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 10

Voting System

Software Design Document

Name (s): Jacob McIsaac,
Dan Kong, Caleb Otto,
Ruolei Zeng
Lecture Section: 1

Date: 03/03/2023

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose

This software design document is focused on describing the system design and architecture of the 001 version of the Voting System Software. This document will be resourceful for several audiences including testers, developers, election officials, and any other notable stakeholders who will be involved in the software's testing and development.

1.2 Scope

The voting system being developed here will act as a tool that will assist election officials in determining election winners in two different voting system types (Closed Party-List Voting and Instant-Runoff Voting). This system will be taking in CSV files having ballot information then determining election winners based on the specified/conducted election type. This software is designed to be producing audit files for reviews after elections and sharing the election results with notable stakeholders such as the media personnel. This software will mainly be beneficial in that it will allow election results to be determined in a fast, safe, and clear manner. These results can then be quickly reported and verified. The hope is that this software can be used for every election, which will mean many times per year.

1.3 Overview

In this document, there will be a brief introduction to the key functions and purpose of the voting system which will be followed by an in-depth analysis of the system overview. This document will proceed to give another in-depth assessment of the voting system's architecture and design (component designs, human interface designs, data designs, and pseudocode design), as well as the requirements matrix.

1.4 Reference Material

SRS_Team#10: Software Requirements Specification document developed for this Voting System. This document can be found under the "SRS" directory in this repository.

UseCases_Team10: This is the document that contains all of the information regarding the use cases for this Voting System. It is located in the "SRS" directory in this repository. Refer to Section 7 of this document to see how these use cases are accounted for in the system design.

1.5 Definitions and Acronyms

IRV: Instant runoff voting. This style of election involves people ranking their preferences for candidates. Each candidate's first place votes are compared. If one candidate has a majority, they are the winner. If no candidate has a majority, the candidate who has the fewest votes has their votes distributed to the remaining candidates; this means that those who voted for the losing candidate will give their second choice a vote. This process of elimination and redistribution continues until one candidate has more than 50% of the votes.

CPL: Closed party list voting. This type of voting is where people vote for a particular party. Using a formula called the largest remainder formula, each party wins an initial number of seats that are allocated to that party based on the ordering of the candidate list. Candidates with the largest number of remaining votes after this initial seat allocation get the remaining seats.

CSV: Stands for "comma-separated value". Describes a file with extension .csv. These files contain values on each line, and these values are separated by commas.

TXT: Short for text. Specifies a file type that has the extension .txt. These files contain ASCII characters.

2. SYSTEM OVERVIEW

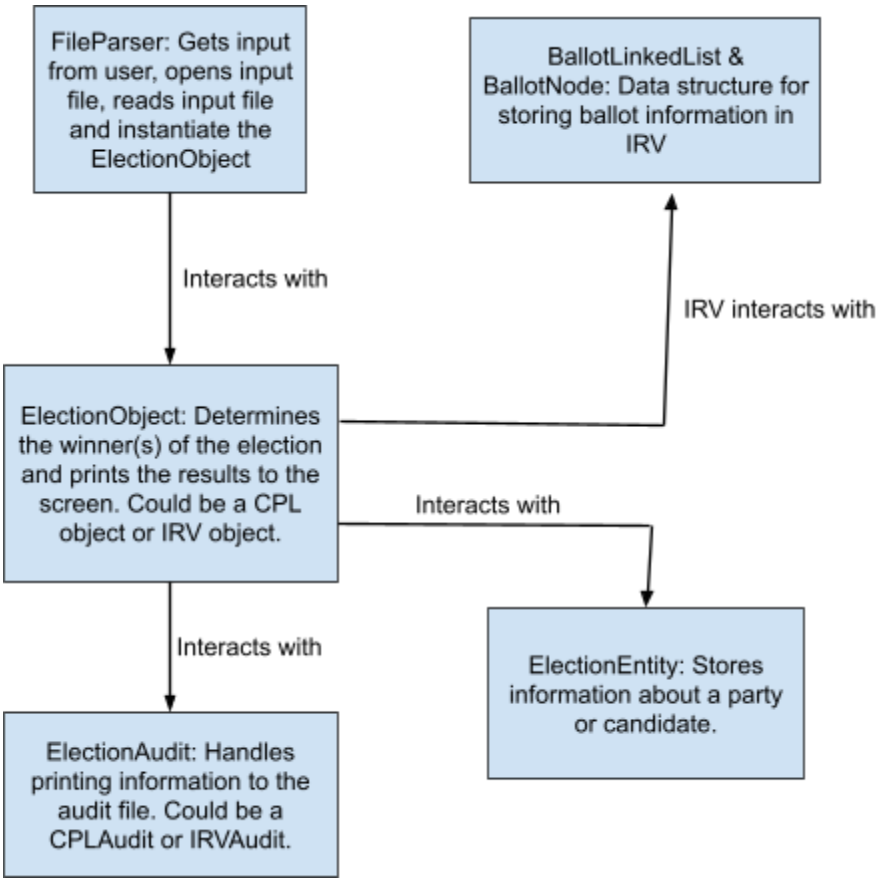
This voting system has been specifically designed for the optimization of the overall election process through automating the analysis of ballot information (for instance, counting votes). The software will be utilized to determine election winners contingent upon the type of election that has been conducted (either IRV or Closed Party-List voting). This software will then produce audit files that will be used to review the credibility and accuracy of the election outcomes. It will also print a summary of the results of the election to the screen which could then be shared with media personnel. Interaction with the user is minimal; the user may be asked to input the input file name if no file name was given on the command line and the user is asked to enter the election date. The system design, which you will see below, is designed to account for all of this functionality.

