

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](http://www.cs.concordia.ca/~ormandj/comp354/2003/Project/ieeeSDD.pdf)¹ for the full IEEE Recommended Practice for Software Design Descriptions.

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

Team 7

Project 1: Waterfall Methodology
Software Design Document

Names:

Marcus Rana (rana0066)

Lucas Olsen (olse0280)

Liam McGuigan (mcgui479)

Justin Lau (lau00054)

Date: (03/02/2023)

TABLE OF CONTENTS

1. INTRODUCTION	2
1.1 Purpose	2
1.2 Scope	2
1.3 Overview	2
1.4 Reference Material	3
2. SYSTEM OVERVIEW	3
3. SYSTEM ARCHITECTURE	4
3.1 Architectural Design	4
3.2 Decomposition Description	5
3.3 Design Rationale	7
4. DATA DESIGN	7
4.1 Data Description	7
4.2 Data Dictionary	8
5. COMPONENT DESIGN	11
6. HUMAN INTERFACE DESIGN	16
6.1 Overview of User Interface	16
6.2 Screen Images	17
6.3 Screen Objects and Actions	20
7. REQUIREMENTS MATRIX	20
8. APPENDICES	22

1. INTRODUCTION

1.1 Purpose

This software design document describes the architecture and system design of the voting system. The intended audience of this document is election officials, audit officials, programmers, testers, and future developers.

1.2 Scope

This document will contain a complete description of the design of the voting system. The main objective of this project is to design a system that will efficiently count votes of an election in a method that is fair to all candidates and is able to be analyzed and verified by election officials.

The voting system will incorporate two different methods of voting, Instant Runoff and Closed Party List voting.

The basic architecture for this system will be written in C++ and will utilize CSV files for reading votes and other data.

1.3 Overview

This document will cover the following features in the following sections:

Section 2 will cover an overview of the system, consisting of the functionality, design, and context of the system.

Section 3 will cover the System Architecture, specifically the Architectural Design, Decomposition Description, and Design Rationale

Section 4 will cover the Data Design, which encompasses descriptions of the major entity types used and major data sources, along with an alphabetical list of all system entities and major data types.

Section 5 will cover the Component design of the project, which includes a closer look at every component in the project and a summary of various algorithms used.

Section 6 will describe the system's functionality from the user's perspective and show various screenshots of how the system's interface will look from the perspective of the user.

Section 7 will provide a cross reference that traces components of the system to the SRS of the voting system, and section 8 will list all appendices.

1.4 Definitions and Acronyms

Acronym	Definition
CPL	Closed Party List
IR	Instant Runoff
CSV	Comma Separated Values, type of file
SDD	Software Design Document

2. SYSTEM OVERVIEW

This SDD describes the implementation of an electronic voting system which can handle both Instant Runoff Elections and Closed Party Elections in order to calculate the winner of an election in a fair manner with verifiable results.

In the modern age, elections are becoming larger and more difficult to count results. Electronic voting is a method of counting votes in an efficient manner in order to count votes in increasingly larger elections.

This application will provide an interface to run an election via the command line. The electronic voting system will require all the ballots to be summarized in a CSV file, once the program is ran the system will generate an audit file which will give the users transparency and allow for the verification of results.

This application will be an executable file, which was compiled using C++. It is expected that it should be run on a Linux Machine (Ubuntu 20.04). It also requires a machine which is capable of supporting CSV files in order for the executable to read what the ballots are and produce results .

3. SYSTEM ARCHITECTURE

3.1 Architectural Design

The activity diagram for the instant runoff election is located below. It can also be referenced at *UMLActivityDiagramIR_Team7.pdf* for a higher-quality photo.

