1.1

Having pockets -> having good usable data structures
Fabric color -> syntax highlighting
Being stylish -> up-to date/latest version
Comfortable -> large standard library (i.e. there are enough features to keep you “comfortable” when programming)
Inexpensive -> algorithm speed/memory costs
Material -> ease of understanding of syntax

1.2

```
public class NameMaker {
      private String name;
      public NameMaker() {
            this.name = ‟world”;
      }
      public NameMaker(String name) {
            this.name = name;
      }
      public NameMaker(String name, String suffix) {
            this.name = name + ‟ ‟ + suffix;
      }
      public String whatToSay() {
            return ‟Hello ‟ + this.name;
      }
}
```

1.3

- *A UPS driver, who goes from house to house, picking up and dropping off packages, where the packages' approximate places in the truck are determined by their eventual destination, whether inbound or outbound*:
    - LinkedList (each package “points” to the location of the next one, can easily add packages to the end.)

- *The United States Postal Service, who has assigned nine-digits (“Zip plus 4”) codes to geographical regions of the country, to route all mail.*
    - Array (there are a fixed number of zip codes).

- *The bouncer at a popular social establishment, who keeps patrons waiting in a line, but who may allow special handling for distinguished guests or people he doesn't like.*

    - ArrayList (want to easily be able to add elements and reorder existing elements.)

Other examples:

- An apartment complex with labeled apartment numbers is like an array since it has exact locations for each apartment and they are fixed. (Apartments won’t be added)

- A LinkedList is like a conga line since people can enter and leave the conga line whenever and wherever, but they always stay connected to the leader.

- An ArrayList is like a Rolodex contact holder since more contacts can be added to the end of the rolodex and the contacts are all indexed by name.

1.4

(One example for each type listed, answers may vary)
Class: What kind of place do you have?
Methods: What is the “it” that the robot must enter into the system?
Field: Does the robot need to keep track of individual customers?
Performance Requirement: How do you define “good” conversation?
Incompleteness: Does the robot need to collect payment?
Redundancy: Why do you need the “system” if the robot is keeping track of orders?

1.5

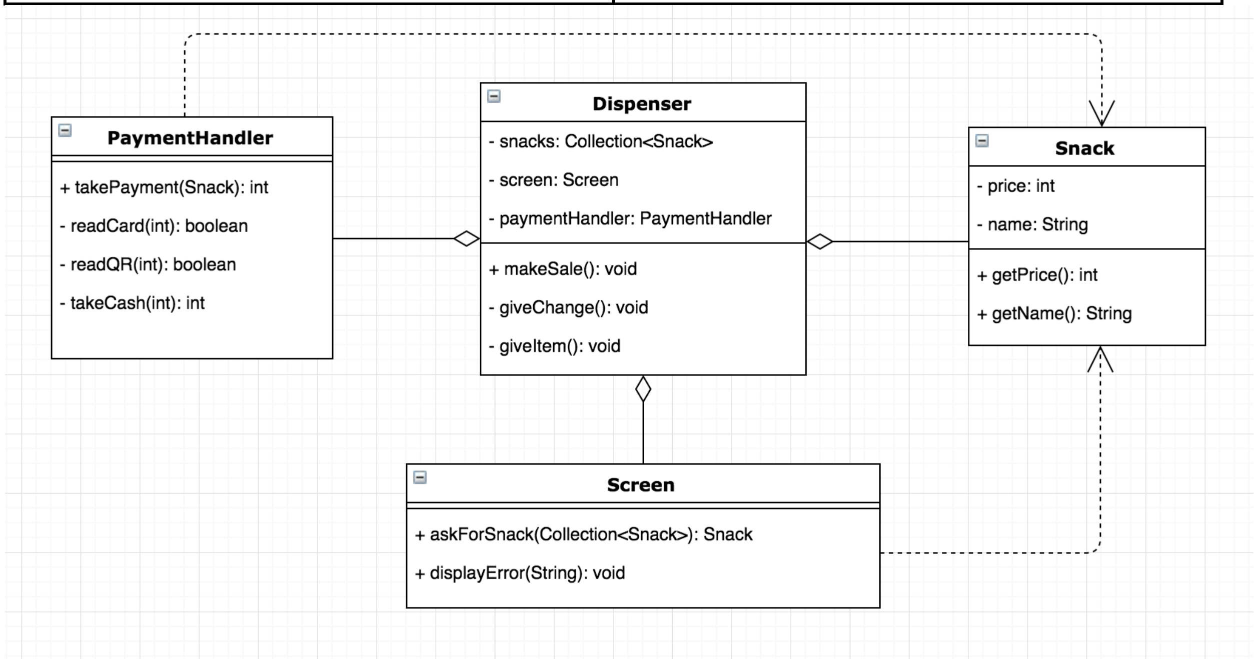
Note: it is neither incorrect nor necessary to include subclasses for different kinds of snacks

