Software Design Document (SDD) Template

Software design is a process by which the software requirements are translated into a representation of software components, interfaces, and data necessary for the implementation phase. The SDD shows how the software system will be structured to satisfy the requirements. It is the primary reference for code development and, therefore, it must contain all the information required by a programmer to write code. The SDD is performed in two stages. The first is a preliminary design in which the overall system architecture and data architecture is defined. In the second stage, i.e. the detailed design stage, more detailed data structures are defined and algorithms are developed for the defined architecture.

This template is an annotated outline for a software design document adapted from the IEEE Recommended Practice for Software Design Descriptions. The IEEE Recommended Practice for Software Design Descriptions have been reduced in order to simplify this assignment while still retaining the main components and providing a general idea of a project definition report. For your own information, please refer to IEEE Std 10161998[1] for the full IEEE

Recommended Practice for Software Design Descriptions.

[1] http://www.cs.concordia.ca/~ormandj/comp354/2003/Project/ieeeSDD.pdf

Team 08

VoteEasy

Software Design Document

Prepared by

Jashwin Acharya (achar061),
Steve Petzold (petzo017),
Vincent Hoang (hoang317), and
Carlos Chasi-Mejia (chasi009)

Lab Section: N/A

Workstation: N/A

Date: 10/27/2023

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose

The purpose of this software design document is to explain with great detail the architecture and system design of the ***VoteEasy*** software.

1.2 Scope

The purpose of ***VoteEasy*** is to provide Election Officials to automate Ballot calculations for an election to determine a winner without significant manual labor. ***VoteEasy*** provides an easy-to-use interface for a user to simply input a CSV file containing Voting Protocol, Candidate, Party Affiliation, and Ballot information, and automatically performs the ballot calculations depending on the voting protocol (Instant Runoff or Open Party List) specified in the file and produces an audit file containing step-by-step details of how the election progressed. The program is designed to be run multiple times in a year and is guaranteed to determine the correct winner 100% of the time.

1.3 Overview

The following document aims to provide in-depth information about the various components of our system, how different components are expected to interact with each other, as well as information about the planned design of our user interface. The document is organized as follows:

- Section 1: This provides an overview of the intended product.
- Section 2: This section describes the overall functionality and design of our system.
- Section 3: This section provides an overview of our system architecture, design rationale, and Object-oriented descriptions of every component in our system. We also include the Activity and Sequence diagrams for the IR and OPL voting processes respectively along with a summary of what each diagram represents.
- Section 4: This section provides an overview of what type of data structures are utilized by our system to store data relevant to performing ballot calculations and tracking Candidate and Party affiliation information. We also provide an alphabetical list of each member function for each class in our system.
- Section 5: This section provides pseudocode for the different member functions for each class in our Object-oriented system design.
- Section 6: This section provides an overview and mock diagram of the user interface our users will be interacting with as well as how winner information will be displayed to the user.
- Section 7: This section provides cross-reference details between each system component and the SRS requirement it refers to.
- Section 8: This section provides supplementary information for our system (if any) which can include extra UML or Activity diagrams to expand on the functionality of our system.

1.4 Reference Material

No other documents were used as reference material.

1.5 Definitions and Acronyms

Some acronyms that might be used in the document are as follows:

- IR - Instant Runoff
- OPL - Open Party List
- CLI - Command Line Interface

2. SYSTEM OVERVIEW

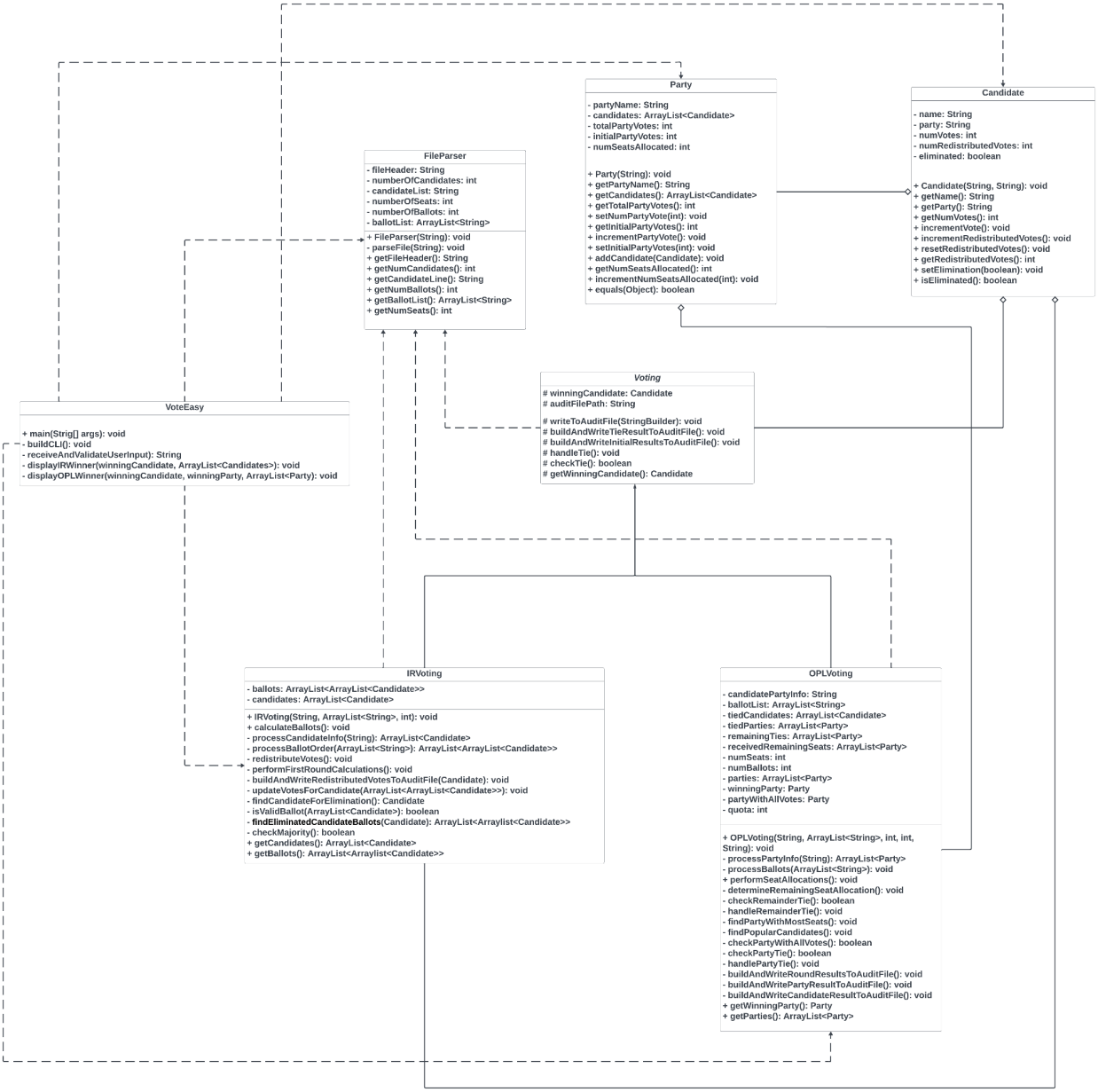
The general functionality of ***VoteEasy*** includes the following:

- A command line interface that allows a user to input a CSV file containing Voting Protocol, Candidate, Party, Ballot allocation, and, optionally, number of seats information.
- Functionality for parsing the file for all the aforementioned information in the above bullet point.
- Functionality for producing an audit file which provides a step-by-step progression of the whole election.
- Functionality for performing IR Voting and declaring a winner by determining the Majority winner or by taking popularity into account when a clear majority is not achieved in the first round of ballot calculations.
- Functionality for performing OPL voting by determining a winner through quote and seat allocation calculations.
- Functionality for handling a tie if it occurs for either voting protocol.
- Functionality for showing winner information on the screen along with a statistical breakdown of how many votes each candidate received.

3. SYSTEM ARCHITECTURE

3.1 Architectural Design

The UML Diagram of our system is below:



Note: A clearer version of the UML diagram is available in this current folder and the file is titled as "UML_Team08.pdf".

The UML diagram above shows how our system has been decomposed into smaller individual components, each working together to process ballots and determine the winner of the election. A brief description of each class/interface component in the UML diagram is as follows:

- **VoteEasy:** This class houses the main() function which contains code for building our user interface and calling the necessary functions for parsing the input CSV file and

calling the appropriate Voting Protocol functions once the file information is parsed. Once a winner is declared, we can display the winner and other candidate/party details using helper functions defined in this class.

- **FileParser:** This FileParser class is responsible for parsing and gathering all necessary information regarding the input CSV election file such as: voting protocol, # of candidates, list of candidates, # of ballots, list of ballots, and # of seats. This function is called within the VoteEasy main() function and primarily consists of getter functions for the attributes listed above which are set through a parsing function defined in the class.

- **Party:** The party component is responsible for storing Party related attributes has such as party name, an ArrayList of candidates, a count of the number of votes a party has, and a number of seats that have been allocated to the specific party.

- **Candidate:** The candidate component is responsible for storing candidate related information such as the name of the candidate, the name of the party, and the number of votes the candidate has.

- **Voting:** This is an abstract class that defines variables and functions that are inherited by two derived classes: IRVoting and OPLVoting. This abstract class does not implement any method; however, both derived classes implement every method declared in the Voting abstract class.

- **IRVoting:** The IRVoting class is responsible for performing ballot calculations when the IR voting protocol is specified in the header of the input CSV file. This class will also handle vote ties and provide functionality for redistributing votes in-case a majority vote hasn't occurred during the first round of voting, and subsequent rounds as well.

- **OPLVoting:** The OPLVoting class handles ballot calculations and determines a winner by implementing the OPL Voting protocol.