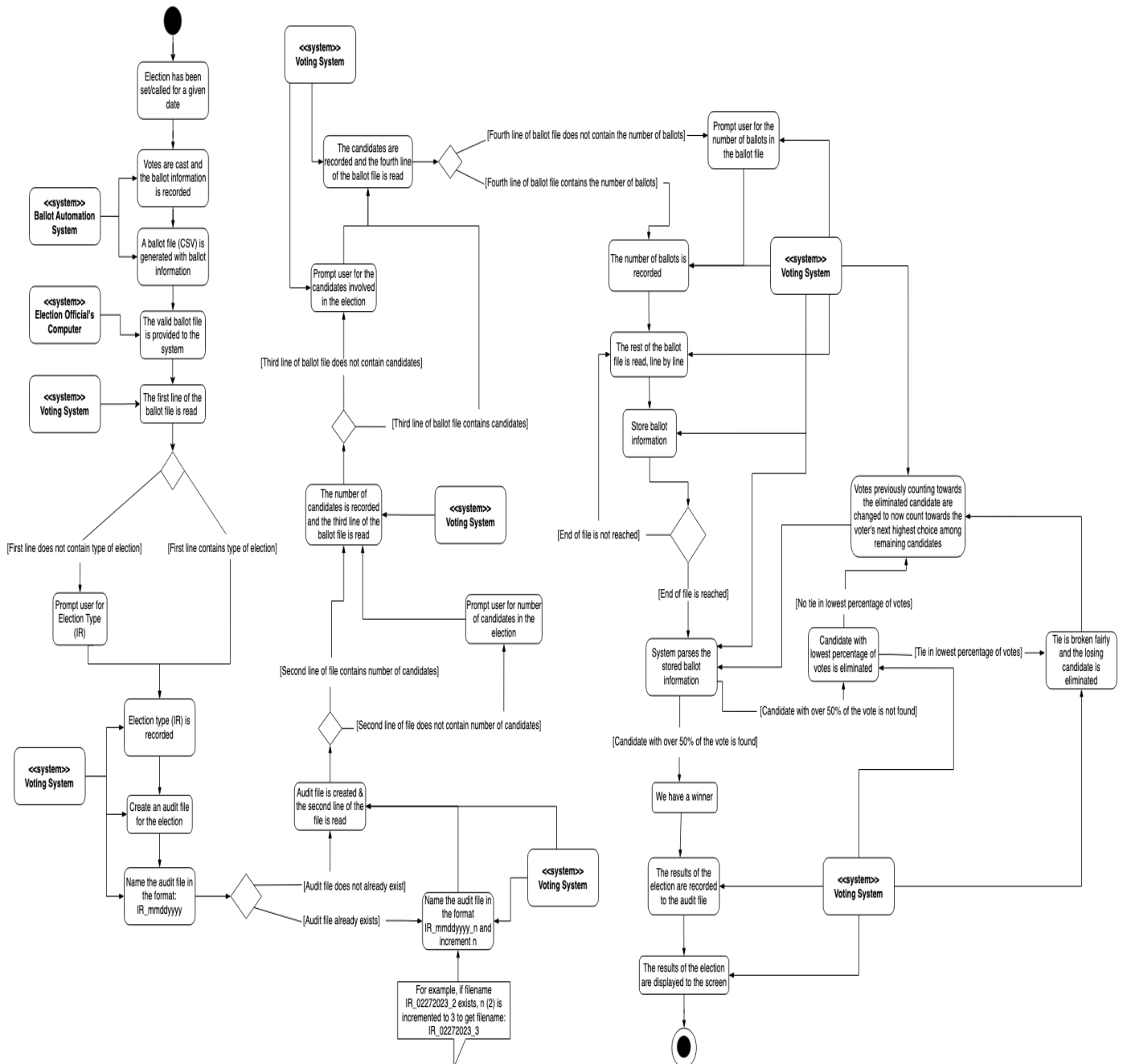


Figure 3.1.1: Activity Diagram for Instant Runoff Election

3.2 Decomposition Description

The UML class diagram for the entire voting system is shown below. It can also be referenced at *ClassDiagram_Team7.pdf* for a higher-quality photo.