Dispenser	
Keeps track of availability Dispenses item Makes change	Screen Snack PaymentHandler

PaymentHandler	
Takes cash Reads credit card Reads QR code Decides whether price is met Calculates change	Snack

Screen	
Asks user which item they’d like to buy Displays error message	Snack

Snack	
Knows its own price and name	`



1.6

Game	
Starts RPSLK game. Organizes interaction between user and game.	Statistics Talker Ruler Player

Talker	
Gets user input. Produces text output to the user.	Statistics Game Ruler

Statistics	
Keeps track of game statistics.	Game Ruler

Ruler	
Keeps track of the valid game rules. Determines game winners.	Game

Repeater	
Plays the same move repeatedly.	Player Game

Recorder	
Keeps track of opponent’s past moves.	Player Ruler

Randomizer	
Plays random moves.	Player

Rotator	
Plays each move cyclically.	Ruler Player

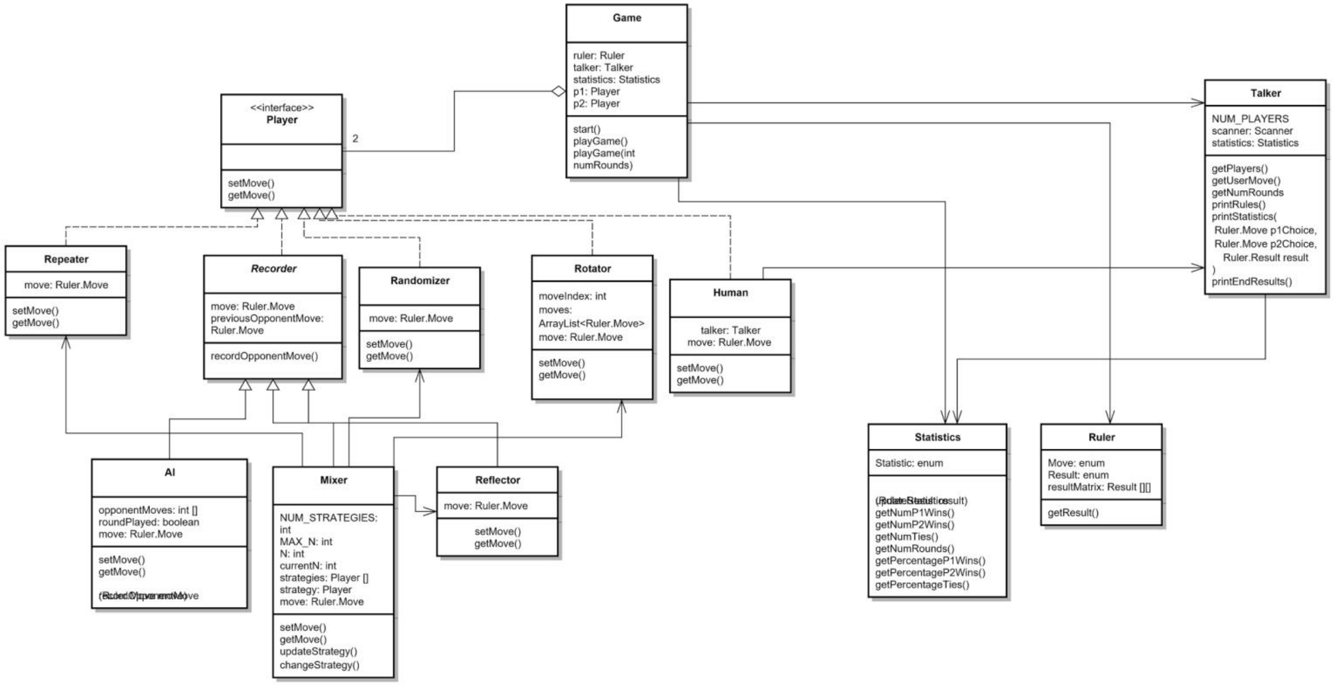
Human	
Plays the user’s move.	Talker Player

