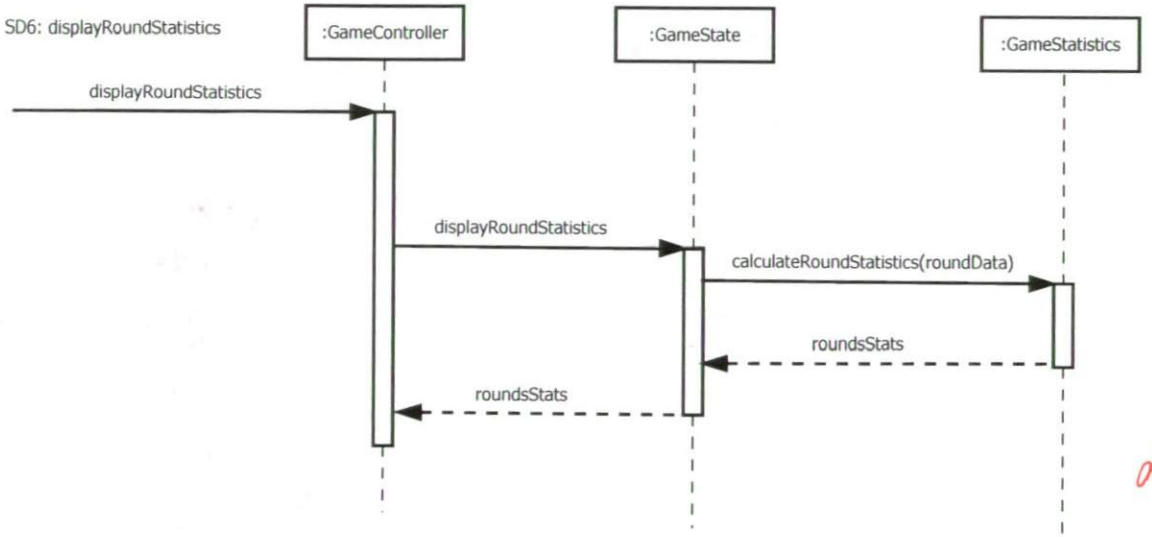

Heading 2
Package Diagram With Classes
GUI
Display Area
Animation
Button
Menu
Image
Game
State «interface»
DisplayCard «interface»
Gamestate
Hand
Card
Game
AIandPlayers
Player «interface»
Parallel Player
Sequential Player
MoveGenerator
violates is-a rule
System
Persistor «interface»
Network
SequentialLearning
TDLambda«interface»
Node
FileWriter
ParallelLearning
Increase cohesion w/ DCD. Try to minimize intra package detail for logical architecture