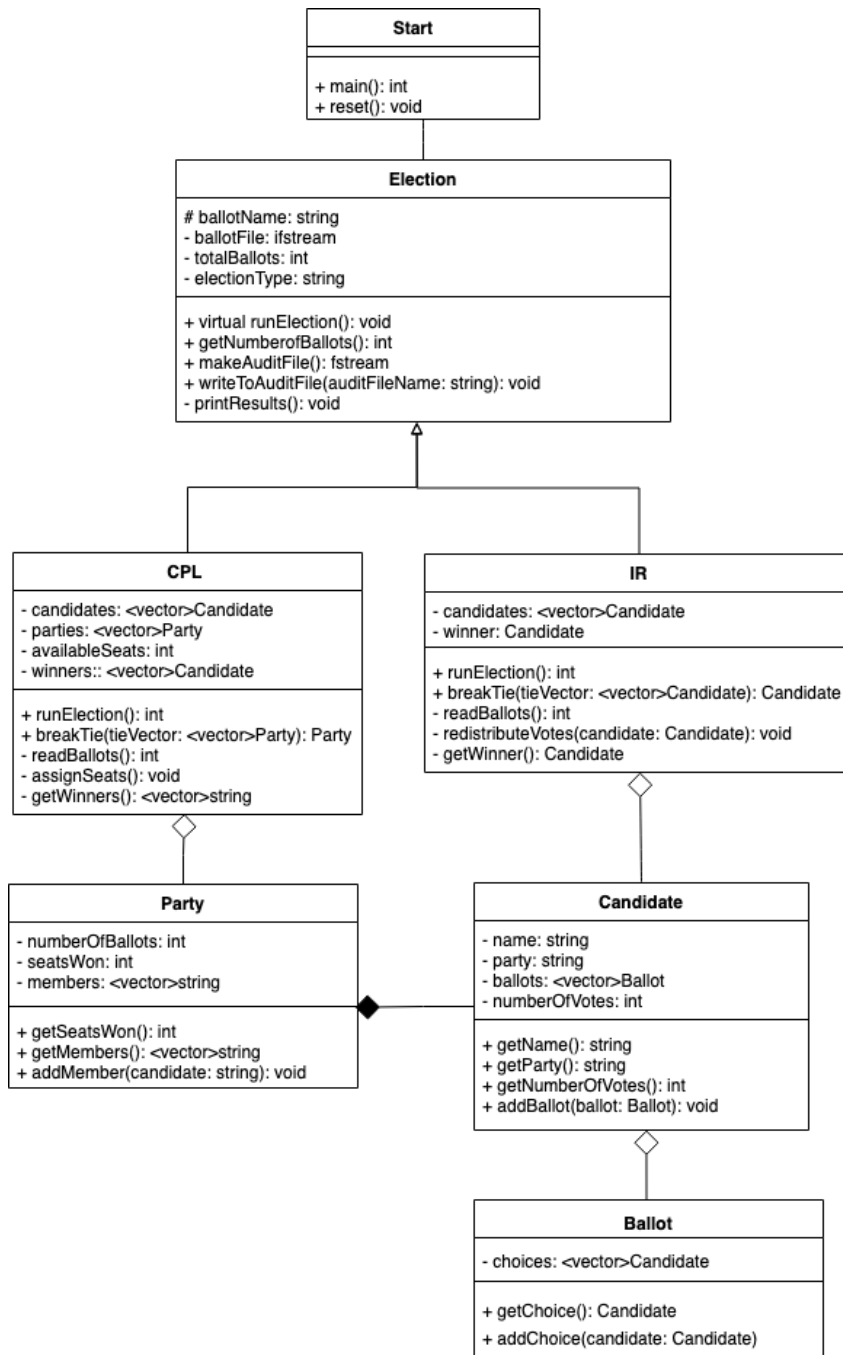


Figure 3.2.1: Class Diagram for Voting System

The *Start* Class holds the main function which begins the voting system. The *Election* Class is the class that handles the processing of the election such as getting the number of ballots included in the ballot file, making and writing to the audit file, and printing the results of the election. It is a base class for the CPL and IR classes, so the election process will be different between an IR election and a CPL election. The *CPL* Class runs a Closed Party List Election by reading ballots from a provided ballot file. It also handles ties in the highest remainder of a party and if a party is allocated more seats than they have candidates. The *IR* Class runs an Instant Runoff Election by reading ballots from a provided ballot file. It handles the redistribution of votes if there is no candidate with more than 50% of the votes, and also handles ties. The *Candidate* Class stores ballots to improve efficiency, and it contains information about each candidate in the election. The *Party* Class is used to identify what party a candidate belongs to. It is especially useful in a CPL election where votes are cast for parties rather than candidates. The *Ballot* Class is used to determine the rankings for candidates from a ballot in an IR election

The UML sequence diagram for a Closed Party List election is shown below. It can also be referenced at *SequenceDiagramCPL_Team7.pdf* for a higher-quality photo.

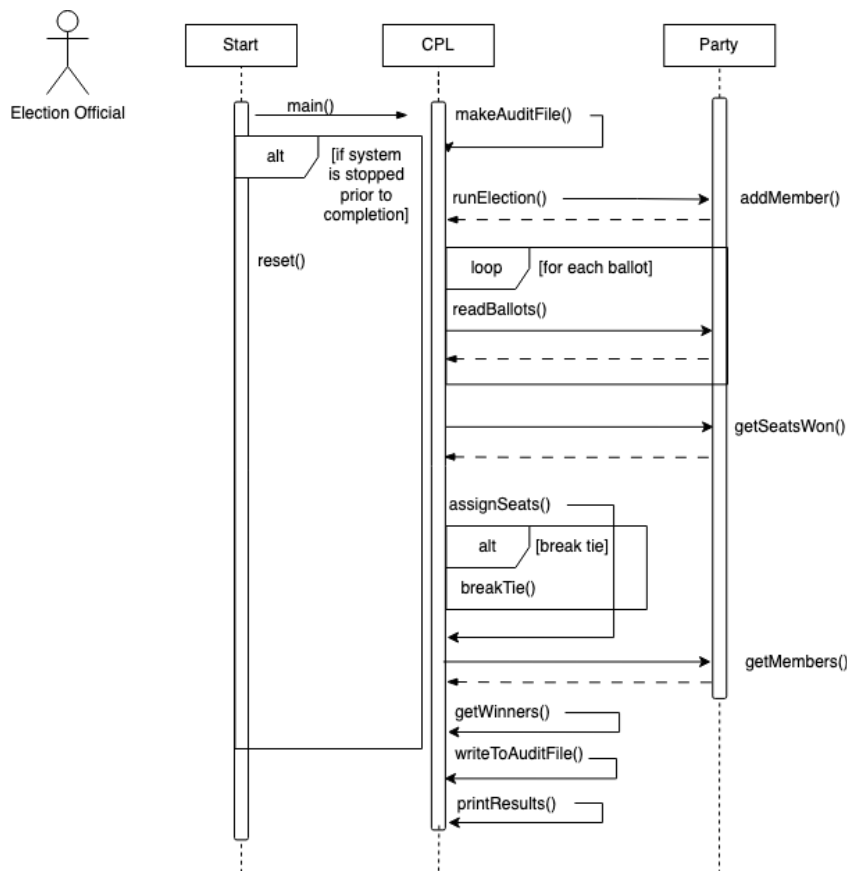


Figure 3.2.2: Sequence Diagram for Closed Party List Election

An election official begins a Closed Party List election by entering a valid ballot file. Once the voting system has been started, it handles the ballot file in the main function of the start class. Then, the system creates an audit file and begins to parse the provided ballot file. The type of election is determined to be CPL (or user input if the type of election is not included in the file), and the total number of ballots is parsed from the ballot file. After that, the voting system will continue to parse the ballot file to determine the candidates of each party, and their rankings as they are shown on the ballot. Once the header information is read and stored by the system, the system will go through the ballot information line-by-line. The system will determine which party the ballot counts towards and that ballot will be stored by that party. Once the system has reached the end of the ballot file, it will determine how many seats have been won by each party. If there are multiple parties with the same highest vote remainder or if a party has been allocated more seats than they have candidates, the system will break the tie fairly to determine the outcome. After that, candidates are assigned seats based on their ranking on the ballot and the number of seats their party won. Lastly, the results are recorded in the audit file and are also shown on the screen for the election official to view.

3.3 Design Rationale

The *Start* class was decided to be used to ensure not all of the code would be in one place. This class only deals with starting the system and handling the ballot file. We decided to use the classes *IR* and *CPL* for our voting system because there are different objects with an Instant Runoff Election and a Closed Party List Election. In an IR election, we decided to store ballots in the *Candidate* class which will save time and space and ensure efficiency is maintained. The *Ballot* class will be used in an IR election to count the votes of each candidate. Lastly, it was decided to use vectors to store candidates and parties since they are dynamic and allow for easy additions and deletions.