3. SYSTEM ARCHITECTURE

3.1 Architectural Design

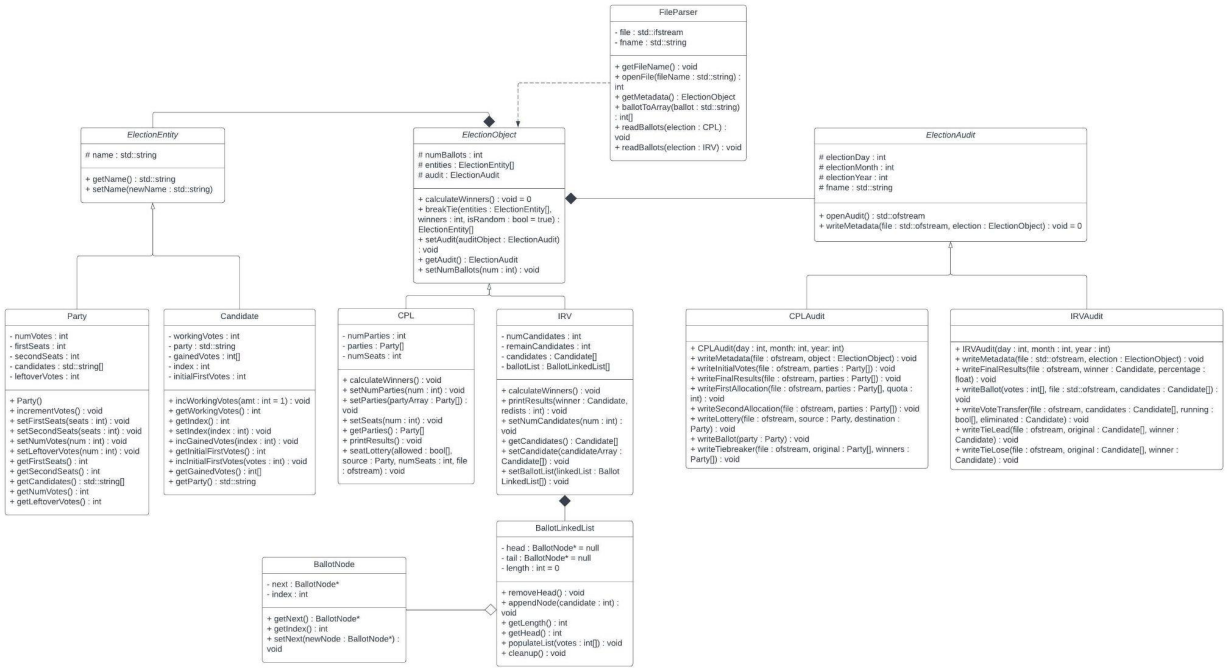
The first major responsibility of the system is to process the input file for information while also prompting the user for the file name and any other necessary information. These tasks have been delegated to the FileParser class. This class will get the file name, open the file, parse the file for information, and populate the fields of an ElectionObject object based on