AI	
Predicts the opponent’s next move to play.	Player Ruler

Mixer	
Plays the repeater, randomizer, reflector, and rotator strategies.	Repeater Recorder Randomizer Reflector Rotator Player

Reflector	
Plays the move the opponent just played.	Recorder Player

1.7


Game
ruler: Ruler
talker: Talker
statistics: Statistics
p1: Player
p2: Player
start()
playGame()
playGame(int
numRounds)
<<interface>>
Player
setMove()
getMove()
2
Repeater
move: Ruler.Move
setMove()
getMove()
Recorder
move: Ruler.Move
previousOpponentMove:
Ruler.Move
recordOpponentMove()
Randomizer
move: Ruler.Move
setMove()
getMove()
Rotator
moveIndex: int
moves:
ArrayList<Ruler.Move>
move: Ruler.Move
setMove()
getMove()
Human
talker: Talker
move: Ruler.Move
setMove()
getMove()
AI
opponentMoves: int []
roundPlayed: boolean
move: Ruler.Move
setMove()
getMove()
(RuledOpponentMove
Mixer
NUM_STRATEGIES:
int
MAX_N: int
N: int
currentN: int
strategies: Player []
strategy: Player
move: Ruler.Move
setMove()
getMove()
updateStrategy()
changeStrategy()
Reflector
move: Ruler.Move
setMove()
getMove()
Talker
NUM_PLAYERS
scanner: Scanner
statistics: Statistics
getPlayers()
getUserMove()
getNumRounds
printRules()
printStatistics(
Ruler.Move p1Choice,
Ruler.Move p2Choice,
Ruler.Result result
)
printEndResults()
Statistics
Statistic: enum
(updateStatisticsesult)
getNumP1Wins()
getNumP2Wins()
getNumTies()
getNumRounds()
getPercentageP1Wins()
getPercentageP2Wins()
getPercentageTies()
Ruler
Move: enum
Result: enum
resultMatrix: Result [][]
getResult()

1.8

Names:
class Statistics: this is a noun that encompasses the purpose of the entire class, which is to keep track of and return statistics on the game.
getNumRounds() follows the guidelines for method names because it is an accessor prefixed with get.
numRoundsUntilSwitch: this field accurately describes the intention of the variable because the purpose is in its name - it is the number of rounds in which the Mixer plays a certain strategy before switching to a new one.

Methods:
updateStatistics() follows the rule that functions should be small.
getResult() performs one function, which is to decide which player won based on the moves passed into the method.
recordOpponentMove() is descriptive of its intended functionality, which is to record the opponent's last move.

Comments:
The comment in getUserMove() is useful to explain that the while loop continues until a valid move is given.
The comment in the Mixer's constructor clarifies the purpose of randomizing numRoundsUntilSwitch, because otherwise, it would not be obvious why the number of rounds to play a strategy might differ between instances.