Team 25
Voting System
Software Design Document

Name (s): Ashton Berg (ber00036), Caleb Tracy (tracy255), Elias Caceres (cacer019), and Garrett Abou-Zeid (abouz009)
Lab Section: 001

Date: (03/03/2023)

Table of Contents

1. Introduction

1.1 Purpose

This Software Design Document defines and describes the design for a voting system. The system's goal is to unbiasedly and consistently determine the winner of instant runoff votes and closed-party list elections. The following sections will describe and organize the architecture and systems design. This document is intended for developers implementing these systems, project reviewers, and testers.

1.2 Scope

The voting system should be able to intake ballots, analyze ballots and election information, and produce a winner for closed-party listing and instant runoff elections. This process will automate the tedious and delicate process of ballot counting, and any calculations needed to determine an election winner. Another benefit the system provides is security. The system prevents potential tampering during ballot processing, outputs an audit file detailing all computations, as well as ensures fair and unbiased results. The voting system does not ensure the information inputted into the system is valid or correct. The systems detailed in this design document will be written in Java. Users who wish to learn about system specifications should review the software requirements specification document.

1.3 Overview

This document is split into eight sections. The first section is an introduction providing must-know information about the system and this document. Section two contains a short system overview describing the general system and the context it is built upon. Section three covers three system models, a class diagram, a sequence diagram, and an activity diagram. The following subsections describe their design, provide a top-level description, and discuss the rationale for their choices. Section four describes how data is organized within the system and describes any objects or non-primitive data types the system utilizes. Section five covers lower-level design. Each subsection provides pseudo-code for a specific system component. Section six provides an overview of the limited user interaction with system interfaces. Section seven identifies where user cases are satisfied within this document. Section eight contains any appendixes.

1.4 Reference Material

Voting system’s software requirements specification document:

https://github.umn.edu/umn-csci-5801-01-S23/repo-Team25/blob/main/SRS/SRS_Team25.pdf

IEEE Software Design Descriptions best practices:
https://standards.ieee.org/ieee/1016/1478/

1.5 Definitions and Acronyms

CPL	Closed Party List
CSV	Comma Separated Value
IEEE	Institute of Electrical and Electronics Engineers
IRV	Instant Runoff Voting
Java	Popular object-oriented programming language.

2. System Overview

The purpose of the Voting System software is to facilitate and oversee elections by processing inputted ballots according to the designated election protocol. This software supports two election protocols, Instant Runoff Voting (IRV) and Open Party List (OPL) Voting, and is designed to function as a standalone system. The system can perform two types of voting algorithms: Instant Runoff Voting (plurality/majority) and Party List Voting (proportional voting).

For Instant Runoff Voting, voters rank candidates based on their preferences, and if a candidate receives over 50% of the #1 choice votes, they are declared elected. If there is no majority, the candidate with the fewest votes is eliminated, and their ballots are transferred to the candidate marked as their #2 choices. The votes are then recounted to see if a majority has been achieved. This process repeats until a candidate receives over 50% of the votes.

Party List Voting allows voters to mark their preference for a political party on the ballot, and seats are awarded to parties in proportion to their share of the vote. There are two list systems: closed list and open list. In a closed list system, the party votes for the entire party, and winning candidates are selected based on the order in which they appear on the ballot. In an open list system, voters can express their choice for particular candidates to represent their party, and the order of the candidate list depends on the number of votes won by each candidate.

After adhering to the protocol, the Voting System software declares the winners of the election and produces audit files to allow for regenerability of the election results.

Some features of the system include its ability to process votes based on the election type, reorder the votes in cases where there are no distinct winners, break ties with a fair coin toss in case there are candidates with the same number of votes, and lastly, an auditing mechanism that will create an audit file containing information about the election and the documentation of the election process for validation purposes.