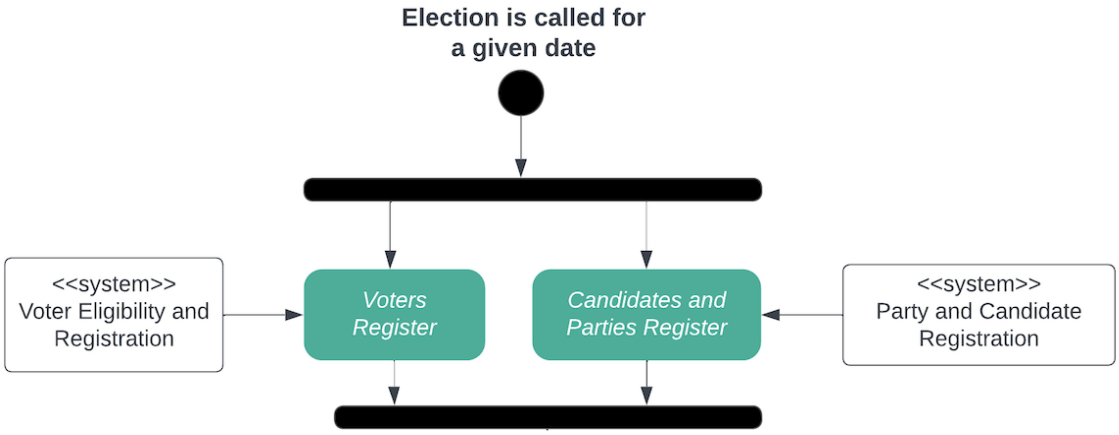
The interactions between all classes are explained as follows:

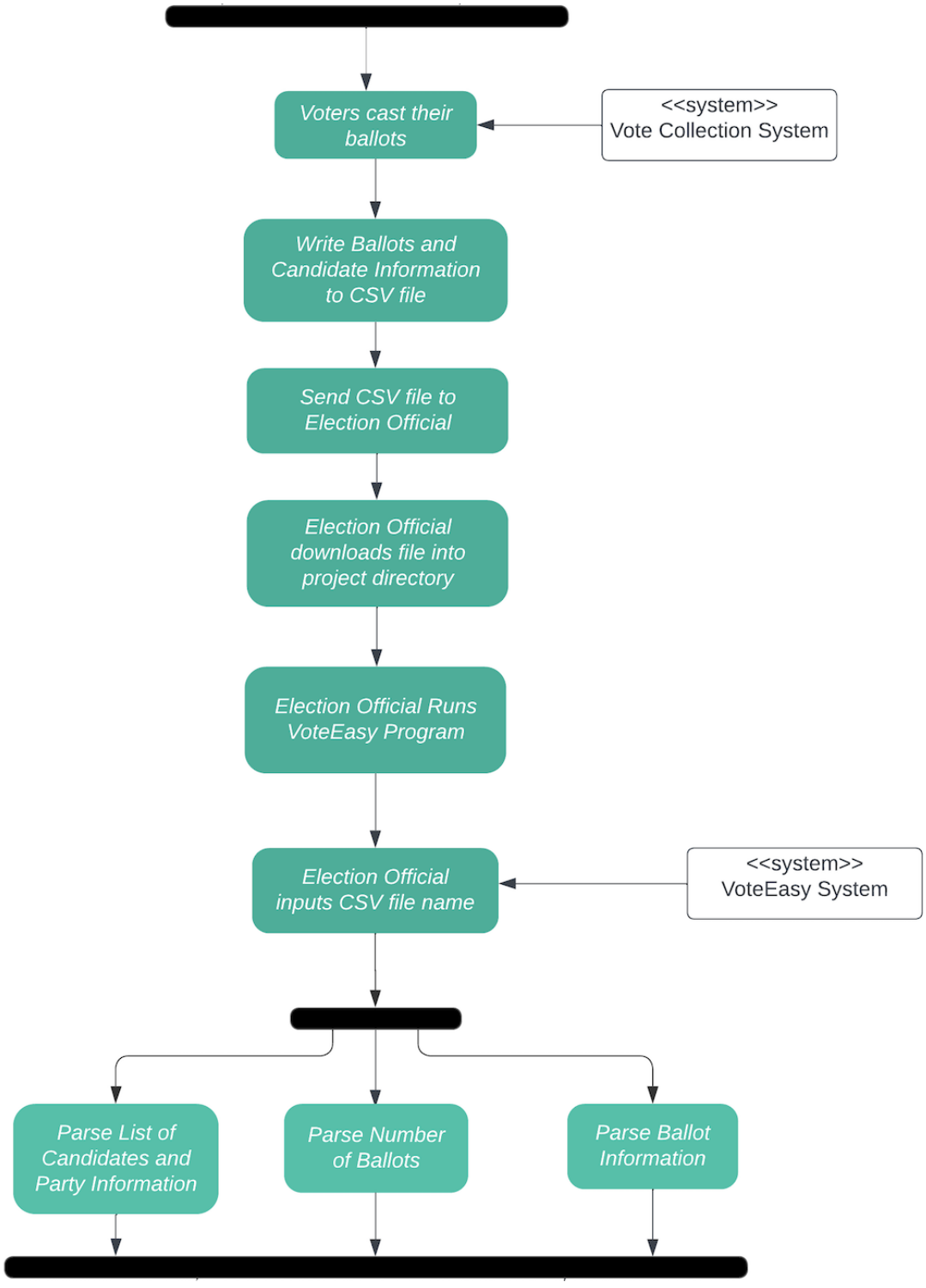
- Class VoteEasy has a dependency relationship with Classes FileParser, Party, Candidate, OPLVoting and IRVoting. This is because VoteEasy contains the main() function that supplies the FileParser class with the CSV file that contains all the information needed for ballot calculations and declaring a winner candidate/party. The parsed CSV file values are then later on passed into OPLVoting or the IRVoting class depending on which voting protocol is specified and these interactions from the VoteEasy class are imperative for declaring a winner for the election. The VoteEasy class also interacts with the Party and Candidate class as it is dependent on the information stored in these classes to display winner information on the screen.
- The Voting abstract class, OPLVoting and IRVoting classes have a dependency relation to the FileParser class since they require the CSV file to be parsed correctly in order to receive candidate and ballot information necessary to declare a winner.

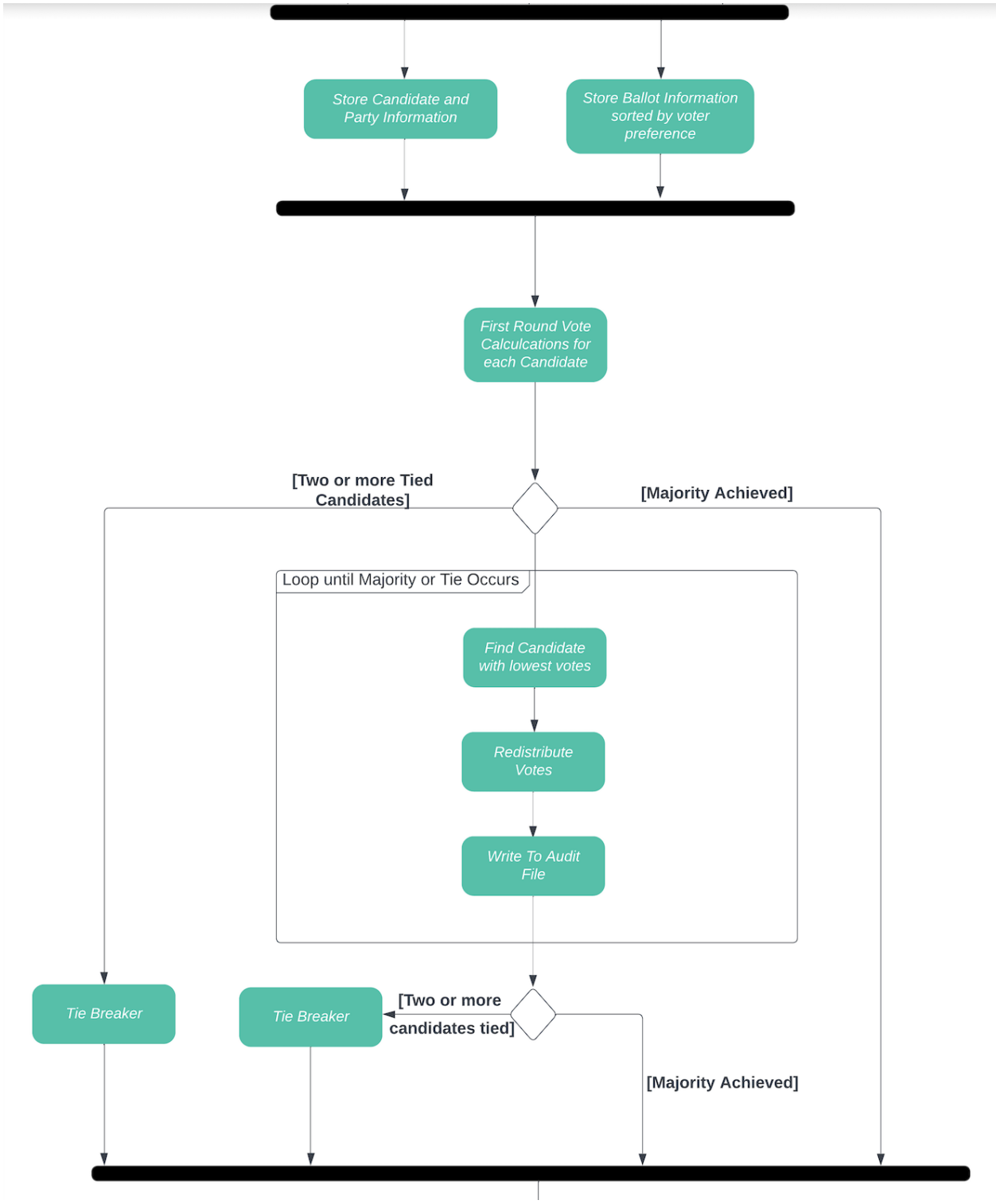
- The Voting abstract class has an aggregate relationship with the Candidate class since the interface houses member variables with type Candidate.
- Class OPLVoting has an aggregate relationship with the Party class since OPLVoting houses variables that track lists of parties and other member variables with type Party.
- Classes IRVoting and OPLVoting inherit from the Voting abstract class and thus have a generalization relationship with the Voting abstract class.
- Party has an aggregate relationship with the Candidate class since the Party class contains an arraylist of Candidate objects for keeping track of each candidate assigned to that party along with how many votes each candidate received.