4. DATA DESIGN

4.1 Data Description

The voting system gathers election information from a CSV ballot file. The important header information is stored by the Election class for subclasses to use as needed. In an IR election, after the header information is read and stored, the rest of the file is read line-by-line and each line is stored as a Ballot object in the Candidate class. In a CPL election, after the header information is read and stored, the rest of the file is read line-by-line and the choices of each ballot are stored in the CPL class. For an IR election, the IR class redistributes votes for a Candidate. For a CPL election, the CPL class assigns seats to the winning Party.

4.2 Data Dictionary

Candidate Class:

Attributes:	Attributes Accessibility:	Attribute Type:
ballots	private	<vector>Ballot
name	private	string
numberOfVotes	private	int
party	private	string

Methods:	Method Return Type:	Method Parameters:
addBallot()	void	ballot: Ballot
getName()	string	–
getNumberOfVotes()	int	–
getParty()	string	–

CPL Class:

Attributes:	Attributes Accessibility:	Attribute Type:
availableSeats	private	int
candidates	private	<vector>Candidate
parties	private	<vector>Party
winners	private	<vector>Candidate

Methods:	Method Return Type:	Method Parameters:
assignSeats()	void	–
breakTie()	Party	tieVector: <vector>Party
readBallots()	int	–

runElection()	int	–
getWinners()	<vector>String	–

Election Class:

Attributes:	Attributes Accessibility:	Attribute Type:
ballotFile	private	ifstream
ballotName	protected	string
totalBallots	private	int
electionType	private	String

Methods:	Method Return Type:	Method Parameters:
getNumberOfBallots()	int	–
makeAuditFile()	fstream	–
printResults()	void	–
runElection()	void	–
writeToAuditFile()	void	auditFileName: string

IR Class:

Attributes:	Attributes Accessibility:	Attribute Type:
candidates	private	<vector>Candidate
winner	private	Candidate

Methods:	Method Return Type:	Method Parameters:
breakTie()	Candidate	tieVector: <vector>Candidate
readBallots()	int	–
redistributeVotes()	void	candidate: Candidate

runElection()	int	—
getWinner()	Candidate	—

Ballot Class:

Attributes:	Attributes Accessibility:	Attribute Type:
choices	private	<vector>Candidate

Methods:	Method Return Type:	Method Parameters:
addChoice()	void	candidate: Candidate
getChoice()	Candidate	—

Start Class:

Attributes:	Attribute Accessibility:	Attribute Type:
—	—	—

Methods:	Method Return Type:	Method Parameters:
main()	int	—
reset()	void	—

Party Class:

Attributes:	Attributes Accessibility:	Attribute Type:
members	private	<vector>String
numberOfBallots	private	int
seatsWon	private	int

Methods:	Method Return Type:	Method Parameters:
----------	---------------------	--------------------

addMember()	void	candidate: String
getMembers()	<vector>String	–
getSeatsWon()	int	–

5. COMPONENT DESIGN

The following section contains pseudo code implementations of all the classes within the program and their corresponding methods.

5.1 Start

main(int argc, char* argv[]):

if *argc* > 1

Set *ballotFileName* equal to the first command line argument

else

Prompt user for *ballotFileName*

Open *ballotFileName* to an fstream file pointer *ballotFile*

[*]while *ballotFile* == NULL

Prompt the user to re-enter *ballotFileName*

Attempt to reopen *ballotFile* with the new *ballotFileName*

Read in the first line of data from *ballotFile*, store it to String *electionType*

if *electionType* = 'IR'

Create new IR object *IRElection*, initialize known fields

IRElection.runElection()

IRElection.printResults()

IRElection.writeToAuditFile()

else if *electionType* = 'CPL'

Create new CPL object *CPLElection*, initialize known fields

CPLElection.runElection()

CPLElection.printResults()

CPLElection.writeToAuditFile()

else

Alert user to improper ballot file formatting

Return to the (*ballotFile* == NULL) loop, indicated by [*]

**if any of the IRElection or CPLElection function calls return 1 indicating error, the program will return to the (ballotFile == NULL) loop indicated by [*] **

```
void reset():  
    if (cin.eof())  
        cout << "System interrupted unexpectedly. Resetting to main menu"  
        main()
```

5.2 Election

```
void runElection():  
    (This function is to be implemented by inheriting classes CPL and IR)
```

```
int getNumberOfBallots():  
    return totalBallots
```

```
fstream makeAuditFile(String auditFileName):  
    Check if the directory AuditFiles exists in the current working directory  
    if not, create this AuditFiles directory  
    Attempt to open file auditFileName to an fstream file pointer auditFile  
    if auditFile failed to open  
        Print reason of failure  
        return NULL  
    return auditFile
```

```
void writeToAuditFile():  
    makeAuditFile()  
    if electionType == 'IR'  
        *write IR election auditing information to the auditFile*  
    else if electionType == 'CPL'  
        *write CPL election auditing information to the auditFile*
```

```
void printResults():  
    if electionType == 'IR'  
        *printout election results formatted to an IR election (figure 6.2.5)*  
    else if electionType == 'CPL'  
        *printout election results formatted to a CPL election (figure 6.2.6)*
```

5.3 CPL

```
int runElection():  
    readBallots()  
    assignSeats()
```

upon error (i.e. readBallots call returns 1), this function returns 1

Party breakTie(Party vector tieVector):