3. System Architecture

3.1 Architectural Design

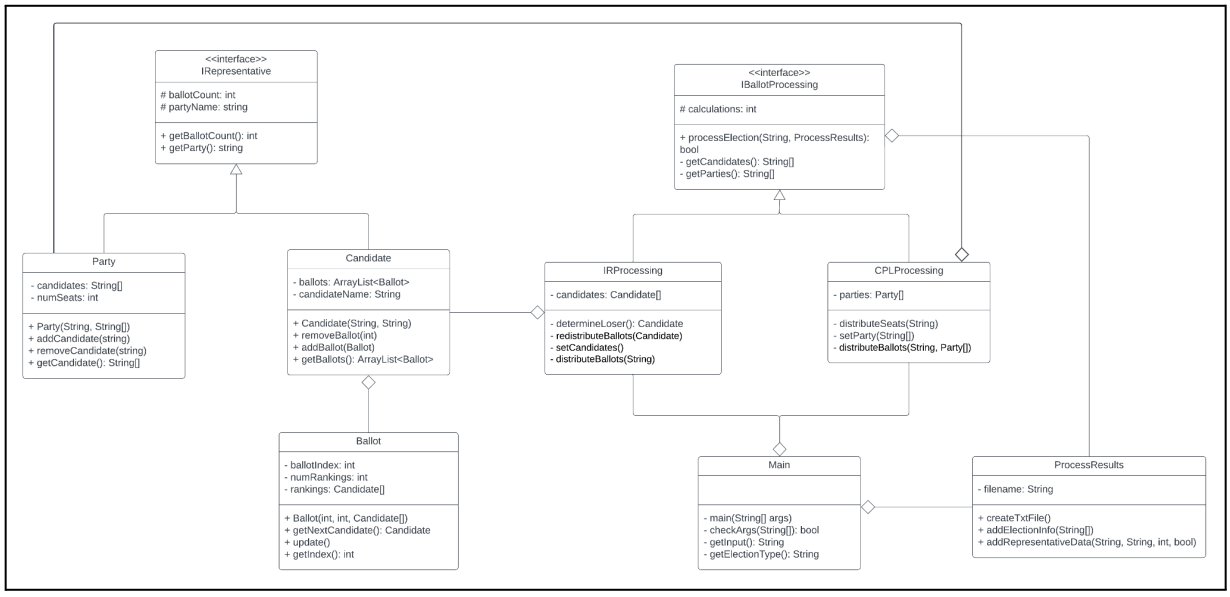

Figure 1: Class Diagram (figure included on Github repo)

At the core of the voting system are four subsystems with distinct functionality. These are representative objects, ballot processing, results processing, and main. The objects *Party* and *Candidate* implement the functionality inherited from the interface *IRrepresentative* on top of their own logic. Those objects are only used by their associated election processing class, as their purpose is to hold and access election data. *Candidate* objects also interact with *Ballot* objects which are used to swap ballots between candidates during instant runoff elections. The ballot processing subsystem holds the majority of data and executes the majority of internal system logic. Similarly to the representative objects, the *CPLProcessing* and *IRProcessing* classes implement an interface and their own election-specific logic. This subsystem creates and manages the representative objects performing election calculation, information gathering, and vote distribution to determine winners and losers.

Because the *main* and result processing classes were distinct in their design, each is considered to be its own subsystem. Main's primary purpose is to intake or request user input and execute the ballot processing subsystem. The *ProcessResults* class houses all the capabilities required to output information to an audit file.

The *main* subsystem interacts with both the results processing and election processing subsystems and is responsible for initiating both. Main passed on the functionality of the results processing subsystem to election processing which continues to interact with results processing. The representative objects subsystem only interacts with the elections processing subsystem, where processing classes create and use multiple instances of the representative objects.