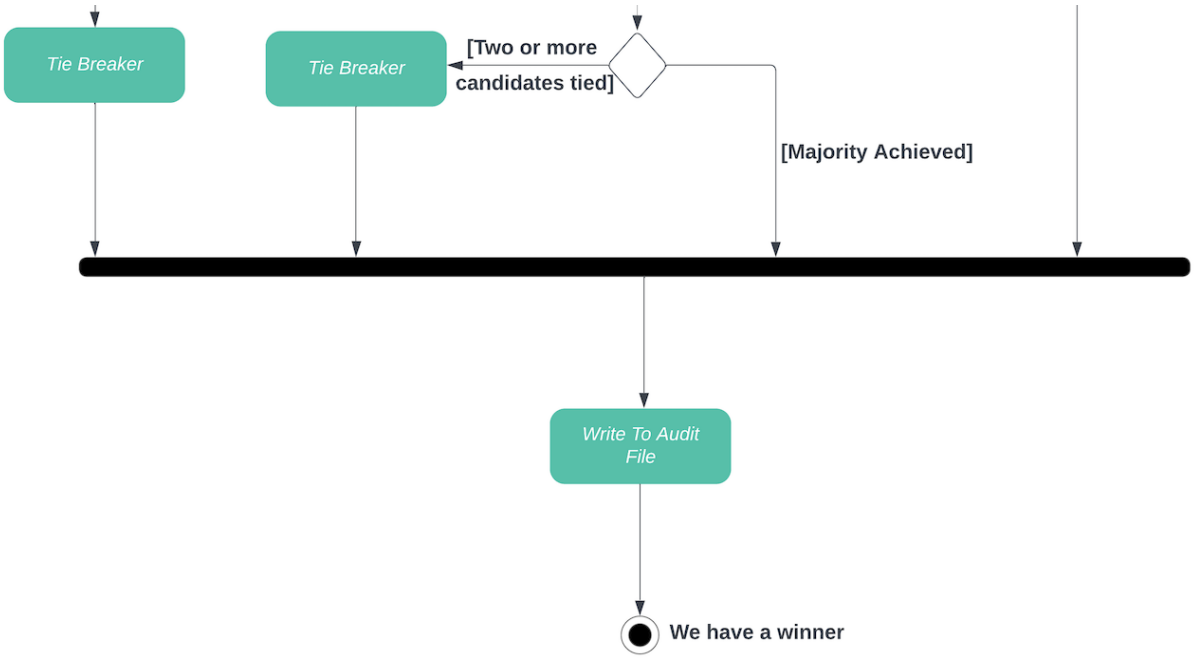
3.2 Decomposition Description

Following is a high level Activity diagram showing how a winner is declared when using the IR Voting protocol to calculate an election result:




Voters cast their ballots
<<system>>
Vote Collection System
Write Ballots and Candidate Information to CSV file
Send CSV file to Election Official
Election Official downloads file into project directory
Election Official Runs VoteEasy Program
Election Official inputs CSV file name
<<system>>
VoteEasy System
Parse List of Candidates and Party Information
Parse Number of Ballots
Parse Ballot Information

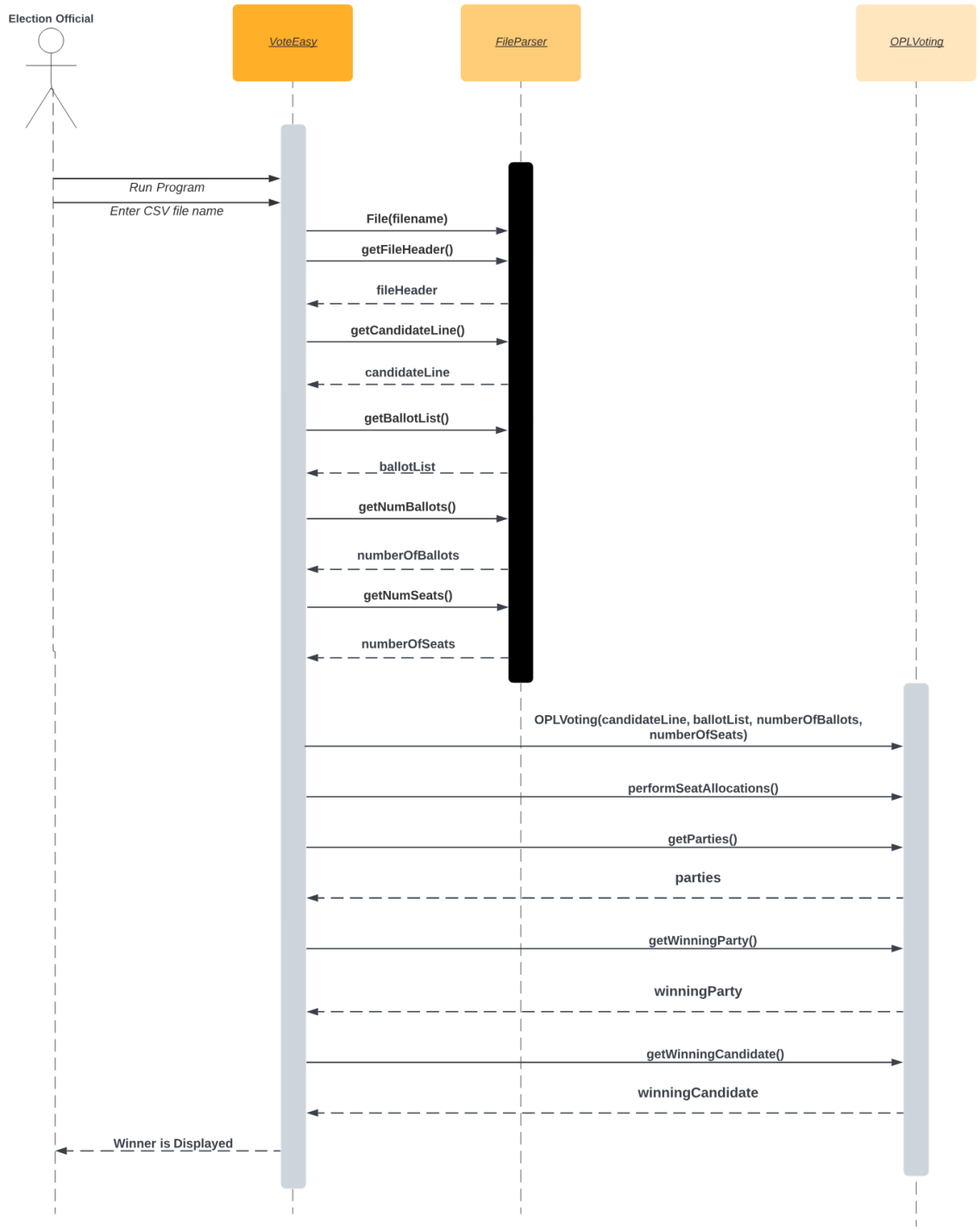

Store Candidate and Party Information
Store Ballot Information sorted by voter preference
First Round Vote Calculcations for each Candidate
[Two or more Tied Candidates]
[Majority Achieved]
Loop until Majority or Tie Occurs
Find Candidate with lowest votes
Redistribute Votes
Write To Audit File
Tie Breaker
Tie Breaker
[Two or more candidates tied]
[Majority Achieved]



Note: A more clear version of the activity diagram is available in the in the same directory as this document under the name "ActivityDiagram_Team08.pdf".

Summary of Activity Diagram: Starting from the top, once an election is declared, people can register as voters using the "Voter Eligibility and Registration" System which ensures that the voters are of legal age and eligible to vote as citizens. Candidates and parties can register through a separate registration system called the "Party and Candidate Registration" System. Once voters have registered, they can begin voting and their ranked ballots are collected by a separate "Vote Collection System". Once the voting process is complete, ballot and candidate information is written to a CSV file that is subsequently and securely emailed to an election official. The election official then uses the VoteEasy system to automate the ballot calculations. The file is first parsed by the VoteEasy system for candidate, party and ballot information before proceeding to the storage step where we construct lists of Candidate objects and a ballots list where each element is a list of candidates sorted according to a voter's preference. We then perform the first round of vote calculations and check if we have a majority or a tie. If a tie has occurred, we enter a tie breaker event to declare a winner. If neither have occurred, that means we have to redistribute the votes of the lowest candidates and write the result of each redistribution round to the audit file. The redistribution process continues until a majority or tie is reached, after which we write the final result of the majority or tie break result to the audit file and declare the winner of the election.

Sequence Diagram for OPL is below:



Note: A clear version of this diagram is available in the same folder as this document and the file is titled "SequenceDiagram_Team08.pdf".

Summary of Sequence Diagram: Once an election official runs the VoteEasy program and enters the correct CSV file name, the file is parsed by the FileParser class for information such as name of the candidates and party affiliations; the number of ballots; number of seats; and the list of ballot lines containing information about which candidate and party an eligible voter voted for. Once the entire CSV file is parsed, we instantiate an OPL class object with the aforementioned parameters and call the performSeatAllocations() function that is responsible for performing multiple rounds of seat allocations until no seats are left un-assigned. The performSeatAllocations() function calls other private functions defined in the OPLVoting class to determine who the winning Party and candidate is along with writing party and candidate information to the audit file. Once a winning party and candidate are determined, the flow of control comes back to the VoteEasy class which retrieves information for all parties, the winning party and candidate and passes these parameters to a private function called DisplayOPLWinner() which displays the result of the election to the end user.

Apart from just the Sequence and Activity diagrams, following is a summary of what the member functions of each class in our system offers.

Class VoteEasy houses the following functions:

- **main(String[] args): void,** This is our main function from where program execution begins and calls are made to respective functions and classes for receiving user input, parsing the CSV file, and running ballot calculations depending on the voting protocol specified in the CSV file.
- **buildCLI(): void,** This function defines the interface that allows the user to enter the name of the CSV file containing voting protocol, candidate and ballot information etc.
- **receiveAndValidateUserPrompt(): void,** This function checks if the filename the user entered does exist in the current directory, and infinitely prompts the user for the correct file name in-case the wrong filename is entered. If “help” is entered by the user, then a help message is displayed which dictates what file type is required.
- **displayIRWinner(Candidate, ArrayList<Candidates>): void,** This function will display the details of the candidate that won an election whose ballots were calculated using the IR Voting Protocol. We can loop through our arraylist of candidates and display each candidate’s name, party affiliation, as well as the number and % of votes they received. We can also display the winning Candidate’s details separately on the UI using this function.
- **displayOPLWinner(Candidate, Party, ArrayList<Party>): void,** This function will display the details of the candidate and party that won an election whose ballots were calculated using the OPL Voting Protocol. We can loop through our arraylist of parties and display each party’s name along with the number and % of seats/votes they received. We can also display the winning Candidate and Party’s details separately on the UI using this function.