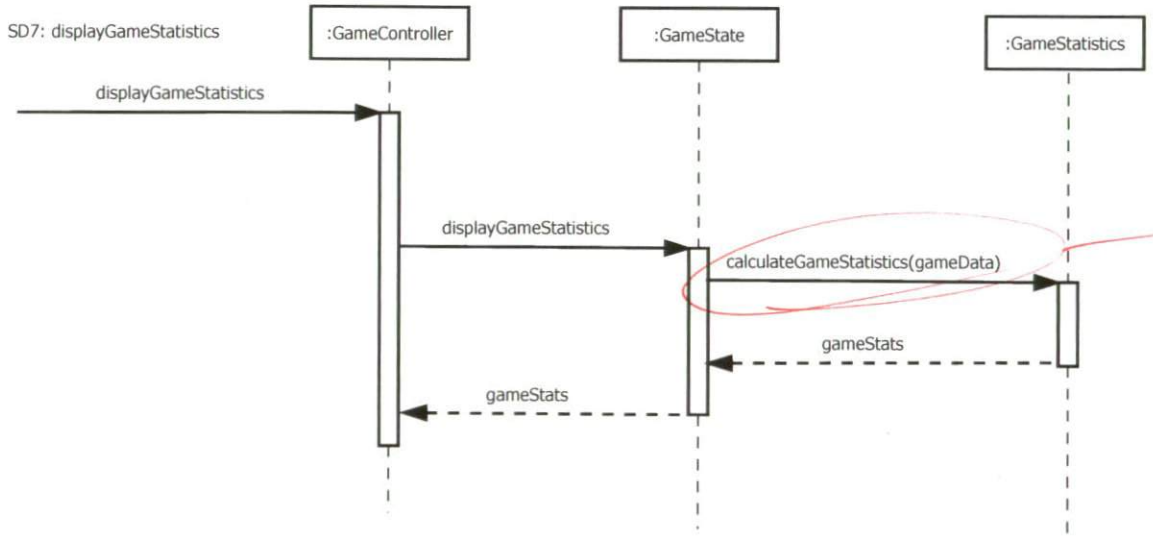

SD3: createPlayers
:Game
:ParallelPlayer
:SequentialPlayer
createPlayers
loop
[playersToCreate > 0]
alt
[type == parallel]
createParallelPlayer
[type == sequential]
createSequentialPlayer



parameters?
If no parameters, the G.S. would just be initialized in its constructor.
(Even w/ parameters, why not just pass them in the constructor?)