3.2 Decomposition Description

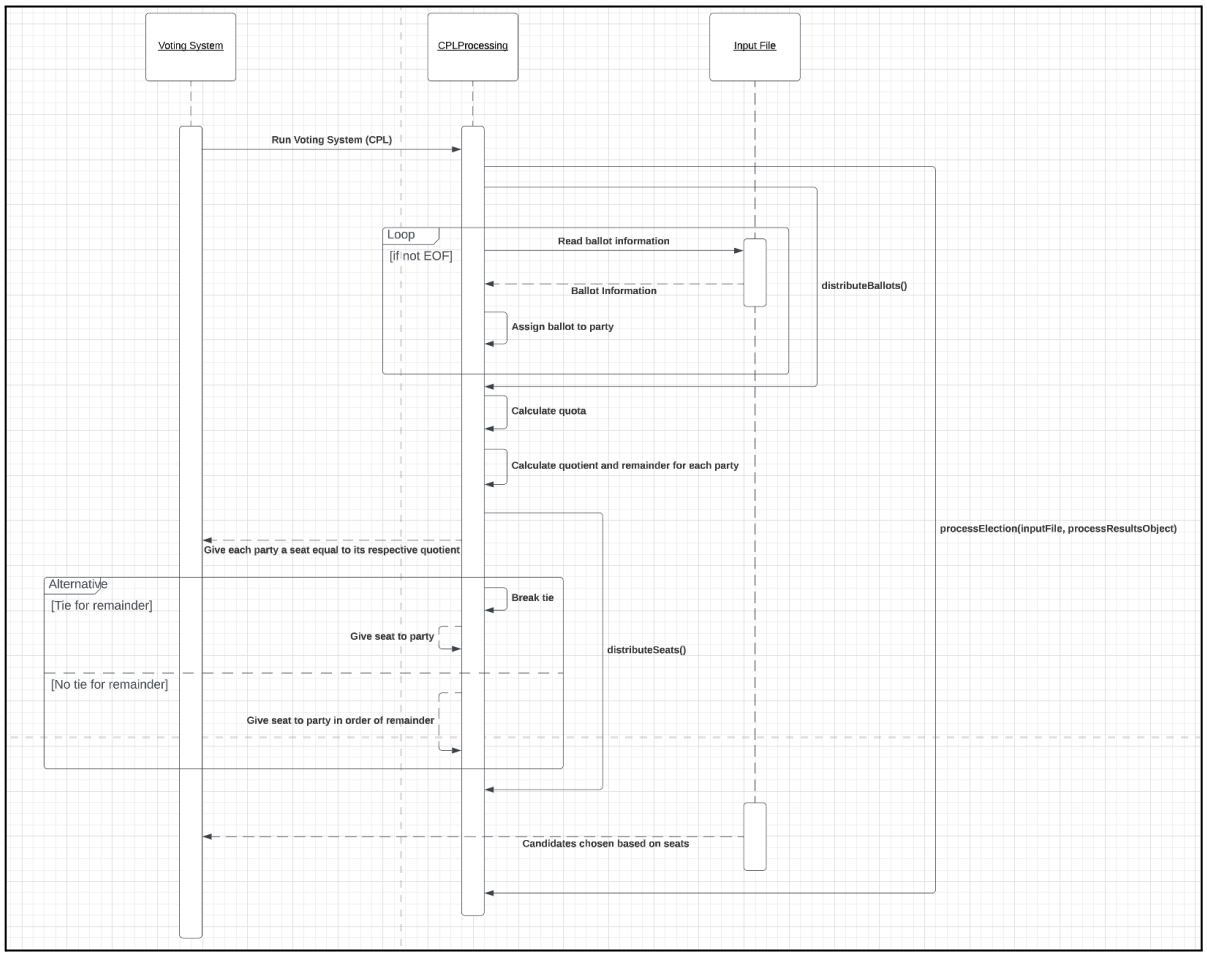

Figure 2: Sequence diagram for CPL election (figure included on Github repo)

This is a sequence diagram of the Closed Party Listing (CPL) voting system. The solid arrows are what methods or calls that are being made. The dotted arrows are what are being returned. It is assumed the file is already brought into the system and CPL is the voting system being run.