Class FileParser contains the following functions:

- **FileParser(string): void,** This is the constructor of the FileParser class and calls the parseFile() function to parse each line of the input CSV file and store the values in relevant member variables. This class houses the following member variables:
    - **fileHeader: String,** This is the header of the file that contains the name of the voting protocol to be used (IR or OPL).
    - **numCandidates: int,** This stores the number of candidates taking part in the election.
    - **candidateLine: String,** This stores the line containing candidate and party information in the CSV file.
    - **numBallots: int,** This stores the number of ballots cast in the election.
    - **ballotList: ArrayList<String>,** This stores the list of ballot lines in the CSV file.
    - **numSeats: int,** This stores the number of seats allocated for the election if the file header is "OPL".
- **parseFile(): void**, this function opens the input CSV file in read mode, reading line by line, and begins parsing and assigning the file's contents to its given attribute according to the file line. It also considers whether or not to assign a value for the "number of seats" attribute depending on the file's voting protocol.
- **getFileHeader(): string**, this function retrieves the voting protocol of the file and returns it to the caller.
- **getNumCandidates(): int**, this function retrieves the number of candidates in the election and returns it to the caller.
- **getCandidateLine(): string**, this function retrieves the line of the CSV file that contains the name of the candidate and party they belong to.
- **getNumBallots(): int**, this function retrieves the number of ballots in the election and returns it to the caller.
- **getBallotList(): ArrayList<String>**, this function retrieves the list of ballots in the input csv file and returns it to the caller.
- **getNumSeats(): int**, this function retrieves the number of seats in the election and returns it to the caller. If the voting protocol is OPL, the "number of seats" attribute would be set to null.

Class Party houses the following functions:

- **Party(String partyName):** This is the constructor that initializes the class parameters of the same name with the arguments passed to the constructor. "partyName" just refers to the name of the party.
- **getPartyName(): string,** This function retrieves the name of the party and returns the name to the caller.
- **getCandidates(): ArrayList<Candidate>,** This function retrieves the candidate objects assigned to the party and returns the list to the caller.
- **getTotalPartyVotes(): int,** This function retrieves the number of votes associated with the party and returns this number to the caller.
- **incrementPartyVote(): void,** This function will increment the number of votes a party has by 1

- **setNumPartyVote(int): void,** Sets the total number of party votes to the passed in parameter. The parameter value is obtained from the largest remainder formula. After each round of seat allocations, this function is called to ensure that we always have the up-to-date number of votes left for each Party.
- **addCandidate(Candidate): void,** This function will add a candidate to the ArrayList of candidates and this function is void.
- **getNumSeatsAllocated(): int,** This function retrieves the number of seats allocated to the party and returns the number to the caller.
- **incrementNumSeatsAllocated(int): void,** This function will increase the number of seats a party has the amount passed in as a parameter. The function does not return anything.
- **getInitialPartyVotes(): int,** This function returns the total number of party votes a party started with. This is to ensure that we can calculate the % of votes a party received later on when writing to the audit file. The "totalPartyVotes" changes constantly when performing seat allocations every round due to the largest remainder formula, and thus we make an extra copy of the total party votes before the seat allocation begins.
- **setInitialPartyVotes(int): void,** This function sets the value of member variable initialPartyVotes to the passed in parameter.
- **equals(Object): boolean,** Overrides the equals method of the super class.

Class Candidate houses the following functions:

- **Candidate(String name, String party):** This is the constructor that initializes the class parameters of the same name with the arguments passed to the constructor. "name" refers to the candidate's name and "party" refers to the candidate's party.
- **getName(): string,** This function retrieves the name of the candidate and returns the name to the caller.
- **getParty(): string,** This function retrieves the name of the candidate and returns the name to the caller.
- **getNumVotes(): int,** This function returns the number of votes associated with the candidate.
- **incrementVote(): void,** This function increments the number of votes the candidate has by 1.
- **incrementRedistributedVotes(): void,** This function increments the number of redistributed votes a candidate has by 1.
- **resetRedistributedVotes(): void,** This function sets the number of redistributed votes for a particular candidate back to 0 for each round of vote redistribution when running the IR voting protocol.
- **getRedistributedVotes(): int,** This function returns the current number of redistributed votes for a candidate.
- **setElimination(boolean): void,** This function sets the eliminated status of a candidate to the passed in value.
- **isEliminated(): boolean,** This function checks if this candidate has been eliminated or not.

Abstract Class Voting: No function definitions are provided here.

Class IRVoting houses the following functions:

- **IRVoting(String candidateLine, ArrayList<String> ballotList, int numBallots): void,** This is the constructor that initializes the class parameters of the same name with the arguments passed to the constructor. "candidateLine" is the line containing the candidate name and party affiliations; "ballotLine" contains the list of ballot lines in the CSV file; and "numBallots" refers to the total number of ballots cast for the election. It also defines an audit file name that can be used later on to create and write to the audit file.

- **processCandidateInfo(String candidateLine): ArrayList<Candidate>**, This function receives the line of the file that contains the names of the candidates and parties they are associated with, and parses the line to store the aforementioned information for each candidate in an arraylist.
- **processBallotOrder(ArrayList<String>ballotLines):ArrayList<ArrayList<Candidate>>**, This function receives a list of lines where each line contains the preference ranking for each candidate a particular person voted for. We construct an arraylist called "ballots" which stores a list of candidates that are arranged according to their ranking on a particular ballot and return it. For example, let's say we have candidates Rosen, Kleinberg, Chou and Royce, and the first two lines of ballot information in the input CSV file are [1,3,4,2] and [1,,2,]. If we take the first line ([1,3,4,2]), it essentially means that Rosen is the first choice; Royce is the second choice; Kleinberg is the third choice; and Chou is their fourth choice. After parsing the first line, our "Ballots" arraylist will look like [[Candidate(Rosen, D), Candidate(Royce, L), Candidate(Kleinberg, R), Candidate(Chou, I)]], where each candidate (object) is sorted according to the voter's preference. Similar, for the second line ([1,,2,]), Royce is the first choice, Chou is the second choice and no candidates are specified as the third or fourth choices. Thus the second element of the "Ballots" arraylist will look like [[...], [Candidate(Royce, D), Candidate(Chou, I), None, None].
- **calculateBallots(): void**, This function is the entry point for our ballot calculations and calls other private functions for performing the first round of ballot calculations, writing to the audit file, and then determining whether a majority has occurred or not. Once the first round of vote calculations are complete, the following 3 situations could occur:
    - A majority has occurred (handled by CheckMajority() which is detailed later)
    - A tie has occurred (handled by CheckTie() which is detailed later)
    - Votes need to be redistributed (handled by RedistributeVotes() which is detailed later)
- **performFirstRoundCalculations(): void**, This private function increments votes for the first choice candidate listed on each ballot.
- **checkMajority(): boolean**, This private function checks whether any candidate received a majority number of votes. If no candidate received a majority, then we return False; otherwise True.

- **checkTie(): boolean**, This function checks if all non-eliminated candidates are currently tied. It returns true if there is a tie; otherwise False.
- **handleTie(): void**, If a tie is found, then we perform a tie break by using the SecureRandom class that selects a random index and sets the "winningCandidate" member variable value to the candidate that won the election. SecureRandom is designed to be a cryptographically secure random number generator and should help ensure that we are choosing candidates fairly by essentially simulating a "coin toss" where each tied candidate has an equal probability of being chosen as the winner.
- **redistributeVotes(): void**, If a tie or majority is not found during the first round of ballot calculations, this function is called to perform the redistribution of votes using other private helper functions which are detailed below, until a majority or tie condition is reached.
- **findCandidateForElimination(), Candidate**: Find the candidate currently with the lowest number of votes by looping through the "candidates" arraylist member variable. This ensures that we perform our redistribution correctly by picking the lowest vote candidate to remove from our election.
- **isValidBallot(ArrayList<Candidate>ballot):** This function checks if the passed in ballot is valid or not i.e., does it have at least one ranked candidate who votes can be redistributed to. If the ballot is invalid, then it is removed from our ballots list entirely.
- **findEliminatedCandidateBallots(Candidate eliminatedCandidate): ArrayList<Arraylist<Candidate>>**, This function helps find the ballots that indicated our eliminated candidate as their first choice vote. Since there could be multiple ballots that indicated our candidate as their first choice, we return a subset of the "ballots" arraylist member variable; hence the return type is ArrayList<Arraylist<Candidate>>.
- **updateVoteForCandidate(ArrayList<Arraylist<Candidate>> eliminatedCandidateBallots): void**, This function simply loops through the candidate ballots that had the eliminated candidate as their first choice and redistibutes each voter's ballot to their next valid choice candidate.
- **writeToAuditFile(StringBuilder sb): void,** This function is useful for writing election results to the audit file.
- **buildAndWriteInitialResultsToAuditFile(): void,** This function builds a string containing the candidate names, parties as well as number / % of votes they all received and passes the constructed StringBuilder object to writeToAuditFile() to write the election results to the audit file after redistribution is complete or a majority is found in the first round itself.
- **buildAndWriteTieResultToAuditFile(): void,** This function uses a StringBuilder object to form a string containing the names of the candidates who were tied, and the candidate who won the tie. The StringBuilder object is then passed to the writeToAuditFile() function so that the tie result can be written to the audit file.
- **buildAndWriteRedistributedVotesToAuditFile(Candidate eliminatedCandidate): void,** This function builds a string containing the redistribution results after each round until a majority or tie is achieved. Every Candidate object contains a "redistributedVotes" variable that keeps track of how many votes were given to them during the redistribution

process, and this provides an easy way to write redistribution information to the audit file. The passed in Candidate object is the eliminated Candidate whose information we would need to write to the audit file too. The constructed StringBuilder object is then passed to writeToAuditFile() so that the redistribution result can be written to the audit file.

- **getWinningCandidate(): Candidate,** This function simply returns the value of the "winningCandidate" member variable so that we can display it on the UI from the main VoteEasy class.
- **getCandidates(): ArrayList<Candidate>,** This function returns the value of the "candidates" arraylist member variable so that we can display information on the UI for all the candidates that participated in the election.
- **getBallots(): ArrayList<ArrayList<Candidate>>,** This function returns the value of the "ballots" variable so that it can be used for testing the order of the candidates in each ballot in our unit test.