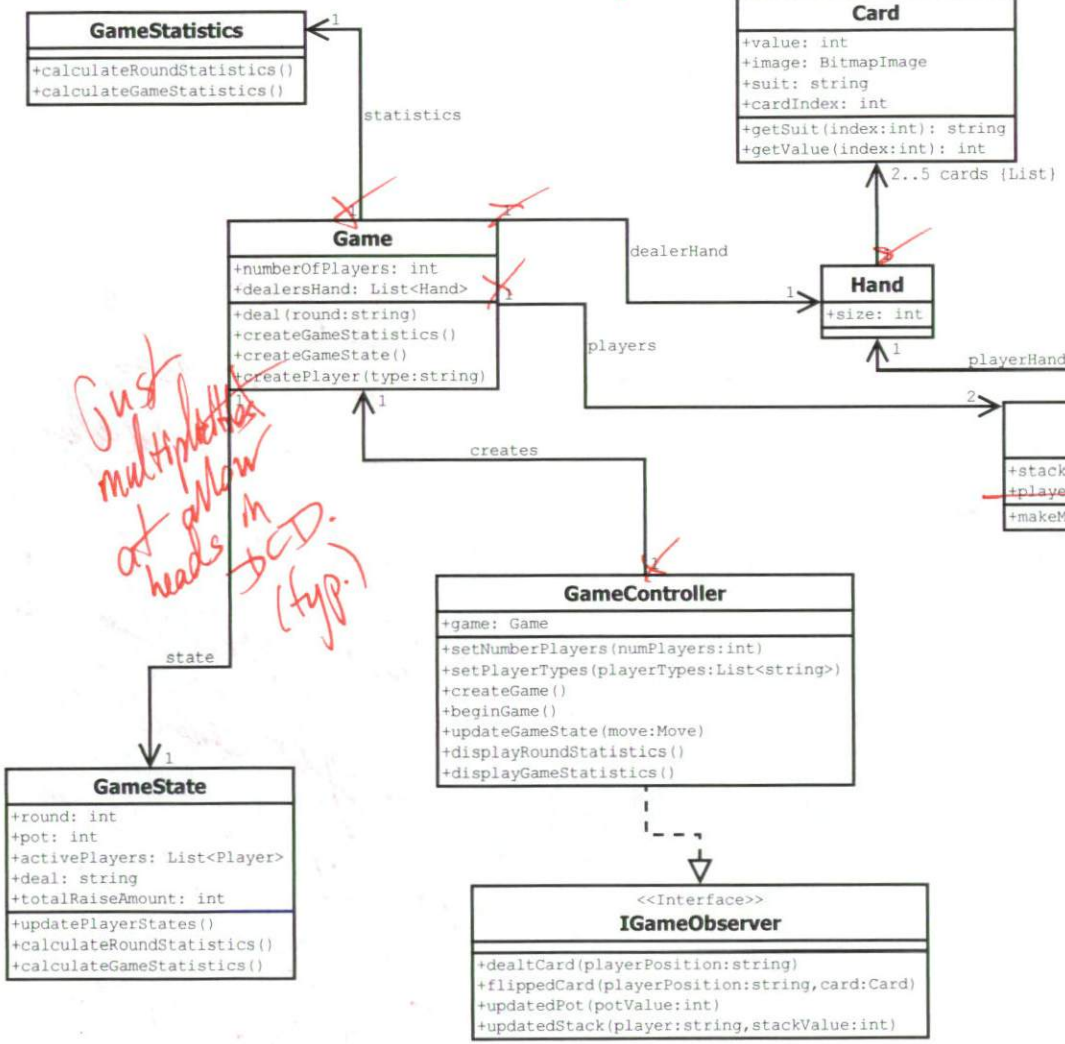

SD5: makeMove
:Move
:GameState
:Player
makeMove
updatePot(amount)
updatePlayerStack(amount)
alt
[Move == Fold]
removeActivePlayer(player)
why is makeMove() being sent to Move?
I can imagine a design where that's how it works, but you need some text describing it. It isn't obvious.
Not a sys. op. per SSDs.
where did this come from?


SD6: displayRoundStatistics
:GameController
:GameState
:GameStatistics
displayRoundStatistics
displayRoundStatistics
calculateRoundStatistics(roundData)
roundsStats
roundsStats



Shouldn't GameStats be accumulating data during game, then you can just getGameStats?

As it is, GameStats looks like it's just a calculator w/out state. that's not OO.



Inconsistent w/ interaction diagrams.

redundant

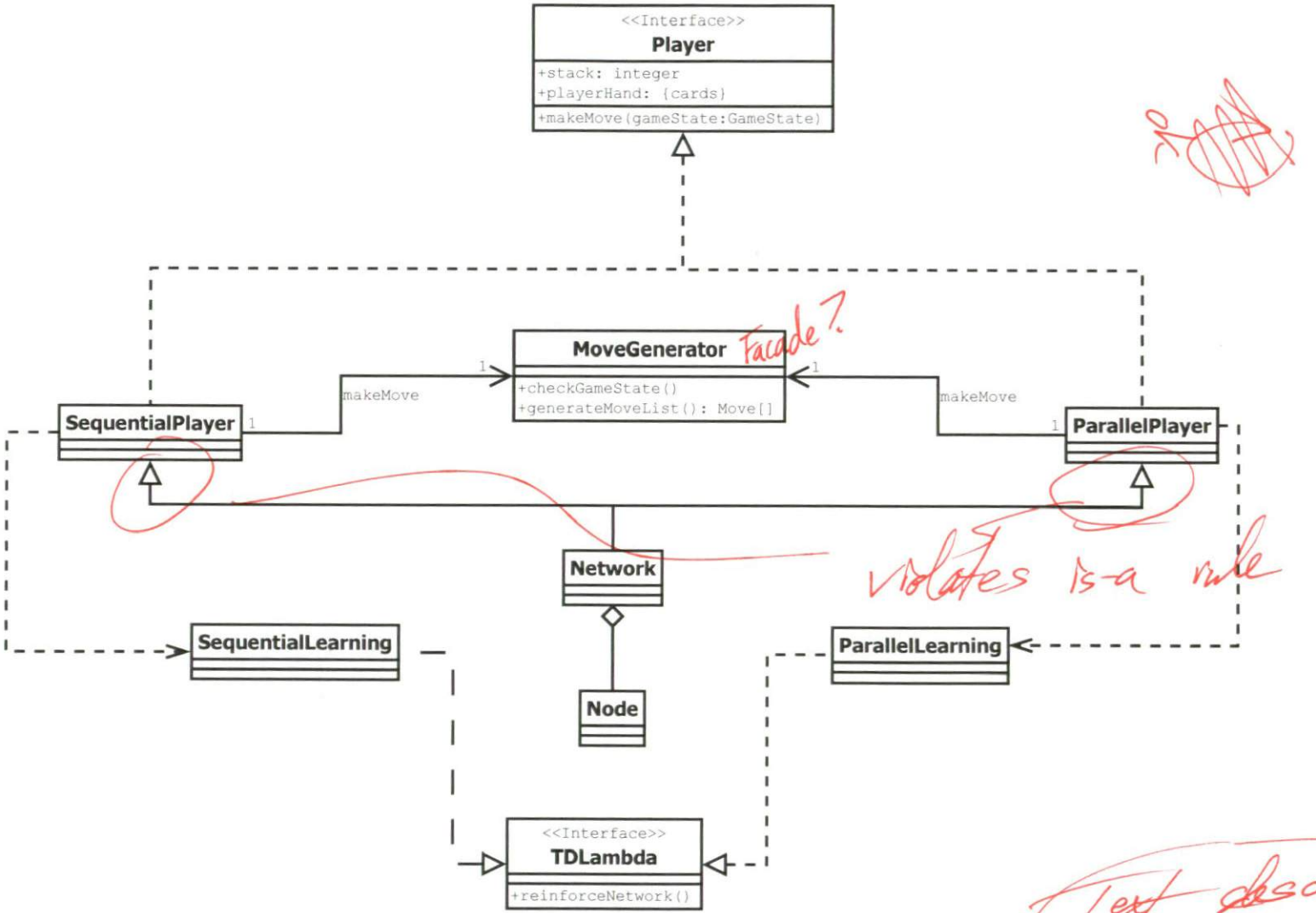
Just multiplicities at allow heads in BCD. (typ.)