The voting system goes through each ballot in the input file and counts how many ballots belong to each party. The system then calculates a quota by dividing the number of ballots by the number of available seats. Then it calculates a quotient and remainder for each party by dividing the number of votes that the party received by the quota. The system gives each party the number of seats equal to its quotient. The remaining seats are distributed by order of the remainder for each party. Ties are determined by flipping a coin, the winner gets the seat. Seats are given to a particular party in order of the candidates listed on the ballot. The results are returned by the voting system.

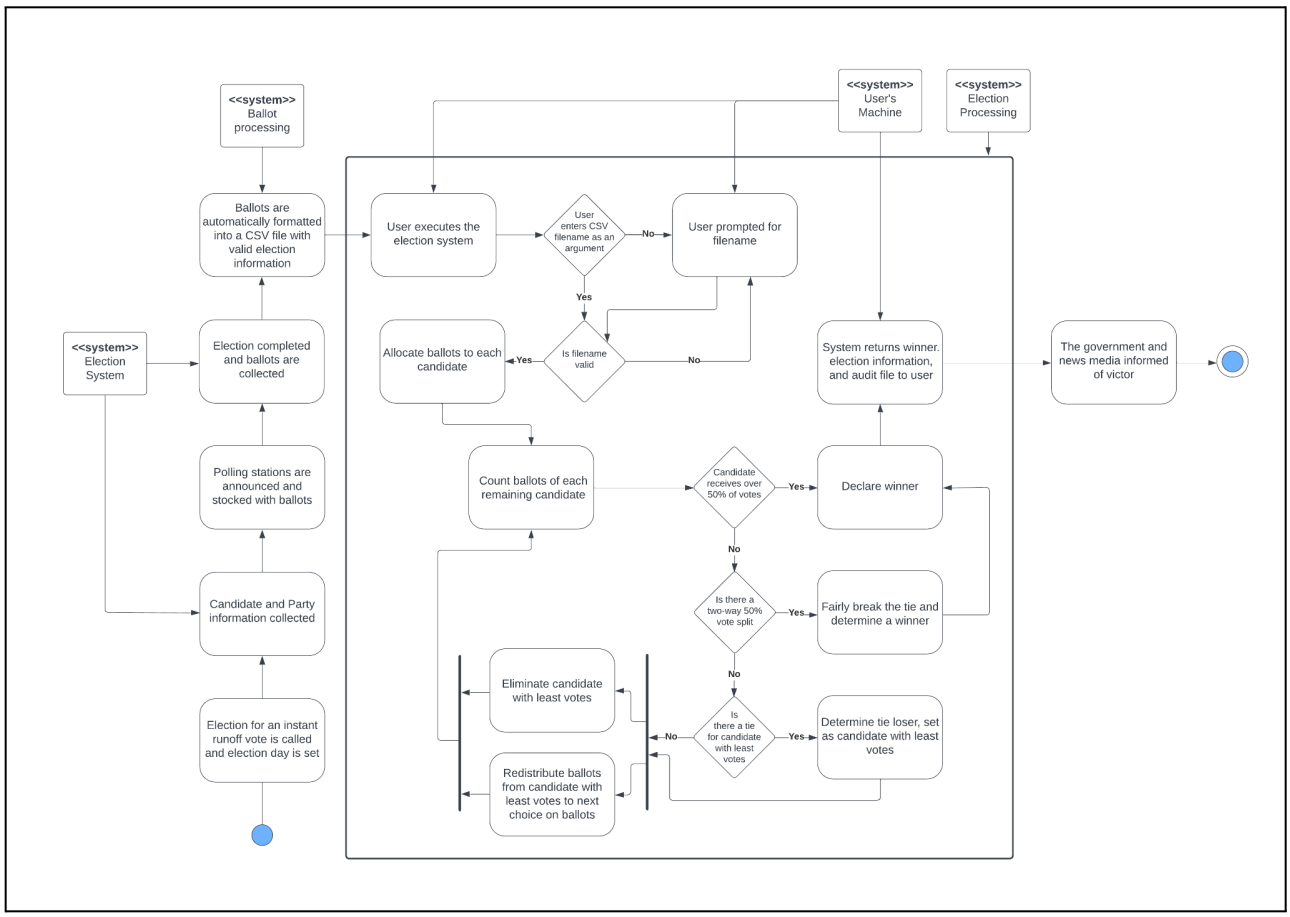

Figure 3: Activity diagram for instant runoff election (figure included on Github repo)

The activity diagram for an instant runoff election begins with several steps outside of the user's machine running the election processing system. It starts when an election for an instant runoff vote is called and the election day is set. From here, the actual election takes place, and an election system collects party and candidate information and ballots. A ballot processing system then formats the information collected by the election system into a CSV file. From here, our election processing system begins acting. The user executes the system on their machine, and provides the name of the CSV file as an argument. If the file is invalid or one was not provided, the system prompts the user for a file name. The system then allocates ballots to candidates, and counts the number of ballots each candidate received. If a candidate received over 50% of the vote, they are automatically declared as the winner in the declare winner action, and the system returns the winner, election information, and audit file to the user. The audit file can then be shared outside of the election processing system with media and government, and the activity diagram ends. If a candidate did not receive over 50% of the vote, the system checks if there are two candidates left who are both tied with 50% of the votes. If this is true, the system fairly determines a tie winner and proceeds to the declare winner action, where it proceeds executing the same steps described before. If this isn't true, the system checks if there is a tie for the candidate with the least votes. If there isn't, the system proceeds to a split where there are concurrent activities. If there is, the

system fairly chooses a tie loser as the least voted candidate, and then proceeds to the concurrent activities split. Here, the least voted candidate is eliminated, and the ballots of this candidate are redistributed to the remaining candidates. The split comes to an end, and goes back to the count and allocate ballots action. It will proceed until a candidate is declared as a winner and the steps following are completed.