Class OPLVoting implements the following functions:

- **OPLVoting(String candidateLine, ArrayList<String> ballotList, int numSeats, int numBallots):** This is the constructor that initializes the class parameters of the same name with the arguments passed to the constructor. "candidateLine" is the line containing the candidate name and party affiliations; "ballotList" contains the list of ballot lines in the CSV file; "numSeats" refers to the total number of seats that are to be allocated and "numBallots" refers to the total number of ballots cast for the election. We also initialize the member variable "quota" which refers to the seat quota necessary for seat allocation as number of ballots divided by number of seats. It also defines an audit file name that can be used later on to create and write to the audit file.
- **processPartyInfo(String candidateLine): ArrayList<Party>**, This function receives the line containing candidate and party names as its input, and parses the line to create a list of Party objects. Each Party object contains the name of the party, the number of votes the party received, the number of seats allocated to that party, and a list of Candidate objects for storing candidate names and the number of votes each candidate receives.
- **processBallots(ArrayList<String> ballotLine): void,** This function receives a list of lines from the input CSV file where each line contains a "1" for the candidate and party the voter has voted for. For example, if we have the following candidates: Pike (D), Foster (D), Deutsch (R), Borg (R), Jones (R), Smith (I), and the first ballot line is "1,,,,,", that means that this particular person voted for Pike who belongs to the democrat party. We loop through each line of the ballots and calculate the number of votes each party receives, as well as update the votes of the candidates within these parties.
- **performSeatAllocations(): void,** This function loops through all parties and performs the seat allocation calculations using the "Largest remainder formula" by calculating a quota and assigning seats to different parties until each party is assigned a seat. Once all seat allocations are complete, we can determine the winner using the findPartyWithMostSeats() function and findPopularCandidates() function which are detailed below. We also write each round of information to our audit file.

- **determineRemainingSeatAllocation(): void,** This function determines which party will receive a remaining seat based on their remaining votes. We can first call the checkRemainderTie() function that finds the list of remainingVoteTies of parties that have the same remaining votes. If the length of remainingVoteTies is greater than 1, we know we have 2 or more tied parties with the same most remaining votes and then handleRemainderTie() is called to perform the tie breaker. If the length of tiedCandidates is equal to 1, that means we have a party with the most remaining vote and we increment the party's seat. We then write the tie result to the audit file.
- **checkRemainderTie(): boolean,** This function checks if there is a tie among the most remaining votes between parties and stores the parties with the same tied remainder votes in the remainderTies member variable. If there's a tie, then we return True; otherwise False.
- **handleRemainderTie(): void,** If a tie is found, then we perform a tie break by using the SecureRandom class that selects a random index, then increments that party's seats and adds the party to the "receivedRemainingSeats" member variable. SecureRandom is designed to be a cryptographically secure random number generator and should help ensure that we are choosing candidates fairly by essentially simulating a "coin toss" where each tied candidate has an equal probability of being chosen as the winner.
- **findPartyWithMostSeats(): void,** This function simply loops through the list of parties defined in our class structure and finds the party with the highest number of seats using simple linear search. We can set the "winningParty" variable to the party object that has the highest number of seats.
- **findPopularCandidates(): void,** Once we know who the winning Party is, this function is called to determine the winning Candidate. We can first call the checkTie() function that finds the list of tiedCandidates who have the same votes. If the length of tiedCandidates is greater than 1, we know we have 2 or more tied candidates and then handleTie() is called to perform the tie breaker. If the length of tiedCandidates is equal to 1, that means we have a majority and we can set the winningCandidate value easily. We then write the tie result to the audit file.
- **checkPartyWithAllVotes(): boolean,** This function checks to see if there is a party that received all the votes in the first round. If there is, then we return True; otherwise False.
- **checkPartyTie(): boolean,** This function checks if there is a tie among the parties and stores that result in the "tiedParties" member variable. If there's a tie, then we return True; otherwise False.
- **handlePartyTie(): void,** If a tie is found, then we perform a tie break by using the SecureRandom class that selects a random index and sets the "winningParty" member variable value to the party that won the election. SecureRandom is designed to be a cryptographically secure random number generator and should help ensure that we are choosing candidates fairly by essentially simulating a "coin toss" where each tied candidate has an equal probability of being chosen as the winner.
- **checkTie(): boolean,** This function checks if there is a tie among the candidates of the winning party and stores that result in the "tiedCandidates" member variable. If there's a tie, then we return True; otherwise False.

- **handleTie(): void,** If a tie is found, then we perform a tie break by using the SecureRandom class that selects a random index and sets the "winningCandidate" member variable value to the candidate that won the election. SecureRandom is designed to be a cryptographically secure random number generator and should help ensure that we are choosing candidates fairly by essentially simulating a "coin toss" where each tied candidate has an equal probability of being chosen as the winner.
- **writeToAuditFile(StringBuilder sb): void,** This function is useful for writing Party information and Candidate information to the audit file by using the passed in StringBuilder object.
- **buildAndWriteTieResultToAuditFile(): void,** This function uses a StringBuilder object to build a string with the names of the tied candidates and the winner of the tie. The StringBuilder object can then be passed to the writeToAuditFile() function to write the tie result to the audit file.
- **buildAndWriteInitialResultsToAuditFile(): void,** This function uses a StringBuilder object to write the initial results of an election before seat allocation is performed. This function is necessary for writing the party names and the initial set of votes they have to the audit file. Once the StringBuilder object is constructed, we can pass to the writeToAuditFile() function to write the initial results to the audit file.
- **buildAndWriteRoundResultsToAuditFile(): void,** This function uses a StringBuilder object to build a string containing information from all parties such as total number of seats allocated in the current round and remaining votes left for the next round. This StringBuilder object is later on passed to writeToAuditFile() to write the round results to the audit file. This function is called once every round of seat allocations.
- **buildAndWritePartyResultToAuditFile(): void,** This function uses a StringBuilder object to build a string containing information for all parties such as party name, total votes, total number of seats allocated after seat allocation is complete and % of Votes to % of seats. We also append a string containing the winning party's name, number seats won and total number of votes received. This StringBuilder object is later on passed to writeToAuditFile() to write the final seat allocation results to the audit file.
- **buildAndWriteCandidateResultToAuditFile(): void,** This function uses a StringBuilder object to build a string containing the winning candidate's information such as name, party name, total number / % of votes won. We do the same for the rest of the candidates too. This StringBuilder object is later on passed to writeToAuditFile() to write the final candidate popularity results to the audit file.
- **getWinningParty(): Party,** This function returns the Party that won the election so that we can display its information on the UI from the main VoteEasy class.
- **getParties(): ArrayList<Party>,** This function returns an arraylist of Party objects so that we can display information from all parties such as number or % of votes/seats received; the candidates names from each party etc on the UI which will be handled by the main VoteEasy class.

3.3 Design Rationale

The reason we went with the design described in section 3.1 is that we thought it would be better to split our functionality across different classes so that each class is responsible for a small part of the entire system and if any modifications are needed in the future, then we can modify an individual part without touching the functions defined in other classes. Distributing functionality across 7 other classes allows us to unit test each individual component of the VoteEasy system to ensure that they are working as expected and can easily perform integration testing later in our testing phase by combining all our disparate components together to ensure the overall system always predicts the correct election winner.

4. DATA DESIGN

4.1 Data Description

Our program is divided into 7 main classes, as stated in Section 3, where each class has responsibilities for storing different information for Parties and Candidates. Within classes like IRVoting and OPLVoting, we make use of arraylists to store candidate and party information, including party and candidate ties, and later on utilize these data structures for displaying the winner on the user's screen. IRVoting and OPLVoting also securely write the election details to the audit file. Class Party stores party name and other information such as number of candidates a party has and the total number of votes and seats they have. The Candidate class contains the candidate name, number of votes the candidate received and the candidate's party affiliation.

4.2 Data Dictionary

Candidate Class:

- Attributes:
    - private String name
    - private String party
    - private int numVotes
    - private int numRedistributedVotes
    - private boolean eliminated
- Methods:
    - public Candidate(String, String)
    - public String getName()
    - public String getParty()
    - public int getNumVotes()
    - public void incrementVote()
    - public void incrementRedistributedVotes()
    - public void resetRedistributedVotes()
    - public int getRedistributedVotes()

    - public void setElimination(boolean)
    - public boolean isEliminated()

FileParser Class:

- Attributes:
    - private String fileHeader
    - private int numberOfCandidates
    - private String candidateLine
    - private int numberOfSeats
    - private int numberOfBallots
    - private ArrayList<String> ballotList
- Methods:
    - public FileParser(String)
    - private void parseFile(String)
    - public String getFileHeader()
    - public int getNumCandidates()
    - public String getCandidateLine()
    - public int getNumBallots()
    - public ArrayList<String> getBallotList()
    - public int getNumSeats()

IRVoting Class:

- Attributes:
    - private ArrayList<ArrayList<Candidate>> ballots
    - private ArrayList<Candidate> candidates
- Methods:
    - public IRVoting(String, ArrayList<String>, int)
    - public void calculateBallots()
    - private ArrayList<Candidate> processCandidateInfo(String)
    - private ArrayList<ArrayList<Candidate>>processBallotOrder(ArrayList<String>)
    - private void redistributeVotes()
    - private void performFirstRoundCalculations()
    - private void buildAndWriteRedistributedVotesToAuditFile(Candidate)
    - private void updateVotesForCandidate(ArrayList<ArrayList<Candidate>>)
    - private Candidate findCandidateForElimination()
    - private boolean isValidBallot(ArrayList<Candidate>)
    - private ArrayList<Arraylist<Candidate>> findEliminatedCandidateBallots(Candidate)
    - private boolean checkMajority()
    - public ArrayList<Candidate> getCandidates()
    - public ArrayList<ArrayList<Candidate>> getBallots()

OPLVoting Class:

- Attributes:
    - private candidatePartyInfo: String
    - private ballotList: ArrayList<String>
    - private tiedCandidates: ArrayList<Candidate>
    - private tiedParties: ArrayList<Party>
    - private remainingTies: ArrayList<Party>
    - private receivedRemainingSeats: ArrayList<Party>
    - private numSeats: int
    - private numBallots: int
    - private parties: ArrayList<Party>
    - private winningParty: Party
    - private partyWithAllVotes: Party
    - private quota: int
- Methods:
    - public OPLVoting(String, ArrayList<String>, int, int, String)
    - private ArrayList<Party> processPartyInfo(String)
    - private void processBallots(ArrayList<String>)
    - public void performSeatAllocations()
    - private void determineRemainingSeatAllocation()
    - private boolean checkRemainderTie()
    - private void handleRemainderTie()
    - private void findPartyWithMostSeats()
    - private void findPopularCandidates()
    - private boolean checkPartyWithAllVotes()
    - private boolean checkPartyTie()
    - private void handlePartyTie()
    - private void buildAndWriteRoundResultsToAuditFile()
    - private void buildAndWritePartyResultToAuditFile()
    - private void buildAndWriteCandidateResultToAuditFile()
    - public Party getWinningParty()
    - public ArrayList<Party> getParties()

Party Class:

- Attributes:
    - private String partyName
    - private ArrayList<Candidate> candidates
    - private int totalPartyVotes
    - private int initialPartyVotes
    - private int numSeatsAllocated
- Methods:
    - public Party(String)
    - public String getPartyName()

    - public ArrayList<Candidate> getCandidates()
    - public int getTotalPartyVotes()
    - public void setNumPartyVote(int)
    - public int getInitialPartyVotes()
    - public void incrementPartyVote()
    - public void setInitialPartyVotes(int)
    - public void addCandidate(Candidate)
    - public int getNumSeatsAllocated()
    - public void incrementNumSeatsAllocated(int)
    - public boolean equals(Object)

VoteEasy Class:

- Methods:
    - public static void main(Strig[] args)
    - private void buildCLI()
    - private String receiveAndValidateUserInput)
    - private void displayIRWinner(winningCandidate, ArrayList<Candidates>)
    - private void displayOPLWinner(winningCandidate, winningParty, ArrayList<Party)

Voting Class:

- Attributes:
    - protected int numBallots
    - protected Candidate winningCandidate
    - protected String auditFileName
- Methods:
    - protected void writeToAuditFile(StringBuilder)
    - protected void buildAndWriteTieResultToAuditFile()
    - protected void buildAndWriteInitialResultsToAuditFile()
    - protected void handleTie()
    - protected boolean checkTie()
    - protected Candidate getWinningCandidate()

5. Component Design

Class VoteEasy houses the following functions:

- **main(String[] args) Implementation:**

  function main(String[] args):
  buildCLI()

Initialize fileName variable to be the return value of receiveAndValidateUserPrompt()

Initialize fileParser variable to an instantiation of the FileParser class and pass in fileName

Retrieve file header using the fileParser object
Retrieve candidate and party information string using the fileParser object
Retrieve ballot information using the fileParser object
Retrieve number of ballots information using the fileParser object

if file header is IR:
    Instantiate instant runoff class object and pass in the candidate and party information string, ballot information and number of ballots

    Call calculateBallots() function using the instant runoff class object

    Initialize local variable winningCandidate with the return value of getWinningCandidate() called using the IR class object

    Initialize local variable candidates with the return value of getCandidates() called using the IR class object

    displayIRWinner(winningCandidate, candidates)
else:
    Retrieve number of seats information using the fileParser object

    Instantiate OPL class object and pass in the candidate and party information string, ballot information, number of seats and number of ballots

    Call performSeatAllocations() function using the OPL class object

    Initialize local variable parties with the return value of getParties() called using the OPL class object

    Initialize local variable winningCandidate with the return value of getWinningCandidate() called using the OPL class object

    Initialize local variable winningParty with the return value of getWinningParty() called using the OPL class object

    displayOPLWinner(winningCandidate, winningParty, parties)

- **buildCLI() Implementation:**

  function buildCLI():

  print “Welcome to the VoteEasy system”

  print “You have to simply type the name of the CSV file below that contains

  all the election information”

- **receiveAndValidateUserInput() Implementation:**

  function receiveAndValidateUserInput():
  while True:
  Prompt user for entering the name of the CSV file
  if file is a CSV file and exists in the current directory:
  return file name
  else if “help” is entered:
  print “Only a CSV file type is required. You should simply type the name of the CSV file that contains ballot information. Example: voting.csv”.
  else
  print “Incorrect filename entered! Please ensure it’s a CSV file and is present in the current directory.”

- **displayIRWinner(winningCandidate, ArrayList<Candidates>) Implementation:**

  function displayIRWinner(winningCandidate, ArrayList<Candidates):
  print “Instant Runoff Voting Election results”
  print winning candidate information such as name, party name and number / % of votes received

  Loop through all Candidate objects in list of candidates:
  print candidate information such as name, party name and number / % of votes received

- **displayOPLWinner(Candidate, Party, ArrayList<Party>) Implementation:**

  function displayOPLWinner(Candidate, Party, ArrayList<Party>):
  print “OPL Voting Election Results:”
  print winning party information such as party name, number / % of votes and seats received.

  print winning candidate information such as name, party name and number / % of votes received

Loop through all Party objects in list of parties:
print party information such as party name, number / % of votes and seats received

Class FileParser houses the following functions**:**

- function FileParser(filename):

  call parseFile function to parse CSV file and assign valid values to all member variables

- function parseFile(filename):

  create an empty list called “file_lines”

  open filename in read mode and read every line separated by a newline as an entry for file_lines

  fileHeader attribute is set to index 0 of file_lines

  numberOfCandidates attribute is set to index 1 of file_lines

  candidateLine attribute is set to index 2 of file_lines

  numberOfBallots attribute is set to index 3 of file_lines

  if fileHeader = “IR”

  ballotList attribute is array of every entry after index 4 of file_lines

  else

  numberOfSeats attribute is set to index 4

  ballotList attribute is array of every entry after index 5 of file_lines

- function getFileHeader():

  return the fileHeader attribute

- function getNumCandidates():

  return the numberOfCandidates attribute

- function getCandidateLine():

  return candiadateLine attribute

- function getNumBallots():

return the numberOfBallots attribute

- function getBallot List():

  return the ballotList attribute

- function getNumSeats():

  return the numberOfSeats attribute

Class Party houses the following functions:

- **Party() constructor Implementation**:

  Party():
  set party name attribute to party_name, set candidates to an empty array,
  set the total number of votes the party has to zero, set the total number of seats allocated to zero

- **getPartyName() Implementation:**

  function getPartyName():
  return the party name

- **getCandidates() Implementation:**

  function getCandidates():
  return the candidates in the candidate array.

- **getTotalPartyVotes() Implementation:**

  function getTotalPartyVotes():
  return the total number of party votes

- **incrementPartyVote() Implementation:**

  function incrementPartyVote():
  increment the total number of party votes by 1

- **addCandidate(Candidate) Implementation:**

  function addCandidate(candidate):
  Add the candidate to the party.

- **getNumSeatsAllocated() Implementation:**

  function getNumSeatsAllocated():

return the allocated number of seats associated with the party.

- **incrementNumSeatsAllocated() Implementation:**

  function incrementNumSeatsAllocated(num_seats):
  Add the number of seats defined by num_seats to the allocated number of seats associated with the party.

- **getInitialPartyVotes() Implementation:**

  function getInitialPartyVotes(self):
  return initial number of party votes before any redistribution occurred

- **setInitialPartyVotes() Implementation:**

  function setInitialPartyVotes(int):
  Set initial number of party votes to the current total number of party votes

- **equals(Object) Implementation:**

  function equals(Object):
  Check if the Object calling this function and the passed in Object parameter are equal

Class Candidate has houses the following functions:

- **Candidate() constructor Implementation:**

  Candidate(String, String):
  Set name and party attributes to the string parameters and num votes to 0.

- **getName() Implementation:**

  function getName():
  return candidate name

- **getParty() Implementation**:

  function getParty():
  return candidate party name

- **getNumVotes() Implementation:**

  function getNumVotes():

return number of votes candidate has

- **incrementVote() Implementation:**

  function incrementVote():
  increment candidate votes by 1

- **incrementRedistributedVote() Implementation:**

  function incrementRedistributedVotes(int):
  increments the number of redistributed votes a candidate has by 1

- **resetRedistributedVotes(int) Implementation:**

  function resetRedistributedVotes(int):
  This function sets the number of redistributed votes for a particular candidate back to 0 for each round of vote redistribution when running the IR voting protocol.

- **getRedistributedVotes() Implementation:**

  function getRedistributedVotes():
  returns the number of redistributed votes

- **setElimination(boolean) Implementation:**

  function setElimination(boolean):
  Set candidate elimination status to passed in boolean value

- **isEliminated() Implementation:**

  function isEliminated():
  returns elimination status of the current candidate

Abstract Class Voting: No function definitions are provided here.

Class IRVoting defines the following functions:

- **IRVoting(String, ArrayList<String>, int numBallots)**

  function IRVoting(String candidateLine, ArrayList<String> ballotList, int numBallots):

  Initialize member variables with passed in parameters.

  Initialize audit file name to “IR_Audit_File.txt”

- **processCandidateInfo(String) Implementation:**

function processCandidateInfo(String candidateLine):

Create an empty list called “candidates”

Split the candidateLine parameter using “, “ as the delimiter and store it in a local variable called candidatePartyInfo

Loop through each element in candidatePartyInfo:

Split the “element” into two local variables “candidate” and “party” using space (“ ”) as the delimiter.