Set *tieCount* equal to *tieVector.size()*
 Calculate *tieWinner* as a random number between 1 and *tieCount* via *rand()* with
 the random seed set to the current time during execution: *srand(time(NULL))*
 return the Party at index *tieWinner* of *tieVector*

int readBallots():

Read the 2nd line of data and set *numParties* equal to the result
 from *index = 0* to *numParties*
 Read the first comma separated element in the 3rd line of data and store it
 as *partyName*
 Create Party *newParty* with the name equal to *partyName*
 Add *newParty* to *parties* vector
 from *index = 0* to *numParties*
 Read the 4th line offset by *index* number of lines, store to *temp*
 Parse *temp* into Candidate objects, add each to the *members* vector
 Read the next line of data after the candidates and store it in *availableSeats*
 Read the next line of data and store it in *totalBallots*
 from *index = 0* to *totalBallots*
 Set *ballotChoice* as the result of parsing the next line of data
 Increment the Party equal to *ballotChoice* in *parties numberOfBallots*
 field by 1
 return 0

**if any file operation fails in readBallots(), the function shall return 1 to indicate error.
 Additionally, if the program finds information is missing from the ballot file, it shall prompt the
 user to input this information when necessary**

void assignSeats():

Set *remainingSeats* equal to *availableSeats*
 Calculate *seatThresh* as $[totalBallots / remainingSeats]$
 for each *party* in *parties* vector
 if *party.numberOfVotes* > *seatThresh*
 party.seatsWon++
 remainingSeats--
 while *remainingSeats* > 0
 Set *largestRemainder* as 0
 Set Party *currParty* as NULL
 Initialize *tieVector* as an empty Party vector
 for each *party* in *parties* vector
 if *party.voteCount* > *largestRemainder*
 largestRemainder = *party.voteCount*
 currParty = *party*

```

        clear tieVector
    else if party.voteCount == largestRemainder
        add party to tieVector
    Define Party winner as currParty
    if tieVector.size() > 0
        add currParty to tieVector
        winner = breakTie(tieVector)
    if (winner.seatsWon + 1) > members.size()
        clear tieVector
        add each Party in parties to tieVector except winner
        winner = breakTie(tieVector)
    else
        winner.seatsWon++
    remainingSeats--
    return

```

<vector>String getWinners()

```

    Initialize winnerVector as an empty String vector
    for each party in parties
        from index = 0 to party.seatsWon
            add party.members.at(index) to winnerVector
    return winnerVector

```

5.4 IR

int runElection():

```

    readBallots()
    Calculate voteThresh as [totalBallots / 2]
    while (1)
        for each candidate in candidates
            if candidate.numberOfVotes > voteThresh
                Set winner = candidate
                return 0
    redistributeVotes()

```

upon error (i.e. readBallots call returns 1), this function returns 1

Candidate breakTie(Candidate vector *tieVector*):

```

    Set tieCount equal to tieVector.size()
    Calculate tieWinner as a random number between 1 and tieCount via rand() with
        the random seed set to the current time during execution: srand(time(NULL))
    return the Candidate at index tieWinner of tieVector

```


int readBallots():

Read the 2nd line of data and set *numCandidates* equal to the result
 from *index = 0* to *numCandidates*
 Read the first comma separated element in the 3rd line of data and store it
 as *candidateName*
 Create Candidate *newCandidate* with the name equal to *candidateName*
 Set *newCandidate.party* equal to the candidate's party shown by the data
 Add *newCandidate* to *candidates* vector
 Read the 4th line of data and store it in *totalBallots*
 from *index = 0* to *totalBallots*
 Create a new Ballot *ballot*
 for each Candidate *choice* contained in the parsed CSV data
 ballot.addChoice(choice)
 Add *ballot* to Candidate *ballot.getChoice()*'s *ballots* vector

**if any file operation fails in readBallots(), the function shall return 1 to indicate error.
 Additionally, if the program finds information is missing from the ballot file, it shall prompt the
 user to input this information when necessary**

void redistributeVotes():

Set *smallestRemainder* to *totalBallots*
 Set *currCandidate* as NULL
 Initialize *tieVector* as an empty Candidate vector
 for each *candidate* in *candidates*
 if *candidate.numberOfVotes < smallestRemainder*
 smallestRemainder = candidate.numberOfVotes
 currCandidate = candidate
 clear *tieVector*
 else if *candidate.totalVotes == smallestRemainder*
 add *candidate* to *tieVector*
 Define Candidate *toEliminate* as *currCandidate*
 if *tieVector.size() > 0*
 add *toEliminate* to *tieVector*
 toEliminate = breakTie(tieVector)
 for each *ballot* in *candidate.ballots*
 Define *nextCandidate* as *ballot.getChoice()*
 nextCandidate.addBallot(ballot)
 nextCandidate.numberOfVotes++

Candidate getWinner()

return *winner*

5.5 Party

int getSeatsWon():
 return *seatsWon*
vector<String> getMembers():
 return *members* String vector
void addMember(String toAdd):
 add *toAdd* to *members* String vector

5.6 Candidate

String getName():
 return *name*
String getParty():
 return *party*
int getNumberOfVotes():
 return *numberOfVotes*
void addBallot(Ballot ballot):
 add *ballot* to *ballots* vector

5.7 Ballot

Candidate getChoice():
 Pop and return the first Candidate from the *choices* vector
void addChoice(Candidate choice):
 Push *choice* into the *choices* vector at the largest index

6. HUMAN INTERFACE DESIGN

6.1 Overview of User Interface

Users will be able to access the program by executing it in a terminal interface. Before starting the program, users will have the option to input a ballot file name as a command line argument to the program's execution, allowing the program to execute without prompting the user for additional information. This command line argument is not necessary, and if the user chooses to execute the program without it, the program will prompt the user for a valid ballot file name via a command line prompt in the same terminal the program was executed. If the ballot file in either startup scenario is invalid (i.e. does not exist, improper formatting), the user will be command-line prompted to re-input a ballot file name until a valid ballot file is identified. Upon success, the program will proceed to read data from the ballot file. If information necessary to running the election is missing from the beginning of the ballot file, then the user will be command-line prompted for the missing information. Once the program has all of the necessary information, it will compute the results of the election in the background, and the user will not see more information until the election results have been determined.