3.3 Design Rationale

For the class diagram, multiple iterations were discarded due to unnecessary complications before the design was finalized. Candidate and Ballot objects were originally used for IR and CPL processing and it was decided a simpler solution was to utilize a separate Party object with an internal ballot count to avoid using Party, Candidate, and Ballot objects all within CPL processing. There was also an intermediate class between the main and election processing subsystems that was collapsed into the main subsystem for simplicity.

One crucial design choice was to make objects inherit from interfaces rather than abstract classes. Both multi-class subsystems had shared functionality that had to be distinctly implemented depending on the election type so interfaces were used. In the few cases functionality did overlap, it was decided the benefits of inherited abstract methods were not significant enough to use abstract classes. Another choice was to make ProcessResults its own class rather than a part of the election processing classes. This was in order to allow main to utilize ProcessResults' functionality and to isolate the editing of the audit file from the processing itself. Election processing still handles direct output to users.

The activity diagram had three major points of contention. The first is which systems interact where. It was decided only four systems were necessary, where the election and ballot processing is handled by a singular system rather than many for simplicity. The user's machine was seen as an important system in order to distinguish points of input and output. However, any other systems such as the system by which the media is informed of a winner were determined to be outside our scope. Coordination bars were used to indicate that candidate elimination simultaneously occurs alongside ballot redistribution. Depending on the implementation, these events may be sequential. However, they can be done simultaneously and both steps must be completed before the next step. Lastly, it was decided it is more efficient to check for a winner before checking for ties or losers because the system should produce a winner as soon as possible.

4. Data Design

4.1 Data Description

Information containing ballot information, including candidate and party information, will be stored in arrays. No major database systems will be used in the software. All data will be stored briefly in memory while the software runs and will no longer be stored after the software stops running. No data will be stored long-term.