this mined information. The next major responsibility is processing the information from the election in order to determine the winner or winners of the election. This is the responsibility of the ElectionObject class; more specifically, it will be the responsibility of the IRV and CPL classes (which both inherit from the abstract class ElectionObject). The ElectionObject class is also responsible for breaking any ties that occur during the winner calculation. It is also worth mentioning that these ElectionObject child classes will also be responsible for eventually printing out the results of the election to the terminal screen. In order to process the election data and determine the winners, the ElectionObject object will interact with and process the data contained in ElectionEntities, which will either be candidate objects (in the case of IRV) or party objects (in the case of CPL). These ElectionEntity objects will contain important information about a candidate or party including the number of votes for that entity, the name, etc. It is worth noting that, in order for the IRV class to determine the winner of an IR election, it will also contain and will interact with a BallotLinkedList object. The BallotLinkedList is made of ballot nodes, and is a specialized data structure which is designed to represent ballots in an IR election and also to make handling vote redistribution possible. The final major responsibility of the system is to populate the audit file. This responsibility is delegated to the ElectionAudit class (which has two children, IRVAudit and CPLAudit). The ElectionObject object contains an instance of the ElectionAudit class. The ElectionAduit class will write any necessary information to the audit file as the winner calculation is being done. These responsibilities and interactions are summarized in the diagram below:

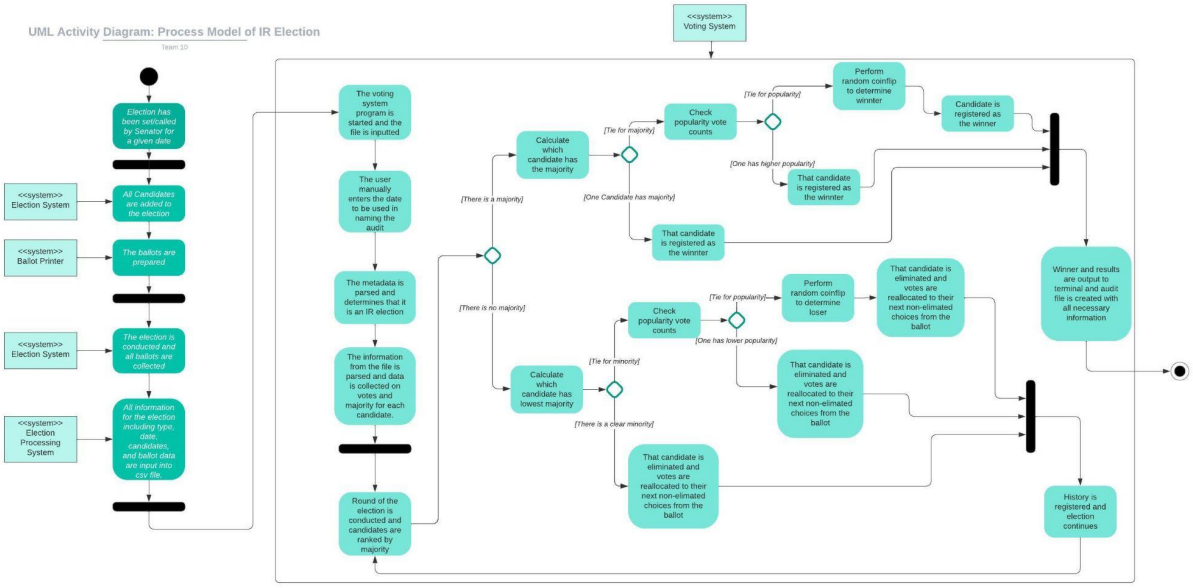


3.2 Decomposition Description

Note that in the UML diagram below, abstract class names are italicized and pure abstract methods are labeled with "=0" after the method signature.