Create a new Candidate object using the “candidate” local variable above and pass in the “party” local variable as a parameter to the Candidate constructor

Add newly created Candidate object to the “candidates” list

return candidates

- **processBallotOrder(ArrayList<String>) Implementation:**

function processBallotOrder(ArrayList <String> ballotLines):

Create an empty list called “ballots”

Loop through each ballotLine in ballotList:

Split ballotLine using “,” as the delimiter and store the value in a local variable called currentBallot

Create an empty list called ballotOrder that has the same length as the length of the list of candidates

Loop through each vote in currentBallots where voteIdx is the index:

Initialize variable candidatePreference which is equal to currentBallot[voteIdx]

if currentPreference is not an empty string or missing:

Initialize local variable called candidateIdx that is set to int(candidatePreference) - 1

Set ballotOrder[candidateIdx] to candidate object at voteIdx in the candidate list

Add ballotOrder to the ballots list

return ballots

- **performFirstRoundCalculations() Implementation:**

function performFirstRoundCalculations():
Loop through each ballot in ballots:
If a candidate object at the first index in ballots exists:
Increment vote for current candidate object

- **checkMajority() Implementation:**

function checkMajority():

Loop through each candidate object in the candidates list:

Initialize local variable fractionOfVotes that is equal to the number of votes a candidate received divided by the total number of ballots.

if fractionOfVotes greater than 0.5:
Set class variable "winningCandidate" to the candidate object that won the election

return True
return False

- **checkTie() Implementation:**

function checkTie():
Use Collections.max() to find the Candidate with max number of votes and store that value in maxVotesCandidate
Initialize numVotes variable to the max number of votes using the maxVotesCandidate object

Loop through each candidate object in the candidates list:
if candidate has not been eliminated and candidate's number of votes is not equal to numVotes and not equal to 0:
return False
return True

- **handleTie() Implementation:**

function handleTie():

Find all non-eliminated candidates and store it in a list variable called activeCandidates

Initialize local variable called secureRandom to a new SecureRandom instance
Initialize local variable randomIdx to a random index in the active candidates list using the secureRandom object
Set winningCandidate member variable to the active candidate object at randomIdx

- **findCandidateForElimination() Implementation:**

function findCandidateForElimination():
    Initialize variable elimCandidate initially set to null
    Initialize variable lowestNumVotes to the max int value in Java
    Loop through each Candidate object in candidates list:
        if current Candidate object has lower votes than lowestVoteCandidate and hasn't been eliminated:
            if current candidate object has 0 votes:
                Set candidate's elimination status to True
                Write eliminated candidate information to audit file
            else
                Set elimCandidate to current Candidate object
                Set lowestNumVotes to current Candidate object's votes
    return elimCandidate

- **isValidBallot(ArrayList<String>) Implementation:**

function isValidBallot(ArrayList<String>ballot):
    Loop through each Candidate object in ballot:
        if Candidate object exists and if Candidate hasn't been eliminated yet:
            return True
    return False

- **findEliminatedCandidateBallots(Candidate) Implementation:**

function findEliminatedCandidateBallots(Candidate eliminationCandidate):
    Initialize variable relevantBallots to an empty list
    Initialize variable ballotsToRemove to an empty list
    Loop through each ballot in ballots:
        if first choice Candidate object in ballot equals eliminationCandidate
            if ballot is valid:
                Add current ballot to relevantBallots
            else:

Add current ballot to ballotsToRemove
Remove all invalid ballots from the “ballots” member variable
return relevantBallots

- **updateVotesForCandidate(ArrayList<Arraylist<Candidate>>) Implementation:**

function updateVotesForCandidate(ArrayList<Arraylist<Candidate>>eliminatedCandidateBallots):
    Loop through each ballot in eliminatedCandidateBallots:
        Initialize variable candidateIdx to 0
        Loop through each Candidate object in ballot:
            if Candidate object exists and has not been eliminated yet:
                Increment Candidate object’s votes
                Increment Candidate object’s redistributed votes variable
                Set candidateIdx to the current Candidate object’s index
                break out of current loop

        Initialize variable reassignedBallot to be a new list containing all Candidate objects starting from the current candidateIdx value to the end of the ballot list variable. Fill in the rest of reassignedBallot with “null” to match the length of “ballot”

        Remove current ballot from ballots list
        Add reassignedBallot to the ballots list

- **writeToAuditFile(StringBuilder) Implementation:**

try:
    Initialize variable filepath with the path to the audit file

    Initialize new FileWriter object called fileWriter
    Initialize new BufferedWriter object called bufferWriter and pass in the fileWriter object

    Write stringbuilder content to the file

    Close BufferedWriter and FileWriter
catch:
    print Exception message
    Exit program

- **buildAndWriteInitialResultsToAuditFile() Implementation:**

function buildAndWriteInitialResultsToAuditFile():
Initialize Stringbuilder object called sb
Append text “Voting Protocol Name: Instant Runoff (IR)” to sb
Append the text “Candidate & Party \t Number of Votes” to sb

Loop through each candidate:
Append current candidate’s name, party, number of votes and % of votes to the StringBuilder object
Append winner Candidate information if winner has been found
writeToAuditFile(sb)

- **buildAndWriteTieResultToAuditFile() Implementation:**

function buildAndWriteTieResultToAuditFile():
Initialize StringBuilder object called sb

Append text “Following candidates are currently tied: ” to sb

Loop through each Candidate object in the list of Candidates:
Append candidate name to sb

Append text “The winner of the tie result is: ” followed by the name of winning Candidate to sb

writeToAuditFile(sb)

- **buildAndWriteRedistributedVotesToAuditFile(Candidate) Implementation:**

function buildAndWriteRedistributedVotesToAuditFile(Candidate eliminatedCandidate):
Initialize StringBuilder object called sb

Append text “Name of the candidate eliminated during this round: ” followed by the name of the eliminated candidate to sb
Append the text “Candidate & Party \t Number of Votes” to sb

Loop through each candidate
Append current candidate’s name, party, number of redistributed votes, total number of votes, and % of overall votes to the StringBuilder object

writeToAuditFile(sb)

- **calculateBallots() Implementation:**

function calulcateBallots():

```
performFirstRoundCalculations()
if majority achieved after first round calculations:
        buildAndWriteInitialResultsToAuditFile()
else if tie achieved after first round calculations:
        handleTie()
        buildAndWriteTieResultToAuditFile()
else:
        redistributeVotes()
```

- **redistributeVotes() Implementation:**

```
function redistributeVotes():
        while True:
                if checkTie():
                        handleTie()
                        buildAndWriteTieResultToAuditFile()
                        return

                Initialize variable eliminatedCandidate to the return value of
                findCandidateForElimination()

                Set the eliminatedCandidate object's elimination status to True

                Initialize variable eliminatedCandidateBallots to return value of function
                findEliminatedCandidateBallots()

                if length of eliminatedCandidateBallots is not equal to 0:
                        Make function call to function updateVotesForCandidates() to
                        redistribute votes to different Candidates
                        buildAndWriteRedistributedVotesToAuditFile
                        (eliminatedCandidate)

                        Loop through each Candidate object in list of candidates:
                                if candidate hasn't been eliminated:
                                        Reset each Candidate object's redistributed vote
                                         count to 0
                else:
                        buildAndWriteRedistributedVotesToAuditFile
                        (eliminatedCandidate)

                if majority achieved after redistribution:
                        buildAndWriteInitialResultsToAuditFile()
                        return
```

- **getWinningCandidate() Implementation:**

  function getWinningCandidate():
  return Winning Candidate object

- **getCandidates() Implementation:**

  function getCandidates():
  return list of candidates

- **getBallots() Implementation:**

  function getBallots():
  return list of ballots

Class OPLVoting implements the following functions:

- **OPLVoting(String, ArrayList<String>, int, int):**

  function OPLVoting(String candidateLine, ArrayList<String> ballotLine, int numSeats, int numBallots):
  Initalize member variable names to passed in parameters

  Initialize audit file name to "OPL_Audit_File.txt"

- **processPartyInfo(String) Implementation:**

  Note: candidatePartyInfo is a private member variable that was initialized by splitting the line containing candidate and party information in the input CSV file using the ", " delimiter.

  function processPartyInfo(String candidateLine):
  Initialize local variable "parties" as an empty list
  Initialized local set called addedParties

  Loop through each element in member variable candidatePartyInfo:
  Split the "element" into two local variables "candidate" and "party" using space (" ") as the delimiter.

  if party string is not part of the addedParties set:
  Create Party object and pass in the "party" string
  Make new Candidate object and pass in the "candidate" string
  Add candidate to list of candidates associated with the Party object
  Add newly created Party object to the "parties" list
  Add "party" string to the addedParties set
  else:

Loop through each Party object in parties:
if current Party object's name equals the "party" string:
Make new Candidate object and pass in the "candidate" string
Add candidate to list of candidates associated with the current Party object

- **processBallots(ArrayList<String>) Implementation:**

Note: candidatePartyInfo is a private member variable that was initialized by splitting the line containing candidate and party information in the input CSV file using the ", " delimiter.

function processBallots(ArrayList<String> ballotList):
Loop through each ballotLine in ballotList:
Split ballotLine using "," as the delimiter and store the value in a local variable called ballotInfo

InitializeVariable called indexToUpdate to the index where "1" occurs in the ballotInfo list

Split the "element" at indexToUpdate in the candidatePartyInfo member variable into two local variables "candidate" and "party" using space (" ") as the delimiter.

Loop through each Party object in the list of parties:
if the current Party object's party name is the same as the local variable "party" defined earlier:
Loop through each Candidate object that belongs to the current Party object's list of candidates:
if the current Candidate object's name is equal to the "candidate" string defined earlier:
Increment Candidate object's votes
break out of loop
Increment current Party object's total number of votes
Set Party object's initial number of votes to its current total number of votes
break out of loop

- **determineRemainingSeatAllocation() Implementation:**

function determineRemainingSeatAllocation():
if a tie currently exists:
handleRemainderTie()
buildAndWriteTieResultToAuditFile()

else:
Get first party object of "remainingVotes" member variable and increment its seats
Add party object to "receivedRemainingSeats"

- **checkRemainderTie() Implementation:**

function checkRemainderTie():
Initialize member variable "remainingVoteTies" to an empty list of party objects
Initialize local variable maxRemainder to -1

Loop through each Party object in the list of parties:
if receivedRemainingSeats does not contain the party:
Set maxRemainder to party's remaining votes
break

Loop through each Party object in the list of parties:
if Party object's vote is greater than maxRemainder:
Set maxRemainder to party's remaining votes

Loop through each Party object in the list of parties:
if Party object's vote equal to maxRemainder and is not in receivedRemainingSeats:
Add party to remainingVoteTies

if length of remainingVoteTies is greater than 1:
return True
else:
return False

- **handleRemainderTie() Implementation:**

function handleRemainderTie():

Initialize local variable called secureRandom to a new SecureRandom instance

Initialize local variable idx to a random index in the remainingVoteTies list using the secureRandom object
Increment party at remainingVoteTies idx
Add party at remainingVoteTies idx to receivedRemainingSeats

- **checkTie() Implementation:**

function checkTie():
Initialize local variable maxVotes variable to be the max number of votes a candidate received from the list of candidate objects belonging to the

winningParty variable

Loop through each Candidate object in the list of candidates belonging to the winningParty object:
    if current Candidate object has the same number of votes as maxVotes:
        Add Candidate object to list of tied candidates stored in the tiedCandidates private member variable

if length of tiedCandidates is greater than 1:
    return True
else:
    return False