4.2 Data Dictionary

Attribute Name	**Type**	**Description**
Ballot	Object	Contains information about a ballot.
ballotCount	Int	Number of ballots associated with a candidate (Instant runoff) or party (closed party listing).
ballotIndex	Int	The index of the ballot in the given input file.
ballots	ArrayList<Ballot>	An ArrayList of ballots that are associated with a particular candidate. ArrayLists are dynamically allocated arrays.
calculations	Int	The number of calculations performed to determine the winner of an election.
Candidate	Object	Contains information about a candidate.
candidateName	String	Name of candidate.
candidates (IRProcessing)	Candidates[]	A list of candidates on the ballots in an instant runoff voting system.
candidates (Party)	String[]	Name of candidates for a respective party for a closed

		party listing.
filename	String	File containing information about parties, candidates, ballots, and type of election.
numRankings	Int	Number of rankings for a party for closed party listing.
numSeats	Int	Number for storing number of seats a party has in CPL
parties	Party[]	A list of parties on the ballots in a closed party listing voting system.
Party	Object	Contains information about a party.
partyName	String	Name of party.
rankings	Candidate[]	Rankings for candidates for a closed party listing.

5. Component Design

5.1 Main

Function that first runs when the program starts. Takes in arguments that include the input file that contains information about the type of election, candidates, parties, and ballots.

- checkArgs(String[]): bool

  Function that checks to see if the arguments contain the correct information to be run by the voting system. If the arguments are correct, the program proceeds and runs the voting system. If they are not correct, the program terminates and returns an error message saying the arguments are incorrect.

- getInput(): String

  Returns the input that is given in the arguments in the main() function.

- getElectionType(): String

  Returns the election type that is given in the input file.

5.2 ProcessResults

- filename: String

  File name of the file that will be run by the voting system.

- createTxtFile()

  Creates a text file that will be outputted by the voting system that will contain election information and how the election progressed

- addElectionInfo(String[])

  Function that adds the election info, such as type of voting, number of candidates, candidates, number of ballots, etc, that will be written to the audit file that will be returned by the voting system.

- addRepresentativeData(String, String, int, bool)

  Function that will write information into the audit file, which includes the candidate name, party name associated with the candidate, number of ballots received by the candidate or party, and whether they lost or won the election.

5.3 IRProcessing

- candidates : Candidate[]

  An array of Candidate objects of candidates that are currently in the election. As candidates are eliminated, they will be removed from candidates. setCandidates() function, described below, initially populates candidates.

- determineLoser() : Candidate

  This function is used to find the candidate who received the least first choice votes so that they can be eliminated and have their votes redistributed. The outline of the function is as follows

  The function will store a local int min to keep track of the current lowest ballot count and a local Candidate array lowest_candidates containing the candidates with ballot count equal to min. The function will iterate through candidates:Candidate[], updating min and lowest_candidates when necessary. After all of candidates:Candidate[] has been iterated through, if lowest_candidates has one candidate element, that candidate is returned, else one of the candidates in lowest_candidates are chosen randomly and returned.

- redistributeBallots(Candidate)

  This function takes a candidate selected for elimination as a parameter and redistributes all of their ballots to the remaining candidates based on the next choice vote of each ballot. The outline of the function is as follows.

  The function will store a local candidate cur_candidate and a local ballot cur_ballot. The function will iterate through Candidate.ballots. At each iteration, cur_ballot will be set to the current ballot. Then, cur_candidate = cur_ballot.getNextCandidate(). Next, cur_candidate.addBallot(cur_ballot). Lastly, cur_ballot.ballotIndex++.

- setCandidates( )

  This function is used to create candidate objects to represent the candidates in the election. It uses the getCandidates():String[] and getParties():String[] functions to get the candidates and associated parties. The outline of the function is as follows.