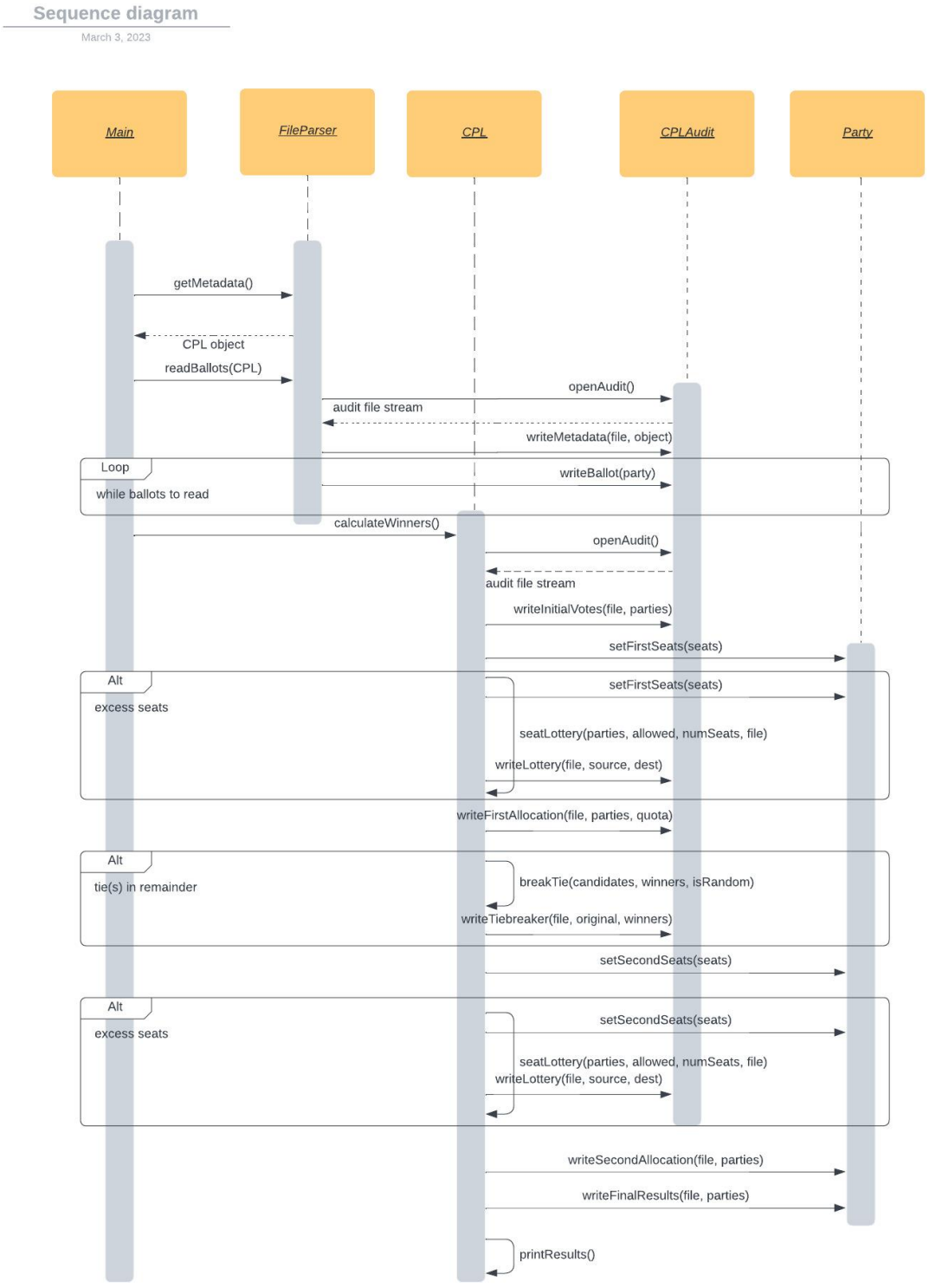


This UML diagram describes the actual structure and organization of the voting system as well as how all the different classes and objects interact with each other. It begins with the FileParser class (at the top) that will parse the input file to gather all necessary information which it will pass to the ElectionObject class to create either a CPL or IRV object depending on the type of election data submitted. The ElectionObject will also create an ElectionAudit object that will either be CPLAudit or IRVAudit depending on the type of election as well. Inside each election object (either CPL or IRV) there are ElectionEntities that will be either Parties (for CPL) or Candidates (for IRV). These classes will interact to run the proper election and then calculate the winner for either a CPL or IRV election and then output this winner and data to the user through the terminal as well as to the audit file for further inspection at a later date. It is also worth noting that the BallotLinkedList and BallotNode classes are part of a specialized data structure that was developed specifically for determining the winner in an IR election; see Section 4.1 for more information on this data structure. See Section 4.2 for a more detailed view into the methods and interactions seen in the UML diagram.



This activity diagram outlines the process for running an instant runoff election. It begins at the point at which an election has been called/set for a given date and it ends at the point that a winner has been determined. Following the path of the diagram, the election first must be officially called by a Senator. The next step is to have all candidates be declared and added to the election as well as preparing the ballots with these candidates using the Election System and Ballot Printer System. The next stage is actually conducting the election and collecting all data using the Election System as well. This data must then be input into a csv file to be used later by our program by utilizing an Election Processing System. Finally, the top of the second column from the left is the point at which the program is actually started and defines the start of the actual Voting System block. It begins with having the file inputted by prompting the user. The user is also prompted to enter the date which will be used to name the audit file correctly. After inputting the file, the program begins parsing it, first collecting the metadata to determine that it is indeed an IRV election and determining the candidates and secondly, collecting all the votes and what majority of votes each candidate has. The next step is where the instant runoff process truly starts by initiating a round of the election and ranking the candidates by their majorities. This leads to two possible paths, one in which there is a majority for candidates, and one in which there is no clear majority(50% of votes). For the path where there is a majority(the top path) it begins with calculating which candidate has the majority. If one candidate has the clear majority, they are declared as the winner and the winner and results are output from the program to the terminal and the audit file. For the path where there is a tie for the majority (50/50) then the next step would be checking the popularity vote counts of the candidates to see if they differ. If one has a higher popularity, they are declared the winner and the program proceeds as it did previously for registering them and outputting the results. If they are tied for popularity, then a random coin flip tiebreaker is performed to determine the winner which is then handled as stated