The results of the program's execution on valid election data will produce two key outcomes. In the terminal where the user first started the program, a results printout will be presented to the user, providing them with a quick display of the results of the program. This results screen will contain important results data such as the election type, date, the candidates, their vote tallies / percentages, and the winner(s). In addition to the results printout, the program will compile an audit file to be used in election verification. Users will be able to access this file within a subdirectory in the same directory as the program titled "AuditFiles", and it will be given a name corresponding to the election type and election date. If any step in the creation of the audit file fails, the user will be notified in the terminal where the program is being executed. After both of these tasks are completed, the program will exit gracefully.

6.2 Screen Images

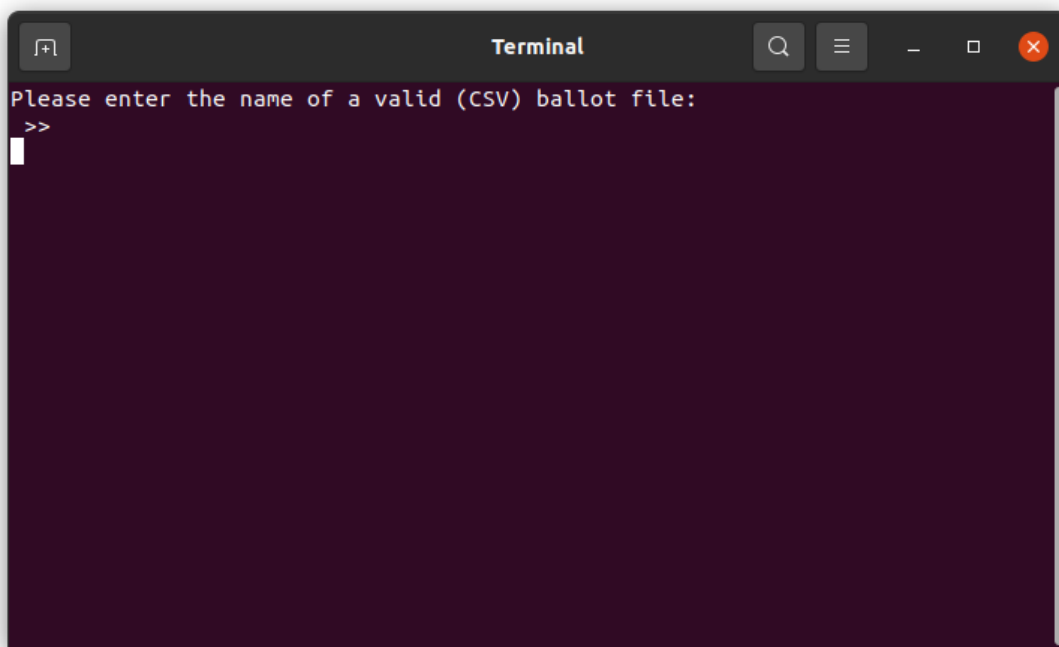


Figure 6.2.1: Example execution without command line arguments



```
Terminal
Valid ballot file identified
Processing data for IR election
[!] Ballot file is missing the number of total candidates
Please enter the number of total candidates:
>>
```

Figure 6.2.2: Example execution with command line input and missing ballot file information

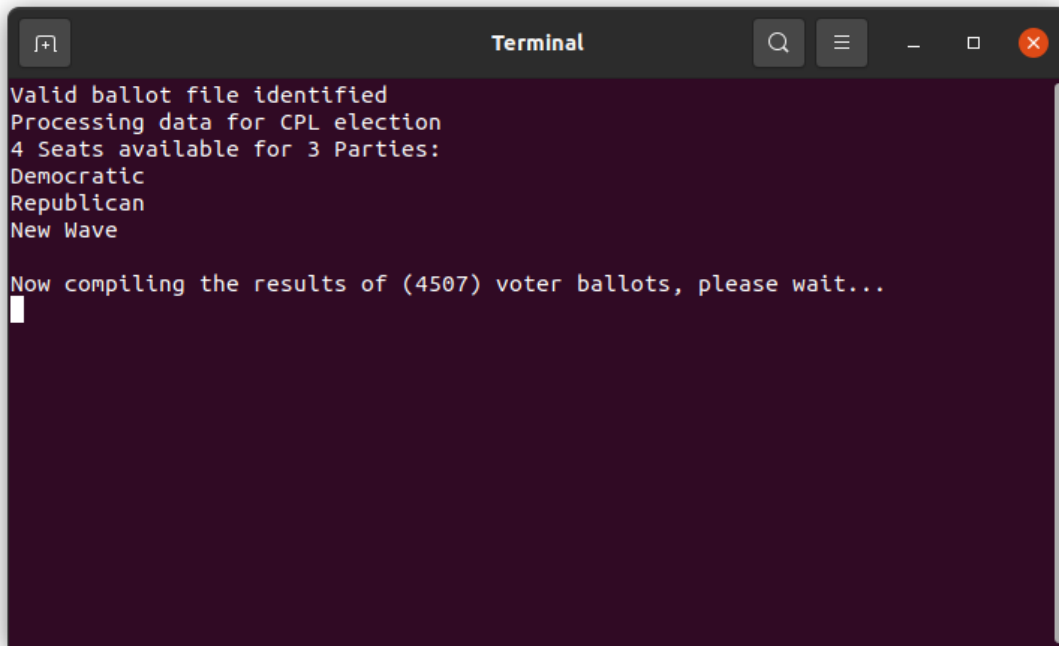


```
Terminal
Valid ballot file identified
Processing data for IR election
4 Candidates:
Rosen (D)
Kleinberg (R)
Chou (I)
Royce (L)

Now compiling the results of (5407) voter ballots, please wait...