  The function stores two local String arrays, cand_parties=getParties(), and cand_names=getCandidates(). It also stores a local Candidate cur_cand. It iterates through a for loop from 0 to length of cand_names, accessing cand_parties and cand_names at the current index i, and does cur_cand = Candidate(cand_names[i], cand_parties[i]). It adds cur_cand to candidates:Candidate[] at each iteration.

- distributeBallots(String)

  This function reads from the CSV file with the name given by the String parameter. As it reads through the CSV file, it will create Ballot objects and add the Ballots to their associated candidate in candidates. The outline of the function is as follows. It skips the first four lines of the file, since these lines don’t contain ballots, and reads the file line by line.

5.4 CPLProcessing

- parties: Party[]

  An array of Party objects of parties that are in the election.

- distributeSeats(String)

  This function assigns a number of seats to each party based on a calculated vote quota. It will determine the vote quota with the number of seats and number of votes casted, which will both be read from the CSV file with the name given by the String parameter. After determining the vote quota, it will calculate the quotient and remainder for each party in parties:Party[] to determine their number of seats, given by the quotient, and will allocate remaining seats based on the remainder, handling all ties in a fair manner. It will update the numSeats variable in each party of parties:Party[] to represent the number of seats that party received.

- setParty(String[])

  This function is used to create Party objects to represent the parties in the election. It uses the getParties():String[] function to get a list of the names of the parties in the election, and the getCandidates():String[] function to get a list of associated candidates for each party. It iterates through the list of parties and creates a Party object for each party, and stores each Party in parties:Party[]

- distributeBallots(String, Party[])

  This function takes a String parameter, which is the name of the CSV file containing election information, and a Party[], which is an array of the parties in the election. It will open the CSV file and iterate through each line to incrementing the ballotCount of each party for each vote they received.

5.5 Candidate

The *Candidate* object is stored in the Candidate class. It contains an integer *ballotCount* for the number of ballots the candidate currently has within ballots, a string *candidateName* that holds the name of that candidate, a string *partyName* for the name of the party the candidate belongs to, and ballots which is an array of Ballot objects which are the ballots currently assigned to that candidate object.

- Candidate(String, String)

  This constructor has two parameters, first being the candidate name and second being the party name. It is expected candidate objects are declared and initialized within *setCandidates()* in the *IRProcessing* class. The constructor sets the *partyName* and *candidateName* attributes according to the parameters, and sets *ballotCount* to zero.

- removeBallot(int)

  The purpose of *removeBallot* is to take an integer corresponding to a ballot's *ballotIndex* within the ballots ArrayList. Taking that index the function uses *remove()* to pop out that specific ballot from *ballots*, and updates the *ballotCount* by subtracting

one.

- addBallot(Ballot)

  This function adds a new *Ballot* object to the front of *ballots*. The function is given a *Ballot* object that is already created, and simply uses *add()* to make a new addition to the ArrayList.

- getBallots( ) : Ballot[]

  Returns the *ballots* attribute that is stored within the *Candidate* object.

5.6 Ballot

Each *Ballot* object has an integer *ballotIndex* indicating its ballot number which is entirely unique. It also has *numRankings* which indicates the cur number of candidates that are ranked and have not been removed. Lastly, *rankings* it a list of *Candi-+date* objects, being the corresponding candidates in order to the rankings that ballot listed.

- Ballot(int, int, Candidate[])

  The constructor takes in an integer which is set to *ballotIndex*, another integer set to *numRankings*, and an array of Candidate objects which is set to *rankings*.

- getNextCandidate( ) : Candidate

  This function returns the current *Candidate* object sitting at index zero within *rankings*. Used to determine whether or not to call *update()*.

- update( )

  Update will reduce *numRanking* by one, or do nothing if *numRankings* is zero. Otherwise, it removes the *Candidate* object at index zero of *rankings*. This action is recorded in the audit file using addRepresentativeData by the caller of *update()*.