previously. Continuing on, for the lower path in which there was no majority, the next step is to determine which candidate is in the lowest position. If there is a clear minority, the candidate in question is eliminated and their votes are reallocated to the next non-eliminated choices of their ballots. This round is registered in the history and then the election continues with another round. If there is a tie for the minority, the next step(similar to if there is a tie for the winner) is to check their popularity votes and then eliminate the one with the lower vote count before running another round. If they are tied for popularity as well, then a random coin flip is performed to decide who to eliminate and reallocate before conducting another round. This process continues until a majority is determined and a candidate declared the winner.


Sequence diagram
March 3, 2023
Main
FileParser
CPL
CPLAudit
Party
getMetadata()
CPL object
readBallots(CPL)
openAudit()
audit file stream
writeMetadata(file, object)
Loop
while ballots to read
writeBallot(party)
calculateWinners()
openAudit()
audit file stream
writeInitialVotes(file, parties)
setFirstSeats(seats)
Alt
excess seats
setFirstSeats(seats)
seatLottery(parties, allowed, numSeats, file)
writeLottery(file, source, dest)
writeFirstAllocation(file, parties, quota)
Alt
tie(s) in remainder
breakTie(candidates, winners, isRandom)
writeTiebreaker(file, original, winners)
setSecondSeats(seats)
Alt
excess seats
setSecondSeats(seats)
seatLottery(parties, allowed, numSeats, file)
writeLottery(file, source, dest)
writeSecondAllocation(file, parties)
writeFinalResults(file, parties)
printResults()

This sequence diagram defines the process of running the closed party listing election using the voting program. The process begins at the moment of receiving the ballots into the system from the file to the point at which a winner is declared. Reading left to right, the objects declared for this system are the actual main sequence, the file parser, the CPL class, the CPL audit class, and the Party class. Main is responsible for actually initiating much of the process and keeping the election moving forward. The FileParser parses the file in order to obtain all the necessary information for running the election. The CPL class takes in the data and uses it to calculate the winning parties as well as seat allocations. The CPLAudit class is responsible for taking in data about the election and using it in order to create the audit file to be used at the end. Lastly, the Party object is what is used for storing the data (including seat order, ballots, etc) for each party in the election. Following the actual sequence diagram, it begins with the main function obtaining metadata from the file parser in the form of a CPL object. It then obtains the actual ballot data and records from the file parser as well. The file parser will also open an audit file upon being called in order to obtain a file stream for writing to the audit along the election. It will use this stream to write the actual metadata and ballot information, within the first loop to the audit. The next stage in the sequence is the main using the CPL object to actually calculate the winners of seats with the CPL class. The CPL class will then write the initial vote counts to the audit. It will then go through the process of actually allocating the first seats which will be written to the audit file once determined. If a party earns excess seats in the 1st alternate path, then there will be a seat lottery to determine where the excess seats go in addition to logging this process to the audit file. The second alternate path would be if there are any ties between parties for seats in which a tiebreaker with a random coin flip would be used to break it, allocate the seats, and then log this in the audit. The 3rd and last alternate path we have is for setting and allocating the second seats to parties if they earned more seats than they have candidates. This also contains a possible lottery like in the first allocation path. Lastly, the CPL object outputs the final results of the election including which parties won seats and who they were allocated to in addition to writing these final results to the audit file as well, concluding the program.

3.3 Design Rationale

There were three major parts to the design in the architecture described in 3.1: file parsing, auditing, and how different election types are handled. A static, singular function was considered for file parsing, however, this was considered lacking for the system, as it would have to perform multiple tasks rather than a singular one. Examples include: handling errors, handling user input for the date (to name the audit file), passing parameters to ElectionObject objects, and also having an overloaded function to handle both types of elections. As can be seen, a single function would not nearly be enough to handle all these tasks.

For the auditing, it was decided that writing to the audit file should be done in tandem with processing/receiving the data, as opposed to writing everything to the file at the end of the program. This is because writing at the end requires storing the data, and most of the data is stored in arrays, resulting in having to iterate through the arrays again. Instead, information about the election is written to the audit as soon as the data is processed, so that there's no

need to store the information, and that the audit is populated with the correct information in a sequential manner (as audit information is printed in the same order as how the data is processed).