```

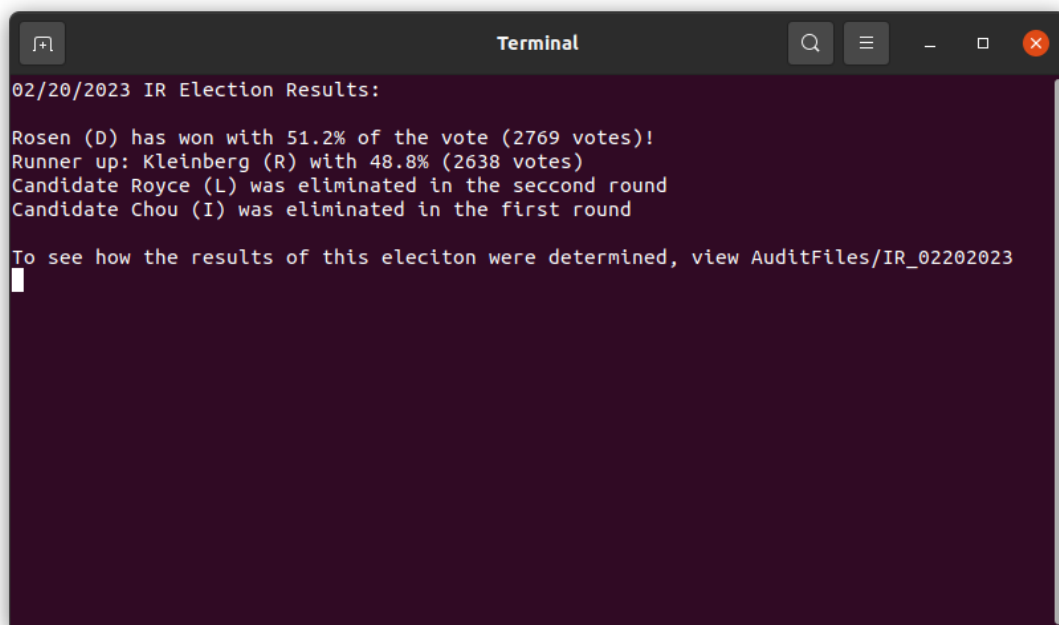
Figure 6.2.3: Processing screen for IR election

A terminal window titled "Terminal" with a dark background and light text. The text inside the terminal reads: "Valid ballot file identified", "Processing data for CPL election", "4 Seats available for 3 Parties:", "Democratic", "Republican", "New Wave", and "Now compiling the results of (4507) voter ballots, please wait...". A cursor is visible on the line following the last message.

```
Valid ballot file identified
Processing data for CPL election
4 Seats available for 3 Parties:
Democratic
Republican
New Wave

Now compiling the results of (4507) voter ballots, please wait...
█
```

Figure 6.2.4: Processing screen for CPL election

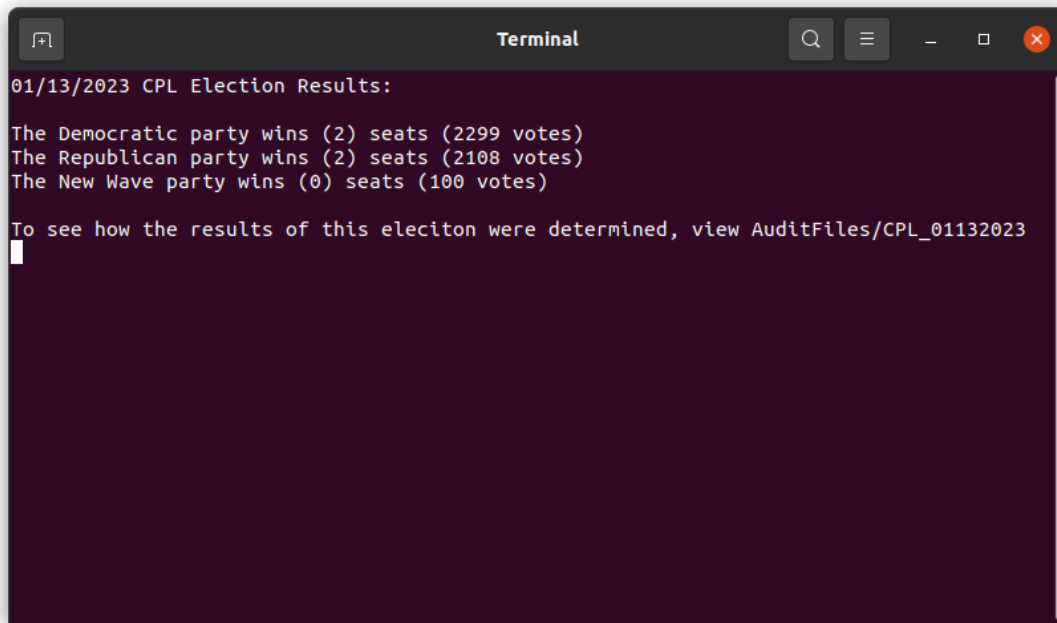
A terminal window titled "Terminal" with a dark background and light text. The text inside the terminal reads: "02/20/2023 IR Election Results:", "Rosen (D) has won with 51.2% of the vote (2769 votes)!", "Runner up: Kleinberg (R) with 48.8% (2638 votes)", "Candidate Royce (L) was eliminated in the second round", "Candidate Chou (I) was eliminated in the first round", and "To see how the results of this election were determined, view AuditFiles/IR_02202023". A cursor is visible on the line following the last message.

```
02/20/2023 IR Election Results:

Rosen (D) has won with 51.2% of the vote (2769 votes)!
Runner up: Kleinberg (R) with 48.8% (2638 votes)
Candidate Royce (L) was eliminated in the second round
Candidate Chou (I) was eliminated in the first round

To see how the results of this election were determined, view AuditFiles/IR_02202023
█
```