- getIndex( ) : int

  This getter returns the Ballot object's ballotIndex, which is unique to that ballot. This may be used to record data to the audit file or to find and move a specific ballot such as in the case of breaking a tie.

5.7 Party

Each *Party* object has a string array of candidates within the object's party. It also has an integer variable called *numSeats* that stores the number of seats a party has in a CPL.

- Party(String, String[])

  This constructor takes in a String which sets the name of the Party object and its string array that stores candidates. This is used in CPL election processing.

- addCandidate(String)

  addCandidate will add candidates to the party object it is associated with. This will add the candidate into its respective party's *candidate* String array.

- removeCandidates(String)

  removeCandidates removes a candidate from a Party object's String candidate array. This could be used when candidates need to be removed after losing an election.

- getCandidate(); String[]

  This function returns a candidate from a Party object's candidate String array. This may be used when it comes to processing the election and assuring correct candidates get their allotted votes.

5.8 IRepresentative

The *IRepresentative* object has the individual voting information of a candidate. It stores an Integer *ballotCount* that holds the number of votes to the candidate in both IR or CPL elections. This object also has a *partyName* String attribute to it that holds that candidate's respective party.

- getBallotCount(): int

  This is a function that returns the number of ballots assigned to a candidate in the form of an integer.

- getParty(); String

  This returns the party of a candidate in the form of a String. This function might be used in determining who wins/loses seats in a CPL election.

5.9 IBallotProcessing

The *IBallotProcessing* object has the total number of calculations done in order to determine the winner of an election. This value is stored to *IBallotProcessing* in the form of an integer called *calculations.*

- processElection(String, ProcessResults);

  This is a function used to process the election given information from *IRProcessing* or *CPLProcessing*.

- getCandidates(): String[]

  This function returns an array of candidates in the form of a String array. This might be used in both elections when assigning election winners/losers.

- getParties(); String[]

  This function returns an array of the parties in the form of a String array. This might be used when seats need to be assigned in a CPL election.

6. Human Interface Design

6.1 Overview of User Interface

Before using the system, the user should already have access to an election file containing the election information and ballots. The user will interact with the system through command line arguments. In the command line, the user will execute the election system and provide the name of the election file. The system will take over from there, processing all of the election information to determine the outcome of the election. After the system has finished processing the election, the results will be displayed to the terminal and a text file containing information on how the election proceeded will be available within the same directory as the system. The user can access the text file, which will be named <election type><date>.txt, to view how the election proceeded and share it with media and government officials.

6.2 Screen Images

No images are available yet. The interface will look like a standard Linux terminal.

6.3 Screen Objects and Actions

There will be no screen objects or actions, all user interfaces will be done through the command line.

7. Requirements Matrix

Use Case	Satisfying System Components
UC_001	- Main (getInput())
UC_002	- Main (checkArgs())
UC_003	- Main (getElectionType()) - IBallotProcessing (getCandidates(), getParties())
UC_004	- IBallotProcessing - IRProcessing - IRepresentative - Candidate
UC_005	- IBallotProcessing - CPLProcessing - IRepresentative - Party
UC_006	- IBallotProcessing - IRProcessing
UC_007	- IRProcessing - CPLProcessing
UC_008	- ProcessResults
UC_009	- ProcessResults
UC_010	- ProcessResults
UC_011	- IBallotProcessing (processElection(String, ProcessResults))

8. Appendices

APPENDIX A: Class Diagram

https://github.umn.edu/umn-csci-5801-01-S23/repo-Team25/blob/main/SDD/ClassDiagram_Team25.pdf

APPENDIX B: Sequence Diagram

https://github.umn.edu/umn-csci-5801-01-S23/repo-Team25/blob/main/SDD/SequenceDiagram_Team25.pdf

APPENDIX C: Activity Diagram

https://github.umn.edu/umn-csci-5801-01-S23/repo-Team25/blob/main/SDD/ActivityDiagram_Team25.pdf