The way that the classes are created depends on the information from the file parser, which instantiates the correct election object by calling the overloaded function, readBallots(). The design of the classes for different election types (CPL or IR) differed greatly from each other, beyond the shared methods from ElectionObject. Since both elections share some functionality (that being that they require a winner, an audit, a tiebreaking function, and printing to screen the results of the election), it was reasoned to create a parent class so that, for extensibility, could be inherited from, and so that certain functions or variables were not copy pasted (such as numBallots, entities, audit, getAudit() or breakTie()), requiring to update code in multiple places in case of a refactor. The IR election type's architecture specifically requires the use of a linked list, as it was reasoned that the fastest and easiest way to handle vote redistribution in the case of candidate elimination was to check the head of a linked list for candidates, and continually remove nodes until a "valid" (still running) candidate was found. This was the most orderly way to retain the order of the candidates without having to iterate over another array, which would cause a significant increase in search time in the case of vote redistribution. To process these IR votes, the system also "translates" the ballot's ordering of the votes to the system's ordering of the votes, as the ballot has the candidate's position at the index, whereas the system has the candidate's place at the index; this was done for ease of processing, as otherwise the linked list pattern would not work in the intended fashion.

For CPL, the handling of the ballots is much simpler. As each party gets only one vote per person, it is simple to calculate the quota and the number of seats that each party gets. Since there are only two allocations, there was no requirement to use such a pattern as described for IR, as this only required handling of integers, so no pattern or object for CPL was deemed necessary. Due to the major differences between the two current election types, it was reasoned that both types needed their own type of audit to write to the audit file, as each election handles the ballots much more differently than the others. However, since both elections required audits, a parent class was created for auditing, for the same extensibility reasons as described above. Each election type stores an audit of the corresponding type (I.E. CPL stores CPLAudit in the audit member variable, and IRV stores an IRVAudit object in its audit member variable). This is because an election requires an audit, and an election should not have an audit that does not match the election type.