this says that Player stores a list of moves in an attribute named 'creates'

Heading?

Test description

What all these. Does every subtype really have these attributes.


<<Interface>>
Player
+stack: integer
+playerHand: {cards}
+makeMove(gameState:GameState)
MoveGenerator
Facade?
+checkGameState()
+generateMoveList(): Move[]
makeMove
makeMove
1
1
1
1
SequentialPlayer
ParallelPlayer
Network
violates is-a rule
SequentialLearning
ParallelLearning
Node
<<Interface>>
TDLambda
+reinforceNetwork()
Doesn't actually wall-off AI behind a façade
Test description

GRASP Patterns

Low Coupling

In order to achieve low coupling, we made sure that there were no unnecessary relations between classes. For instance, we considered having Card related to Game and Hand. Since we considered the Low Coupling pattern, we realized that we only needed Card to be coupled with Hand, since Card was already coupled with Hand and Hand was already coupled with Game.

High Cohesion

In order to have high cohesion, we made sure that our classes all have a number of highly related methods that do not too many operations. For instance, the only method that Player has is makeMove(). Before, we had considered putting methods about calculations of statistics in Player. However, based on the High Cohesion principle, we decided to have only highly related methods in Player.

Information Expert

We used the Information Expert pattern to identify what class should have any operations dealing with changing the GameStatistics class. Since Game has all the information about the game, such as number of players and player types, then Game was the appropriate choice based on the Information Expert pattern. Therefore, Game has the operations related to modifying and creating GameStatistics.

Creator

We used the Creator principle to decide what class should create the Players. Since the Game contains the Players, closely uses the Players, and has the data to initialize the Players, Game was a good choice for the Creator of the players (SD3).

Controller

We used the Controller pattern to decide what class should coordinate system operations. We decided to use a Façade Controller, because the operations are coming in over a single pipeline. Namely, all the operations are coming directly from the GUI. In addition, there are not many system operations. There really is only two main ones: setPreferences (SD1) and beginGame (SD2). Therefore, a Façade Controller was appropriate. Our Façade Controller is our GameController class. It delegates tasks to other objects and controls the activity in the system.