- **handleTie():**

function handleTie():

Initialize local variable called secureRandom to a new SecureRandom instance
Initialize local variable idx to a random index in the tiedCandidates list using the secureRandom object
Set winningCandidate member variable to the candidate object at idx

- **findPartyWithMostSeats() Implementation:**

function findPartyWithMostSeats():
    if a tie currently exists:
        handlePartyTie()
        buildAndWriteTieResultToAuditFile()
    else:
        Set “winningParty” member variable to first Party object in the list of tiedParties

- **findPopularCandidates() Implementation:**

function findPopularCandidates():
    if a tie currently exists:
        handleTie()
        buildAndWriteTieResultToAuditFile()
    else:
        Set “winningCandidate” member variable to first Candidate object in the list of tiedCandidates

- **checkPartyWithAllVotes() Implementation:**

function checkPartyWithAllVotes():

Initialize local arraylist of Party objects called zeroVoteParties

Loop through each Party object in the list of parties:
    if Party object's votes is equal to 0:
        Add party object to zeroVoteParties

if length of zeroVoteParties length is equal to parties length - 1:
    return True
else:
    return False

- **checkPartyTie() Implementation:**

function checkPartyTie():
    Initialize local variable maxSeats variable to be the max number of seats a party received from the list of parties

    Loop through each Party object in the list of parties:
        if current Party object has the same number of votes as maxVotes:
            Add Party object to list of tied parties stored in the tiedParties private member variable

    if length of tiedParties is greater than 1:
        return True
    else:
        return False

- **handlePartyTie() Implementation:**

function handlePartyTie():
    Initialize local variable called secureRandom to a new SecureRandom instance
    Initialize local variable idx to a random index in the tiedCandidates list using the secureRandom object
    Set winningParty member variable to the candidate object at idx

- **performSeatAllocations() Implementation:**

Note: "quota" is a private member variable that's initialized with the number of ballots divided by the number of seats

function performSeatAllocations():
    processBallots(ballotList)
    buildAndWriteInitialResultsToAuditFile()

Initialize local variables round to 1 and seatsAvailable to private variable numSeats

While there are seats left to allocate:
Initialize variable totalAllocationsPerRound to 0

if round is 1:
if there is a party with all the votes:
Loop through each Party object in parties:
if Party object's vote is not 0:
Set party object to partyWithAllVotes
break

Give all seats to partyWithAllVotes
Set partyWithAllVotes votes to 0
Increment totalAllocationsPerRound to seatsAvailable
else:
Loop through each Party object in the list of parties:
Initialize variable seatsAllocated to the total number of votes the party received divided by the quota

Increment the number of seats allocated for the current Party object by seatsAllocated
Initialize variable remainingPartyVotes to the total number of votes the party has currently modulus the quota
Decrement the total number of votes the party has for the next round by remainingPartyVotes
Increment totalAllocationsPerRound by the seatsAllocated variable

Increment local round variable
else:
determineRemainingSeatAllocation();
Increment totalAllocationsPerRound

Decrement seatsAvailable by totalAllocationsPerRound
buildAndWriteRoundResultToAuditFile()

findPartyWithMostSeats()
findPopularCandidate()
buildAndWritePartyResultToAuditFile()
buildAndWriteCandidateResultToAuditFile()

- **writeToAuditFile(StringBuilder) Implementation:**

  function writeToAuditFile(StringBuilder):

  try:

  Initialize variable filepath with the path to the audit file

  Initialize new FileWriter object called fileWriter
  Initialize new BufferedWriter object called bufferWriter and pass in the fileWriter object

  Write stringbuilder content to the file

  Close BufferedWriter and FileWriter

  catch:

  print Exception message
  Exit program

- **buildAndWriteInitialResultsToAuditFile() Implementation:**

  function buildAndWriteInitialResultsToAuditFile():
  Initialize new StringBuilder object called sb
  Append text “Voting Protocol Name: Open Party List (OPL)” to sb
  Append text “Party \t Total Votes \t Seats Allocated” to sb

  Loop through each Party object in list of parties:
  Append Party name and total number of votes to sb

  writeToAuditFile(sb)

- **buildAndWriteRoundResultToAuditFile() Implementation:**

  function buildAndWriteRoundResultToAuditFile():
  Initialize new StringBuilder object called sb
  Append text “Party \t Remainder Votes \t Total Seats Allocated” to sb

  Loop through each Party object in list of parties:
  Append Party name, remainder votes and number of seats to sb

  writeToAuditFile(sb)

- **buildAndWritePartyResultToAuditFile() Implementation:**

function buildAndWriteResultToAuditFile():
Initialize new StringBuilder object called sb
Append text “Party \t Total Votes \t Total Seats Allocated \t % of Votes to % of Seats” to sb

Loop through each Party object in list of parties:
Append Party name, remainder votes and number of seats to sb

Append winning Party name

writeToAuditFile(sb)

- **buildAndWriteTieResultToAuditFile() Implementation:**

function buildAndWriteTieResultToAuditFile():
Initialize new StringBuilder object called sb

if it is a candidate tie:
Append text “Candidates currently tied: ” to sb

Loop through each currently tied candidate:
Append candidate’s name to sb

Append text “Winning candidate is ” followed by the name of the winning candidate
if it is a party tie:
Append text “Parties currently tied: ” to sb

Loop through each currently tied parties:
Append party’s name to sb

Append text “Winning party is ” followed by the name of the winning party
if it is remaining vote tie:
Append text “Parties currently tied: ” to sb

Loop through each currently tied parties with the same most remaining votes:
Append party’s name to sb

Initialize partyReceivedSeats to null
Loop through each party in receivedRemainingSeats:
Loop through each party in list of parties:

if party in receivedRemainingSeats is party in list of parties:
set partyReceivedSeats to current party in receivedRemainingSeats
break

if partyReceivedSeats is not null:
break

Append text “Winning party of remaining votes tie is ” followed by the name of the winning party

writeToAuditFile(sb)

- **buildAndWriteCandidateResultToAuditFile() Implementation:**

function buildAndWriteCandidateResultToAuditFile():
Initialize new StringBuilder object called sb
Append the text “Candidate & Party \t Number of Votes \t % of votes” to sb

Loop through winning Party’s list of candidates:
Append Candidate’s name, total number of votes and % of votes to sb

writeToAuditFile(sb)

- **getWinningParty() Implementation:**

function getWinningParty():
return winning Party object

- **getParties() Implementation:**

function getParties():
return list of parties

6. HUMAN INTERFACE DESIGN

6.1 Overview of User Interface

Once the system is running, the command line interface will prompt the user to input a file. When the correct file has been entered, the system will determine the winner of the election and display the winning Party and Candidate details on screen.

6.2 Screen Images

Sample interface for displaying the winner for an election whose ballots were counted using the IR Voting Protocol:

```
Welcome to the VoteEasy System!

Please enter name of the CSV file: votes.csv

Calculating ballots....

Instant Runoff Voting Election results:

Candidate and Party     Number of Votes         Percentage of votes won
Rosen (Dem.)                 60                         60%
Royce (Rep.)                 25                         25%
Chou (Ind.)                  10                         10%
Kleinberg (Libert.)           5                          5%

Winning Candidate is Rosen from the Democrat party who wins with 60 votes to their name.
```

Sample interface for displaying winning Party and winning Candidate's information for an election run using the OPL Voting protocol:

```
Welcome to the VoteEasy System!

Please enter name of the CSV file: votes.csv

Performing Seat Allocations....

Open Party List Voting Election results:

Parties         Number of Votes         Number of Seats         % of Votes to % of Seats
Democratic           60                      20                        60% / 66.7%
Republican           25                      15                        25% / 25%
Reform               10                      10                        10% / 16.67%
Green                 5                       5                         5% / 8.33%

The Democratic party has won the election with 60 votes and 20 seats.

The winning party's candidate list details are below:

Candidate and Party     Number of Votes         Percentage of votes won
Rosen (Dem.)                 60                         60%
Royce (Dem.)                 25                         25%
Chou (Dem.)                  10                         10%
Kleinberg (Dem.)              5                          5%

Winning Candidate is Rosen from the Democrat party who wins with 60 votes to their name.
```

Sample interface for user entering the wrong name of the file, or typing the help option, or when the CSV file is missing from the project folder:

```
Welcome to the VoteEasy System!

Please enter name of the CSV file: votes.txt
File name error: Only .csv files are allowed.

Please enter name of the CSV file: voting_file.csv
File missing from current folder. Ensure election file is in the same directory as this project.


Please enter name of the CSV file: help
Please enter the name of the CSV file that contains election information. Ensure file is in the same folder as this project.
```

6.3 Screen Objects and Actions

We are only displaying winner information on screen. The only interaction the user has with the interface is when they are prompted to enter the name of the CSV file containing candidate and ballot information.

7. REQUIREMENTS MATRIX

SRS Functional Requirement	SDD System Component
4.0 Command Line Interface	Class **VoteEasy** defines private functions for building the user interface and prompting user for the CSV file name.
4.2 Parse CSV File Header	Class **FileParser** parses the input CSV file and provides a getter function for returning the file header.
4.3 Parse Ballots and Candidate Information	Class **FileParser** parses the input CSV file and provides getter functions for returning ballot and candidate information.
4.4 Produce Audit File	Classes **OPLVoting and IRVoting** have private functions for producing and writing to the audit file.
4.5 Majority wins when using IR Voting Protocol	Class **IRVoting** houses a private function for checking if a majority occurs or not.
4.6 Popularity wins when using IR Voting Protocol	Class **IRVoting** houses private functions for redistributing votes in-case a majority or tie is not found in the first round of voting.
4.7 Implement Open Party List Voting Protocol	Class **OPLVoting** has been designed to perform seat allocations and declare the popular party and popular candidate within that party.

4.8 Breaking a Tie	Classes **OPLVoting and IRVoting** contain private functions for checking and resolving ties, as well as writing the tie result appropriately to the file
4.9 Display Winner and Election Information	Class **VoteEasy** houses two private functions for displaying winner information for both the IR and OPL Voting option.
4.10 Help Option	Class **VoteEasy** provides a private function for receiving user input and provides the necessary help instructions if “help” is entered or if the candidate enter’s the wrong file name or if the file name is not present in the current directory.

8. APPENDICES

N/A