4. DATA DESIGN

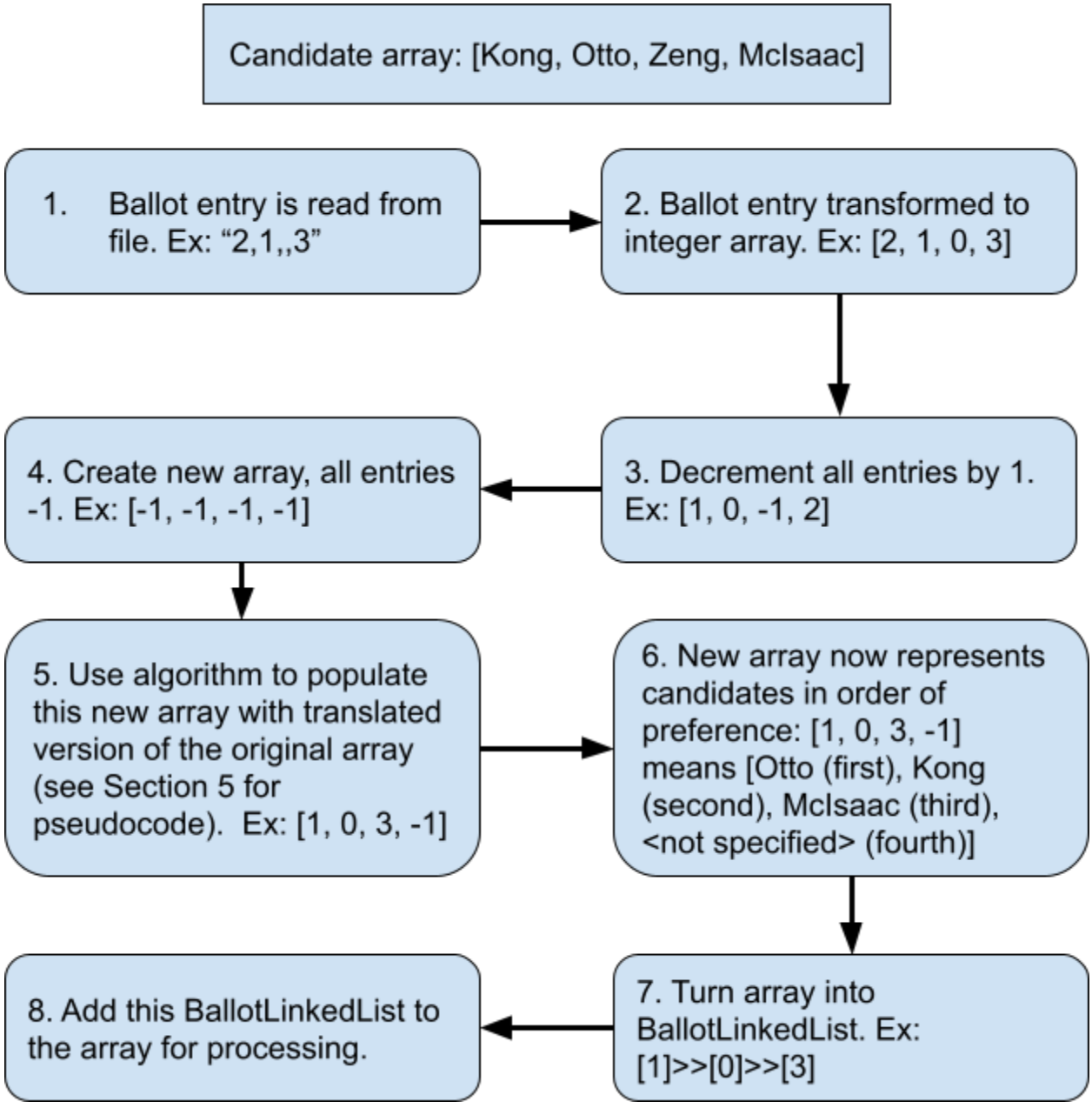
4.1 Data Description

An ElectionObject (which can be either an IRV or CPL object depending on the election type found in the input file) will contain an array of ElectionEntity objects, which will either be Party objects (in the case of CPL) or Candidate objects (in the case of IRV). These ElectionEntity objects will contain relevant information for that party or candidate, including the entity's number of votes, the entity's name, and many other pieces of information (see 4.2 for a detailed list). In the ElectionObject, this array of ElectionEntity objects will be processed by the member functions of the ElectionEntity object in order to determine the winner(s) of the election, print out results about the election, and write the election process to the audit file.

The election file that is given to the system is processed by the FileParser class, and the data is then used to instantiate an ElectionObject and populate its fields (including the ElectionEntity array). Exactly how this is done will depend on the election type. Election metadata, which is found in the header of the file and will depend on the election type, is read first and will be stored as member variables in the ElectionObject object (see 4.2 for all members of this class). The most important data transformation occurs with the ballot data, which is present in the file after the header containing the metadata. There is one ballot per line, and what the ballot information looks like will depend on the election type. For CPL, this will simply be a line of commas with a "1" in a certain position in the comma-separated list. The position of the "1" will dictate the party that that person voted for, as it aligns with the position that the party is listed in the party list of the header. Processing of the ballots is straightforward here; the FileParser class will simply increment the numVotes field of the Party object that gains a vote. This information is then processed in the CalculateWinners method in the CPL object to determine the winners of the CPL election.

When it comes to processing ballots for IRV, the data transformation is less straightforward. We have designed a special data structure to handle the ballot information in the IRV class. Ballots in IRV are comma-separated lists of numbers. See section 2.7 of the SRS document for a full description of what the ballots will look like and the assumptions that are made regarding them. An IRV object will contain a BallotLinkedList array. Each entry in this array will represent a ballot read from the file. The linked list for a ballot will contain a string of BallotNodes. Each BallotNode will contain an integer; this integer represents the index that a particular candidate is located in the Candidate array that is stored in the IRV object. Initially, the order of the BallotNodes in the BallotLinkedList represents the preference that that ballot has for that particular candidate. For example, if the first BallotNode in one of the lists has index "1" that means that this particular ballot chose the Candidate object at index 1 in the Candidate array as its first choice. These BallotLinkedList objects will be manipulated during the runtime of the program to account for IRV vote redistributions (see Section 5 for full details). The difficulty is that ballot information in the input file is not represented in the same way (again, see SRS section 2.7). Therefore, there will need to be a translation process that translates ballot information in the input file into our internal representation of a ballot in

the IRV class. This translation process is outlined in the flowchart below (pseudocode can be found in Section 5).



Note that if a blank is found in the ballot line, it becomes a zero in the ballot entry array initially. Then, when the second array is made into the BallotLinkedList, -1 entries (which will always be at the end of the array if some preferences were not specified), will not be added to the linked list. As a final note, the object that handles writing information to the

audit file is called an “ElectionAudit” object (which will either be a CPLAudit object or IRVAudit object depending on the election) and it will reside in the ElectionObject object. This class contains methods that will process ElectionEntity objects and other information in order to print necessary information to the audit file.