Polymorphism

We used Polymorphism when deciding how to deal with the different types of players. Since ParallelPlayer and SequentialPlayer were very similar but have a few different behaviors, we decided to have an abstract class, Player, and then two subclasses, ParallelPlayer and SequentialPlayer. This will also be useful for any future versions of the system. If a developer wants to add a different type of player, he can easily add another subclass to Player.

Indirection

Since we did not want to have Player directly coupled with the GameState, we used the Indirection pattern. We used an intermediate object, Game, to mediate between GameState and Player. That way, we did not have to have extra coupling between Player and GameState.

Pure Fabrication

Information Expert would lead us into high coupling with regard to presenting the details of the poker game to the user. For instance, each Player would have to output its data about the move decision to the GUI and the game would also have to be related to the GUI. Instead of following the Information Expert pattern for game statistics, we used the Pure Fabrication pattern. We made a GameStatistics class. GameStatistics is used to summarize the statistics about the game without having too high coupling. We were careful to avoid a common contraindication of Pure Fabtrication, which is too much data being passed to other objects for calculations. We made sure that GameStatistics does not request many calculations to be done by other classes.

Protected Variations

The most unstable part of the system is the neural network. Therefore, we used the Protected Variations pattern to create a stable interface around the neural network. We created a MoveGenerator class that is between the system components and the neural network. That way, if the neural network needs to be changed in the future, it can easily be altered without affecting the whole system.

move to diagrams while they appear.

good analysis

or can upload w/ a different move selection strategy

Design is a team activity. How much of this work is being done as a team? The number of inconsistencies in the diagrams gives the appearance that the design is being divided up. It's fine to divide up diagram digitizing, but actual design should really be done together.

Concurrent Poker Player

CSSE 374 – Software Architecture and Design I

Completeness Checklist for Milestone 5

- [x] Domain model
- [x] System sequence diagrams
- [x] Operation contracts
- [x] Logical architecture
- [x] Interaction diagrams
- [x] Design class diagram(s)
- [x] Analysis of the GRASP Principles
- [ ] ~~Analysis of the use of GoF Patterns~~
- [ ] ~~Acceptance test plan~~
- [x] Code
- [ ] ~~Functional Demonstration~~
- [ ] ~~Demonstration of correspondence between design documents and code~~

Scoring Rubric for Milestone 5

Criteria (weight)	5 Exemplary	3 Satisfactory	1 Needs Improvement	Weighted Score
Professionalism (×2)	Document is neatly drawn and formatted. (Apart from any problems with the notation) it could be shared with a stakeholder without changes. Document is free of errors in spelling, grammar and punctuation.	Document is somewhat sloppy, but could be shared with a "real-world" stakeholder after some revisions. Document has a small number of errors in spelling, grammar, or punctuation.	Document is largely unprofessional. It would have to be largely reworked before sharing the document with a savvy stakeholder. Document has many errors in spelling, grammar, and punctuation.	6
Cohesiveness (×2)	The parts of the document reinforce each other. Each piece is consistent with the others and the document as a whole tells a story.	The parts of the document mostly reinforce each other. Each piece is generally consistent with the others with just a few minor differences.	The parts of the document are disjointed. They are largely inconsistent, to the point that it is unclear whether they describe the same system.	4
Clarity of Diagrams (×2)	Diagrams are well labeled and at an appropriate level of abstraction so that stakeholders familiar with the problem domain could readily understand them.	Diagrams are mostly well labeled, with no more than 15% cryptic labels. Diagrams are generally at an appropriate level of abstraction, though a stakeholder familiar with the problem domain might need some guidance to understand them.	Labels are often cryptic or abstraction is used to the point that the actual analysis and design implications would be obscured to all but an expert in both the notation and the domain.	10
Conciseness of Diagrams (×1)	Diagrams appropriately use the abstraction features of the notation to minimize useless redundancy	Diagrams may include some unhelpful redundancy, but the general representations are still readily comprehensible	Diagrams are highly redundant to the point that they are difficult to comprehend.	5
Effectiveness of Analysis (×2)	Analysis artifacts identify all important domain concepts and clearly define the system interface. They demonstrate a deep understanding of the problem domain.	Analysis artifacts identify many important domain concepts and define the system interface. They demonstrate a reasonable understanding of the problem domain.	Analysis artifacts identify only a few of the domain concepts or only cursorily define the system interface. They betray a superficial understanding of the problem domain.	10
Effectiveness of Design Models (×3)	Design conveys all important elements, constructs, and behaviors. It demonstrates a deep understanding of the solution to the problem.	Design conveys many key elements, constructs, and behaviors. Some situations might be treated in an unusual manner, but such treatment is documented.	Design minimally conveys key elements, constructs, and behaviors. It shows a superficial understanding of the problem and its solution.	9