Figure 6.2.5: IR election results screen

A terminal window titled "Terminal" with a dark background and light-colored text. The text displays election results for 01/13/2023. It lists three parties: The Democratic party (2 seats, 2299 votes), The Republican party (2 seats, 2108 votes), and The New Wave party (0 seats, 100 votes). A final line of text instructs the user to view AuditFiles/CPL_01132023 for more details. A cursor is visible on the line following the instruction.

```
01/13/2023 CPL Election Results:

The Democratic party wins (2) seats (2299 votes)
The Republican party wins (2) seats (2108 votes)
The New Wave party wins (0) seats (100 votes)

To see how the results of this election were determined, view AuditFiles/CPL_01132023
```

Figure 6.2.6: CPL election results screen

6.3 Screen Objects and Actions

The design of the program leaves little for user interpretation and remains simple to use. What follows is a brief description of the prompts that the program can send to the terminal and what the user can do in response.

Ballot file prompt: If the user does not specify a ballot file in the command line arguments or the user inputs an invalid ballot file, the program will prompt the user to input a new ballot file name. The user must then type the name of another ballot file to proceed, and if the input file is found to be invalid, the program will re-prompt for another ballot file name. An example of this behavior can be seen in *figure 6.2.1*

Critical information prompt: If the ballot file read by the system is missing election critical information, including the number of candidates, the type of election, or the candidates themselves, the program will prompt for the missing information. The system will tell the user what data they are missing and will prompt them to input this data. An example of this behavior can be seen in *figure 6.2.2*

Two other screens will be seen by the user during execution, but the user will have no ability to interact with them. Below is a short description of each screen.

Processing screen: Once the program has all of the critical information needed to run, it will begin to process the raw votes contained in the ballot file while displaying a brief information window to the user. This window will contain the critical information like election

type or the candidates / parties, and will ask the user to “please wait” as the program processes the data. Examples can be seen in *figure 6.2.4* and *figure 6.2.5*.

Results screen: After the results of the election have been determined, the program will display information regarding the results of the election to the terminal. This will include data such as the winner(s), vote percentages, vote counts, and the name of the audit file made in conjunction with the results processing. Examples can be seen in *figure 6.2.5* and *figure 6.2.6*.

7. REQUIREMENTS MATRIX

Use Cases (Title/#)	Objective	Implementation
UC_1: Starting the System	Start the software that will count votes.	(5.1) Start main()
UC_2: Ballot File Input & Identification	Once software is started, the user prompts a ballot file	(5.1) Start main()
UC_3: Determining the Election Type	Reading in the first line of the ballot file to determine the type of election	(5.1) Start main()
UC_4: Creating the Audit File	While the ballot is processed a file is created to provide transparency in the election	(5.2) Election makeAuditFile(),
UC_5: Naming the Audit File	Follows a certain naming convention when the audit file is created	(5.2) Election makeAuditFile()
UC_6: Ballot File Processing	The rest of the ballot file is processed	(5.3) CPL readBallots() and (5.4) IR readBallots()
UC_7: Calculating a Winner in an Instant Runoff Election	A winner is decided when a candidate receives more than 50% of the vote after elimination	(5.4) IR runElection()
UC_8: Calculating the Winner(s) in a Closed Party List Election	Parties that receive vote totals above the calculated ‘seat quota’ then assigned to candidates within a party based on their order of appearance on the original	(5.2) CPL runElection()

	ballots	
UC_9: Calculating a Winner in a Tie	The system shall decide a winner fairly to break the tie.	(5.3) CPL breakTie() and (5.4) IR breakTie()
UC_10: Display the Winner(s) to the Screen	The program shall display the results of the election, including the winner(s), individual vote tallies, and vote percentages to the terminal for viewing	(5.2) Election: printResults()
UC_11: System Shutdown During Execution	If the system is stopped/shutdown while it is in the process of processing a ballot file, it will reset to the main menu	(5.1) Start reset()

8. APPENDICES

Figure 3.1.1: Activity Diagram for Instant Runoff Election

Figure 3.2.1: Class Diagram for Voting System

Figure 3.2.2: Sequence Diagram for Closed Party List Election

Figure 6.2.1: Example execution without command line arguments

Figure 6.2.2: Example execution with command line input missing ballot file information

Figure 6.2.3: Processing screen for IR election

Figure 6.2.4: Processing screen for CPL election

Figure 6.2.5: IR election results screen

Figure 6.2.6: CPL election results screen