Criteria (weight)	5 Exemplary	3 Satisfactory	1 Needs Improvement	Weighted Score
Correctness of Solution (×3)	The design is viable within assumptions and rationale presented. Key tradeoffs are successfully analyzed and defended.	The design is largely viable within assumptions and rationale presented. Key tradeoffs are presented, but may not be fully or clearly analyzed.	The viability of the design is questionable. Some assumptions and rationale lacking. Key tradeoffs are missing or may be poorly analyzed.	6
Elegance of Solution (×2)	Design effectively applies GRASP principles and design patterns to reduce coupling, increase cohesion, and lower the representation gap.	Design often applies GRASP principles and design patterns to reduce coupling, increase cohesion, or lower the representation gap	Design does not seem to apply GRASP principles and design patterns. It is ad hoc and does not demonstrate commonly accepted design practices.	8
Discussion of Patterns (×2)	Document discusses the application of design patterns such that design decisions are clearly communicated and supported.	Document discusses the application of design patterns, demonstrating a basic understand of the patterns, but not consistently showing how those patterns informed the design decisions made.	Document discusses design patterns in a cursory manner or not at all.	8
Correct Use of Notation (×2)	All notation used in the diagrams is appropriate to the diagram type and is used correctly.	All notation used in the diagrams is appropriate to the diagram type. At most two sorts of errors are made in the application of each diagram type.	Diagrams use notation inappropriate to the diagram type or contain a large variety of errors in the application of the notation.	10
Software Demonstration ~~(×4)~~ × 9	Software is free of obvious defects. ~~Demonstration told a story. The important features of the system were covered in a compelling way that made clear how the problem was solved from the user's perspective.~~	Software shows no more than 4 obvious defects. Demonstration provided concise, but thorough review of the system that made clear how the problem was solved from the user's perspective.	Software shows 4 or more obvious defects. ~~Demonstration was either incomplete or was just a litany of features.~~	~~20~~ 45
Software Style (×1)	Code is clear and well documented with consistent and appropriate naming and formatting. No "magic numbers" are used.	Code is mostly clear and well documented. The majority of identifiers are well named and the formatting is mostly consistent. No "magic numbers" are used.	Code is often unclear or undocumented. Obscure or terse identifiers are the norm. Formatting may be inconsistent. "Magic numbers" may be used.	
Correspondence of code and design (×4)	Code for the system is consistent with design diagrams, both structurally (as documented by Design Class Diagrams) and behaviorally (as documented by Interaction Diagrams).	Code for the system is mostly consistent with design diagrams apart from a few minor discrepancies.	Code for the system is inconsistent with the design diagrams.	
			Subtotal Score (Sum of above / ~~1.5~~ 1.5):	~~76.8~~ 80.7
			~~(Subtotal Score)~~ × (% of Assignment Completed):	100%
			= Total Score:	~~77~~ ~~80~~ 81

N/A for MS4