4.2 Data Dictionary

Concrete Class: BallotLinkedList	
Attribute	**Description**
BallotNode* head	This is the head node of the linked list.
BallotNode* tail	This is the tail node of the linked list.
int length	This is the length of the linked list (number of nodes).
Method	**Description**
removeHead() : void	This function will remove the node currently add the head of the linked list and make the next node the head (if there is one).
appendNode(candidate : int) : void	This function will add a new node containing the given integer to the end of the linked list.
getHead() : int	This will return the number that is stored in the head node of the linked list.
populateList(votes : int[]) : void	This will create a linked list from an integer array. The head node will hold the first array entry and so on.
cleanup() : void	This will free all of the nodes in the linked list.
getLength() : int	This returns the length attribute.

Concrete Class: BallotNode

Attribute	**Description**
BallotNode* next	Pointer to the next node in the list.
int index	The index of the candidate in the candidates array that this node is representing.
Method	**Description**
getNext() : BallotNode*	Returns the next attribute.
getIndex() : int	Returns the index attribute.
setNext(newNode : BallotNode*) : void	Sets the next pointer attribute.

Concrete Class: Candidate	
Attribute	**Description**
int workingVotes	Number of votes that a candidate currently has during IRV.
std::string party	The name of the party that this candidate belongs to.
int[] gainedVotes	An array that keeps track of how many votes this candidate gained for each round of vote redistribution.
int index	The index that this candidate is located at in the candidates array in IRV object.
int initialFirstVotes	The number of first place votes that the candidate had initially.
Method	**Description**
incWorkingVotes(amt : int = 1) : void	This will increase workingVotes by a certain amount; 1 by default.
getWorkingVotes() : int	Returns the workingVotes attribute.
getIndex() : int	Returns the index attribute.
setIndex(index : int) : void	Sets the index attribute.

incGainedVotes(index : int) : void	Increases the gainedVotes attribute by a specified amount.
getInitialFirstVotes() : int	Returns the initialFirstVotes attribute.
incInitialFirstVotes(votes : int) : void	Increases the initialFirstVotes attribute by a specified amount.
getGainedVotes() : int[]	Returns the gainedVotes attribute.
getParty() : std::string	Returns the party attribute.

Concrete Class: CPL	
Attribute	**Description**
int numParties	The number of parties in the CPL election.
Party[] parties	An array of Party objects which represent the running parties in the CPL election.
int numSeats	The total number of seats available for distributing.
Method	**Description**
calculateWinners() : void	Finds the winner of the CPL election.
getParties() : Party[]	Returns the parties array.
printResults() : void	Prints the results of calculateWinners() to the screen.
seatLottery(allowed : bool[], numSeats : int, file : ofstream) : void	Distributes a number of seats randomly to the allowed parties, and calls writeLottery() to write to the audit file which party gained seats.
setNumParties(num : int) : void	Sets the numParties attribute.
setParties(partyArray: party[]) : void	Sets the parties attribute.
setSeats(num : int): void	Sets the numSeats attribute.

Concrete Class: CPLAudit	
Attribute	**Description**
None	N/A
Method	**Description**
CPLAudit(day : int, month : int, year : int)	Contructor for the CPLAudit class. Takes in the day, month, and year of the election.
writeMetadata(file : ofstream, object : ElectionObject) : void	This method writes the CPL metadata (number of parties, names of the parties, names of candidates in the parties, and number of seats) to the audit file stream.
writeInitialVotes(file : ofstream, parties : Party[]) : void	This method writes the number of initial votes that each party has to the audit file stream.
writeFinalResults(file : ofstream, parties: Party[]) : void	This method will write to the audit file each party's name, their number of initial votes, number of initial seats gained, number of leftover votes, number of secondary seats gained, their final seat total, and the candidates that earned seats for the party.
writeFirstAllocation(file : ofstream, parties : Party[], quota : int) : void	This method will write to the audit file what the quota was as well as how many seats were given to each party in the first allocation of seats (the firstSeats attribute of Party).
writeSecondAllocation(file : ofstream, parties : Party[]) : void	This method will write to the audit file the number of seats that were given to each party during the second seat allocation (the secondSeats attribute of Party).
writeLottery(file : ofstream, source : Party, destination : Party) : void	This function simply writes to the audit file that a seat was lotteried off from the source party to the destination party.
writeBallot(party : Party) : void	This function writes to the audit file that the current ballot voted for the party passed in

	as an argument.
writeTiebreaker(file : ofstream, original : Party[], winners : Party[]) : void	This function writes to the audit file the parties involved in a tiebreaker scenario and the party or parties that ended up being chosen as the winner or winners of the tie.

Abstract Class: ElectionAudit	
Attribute	**Description**
int electionDay	Day of the month that the election took place as an integer.
int electionMonth	Month that the election took place in the year as an integer.
int electionYear	Year that the election took place as an integer.
std::string fname	This is the name of the audit file.
Method	**Description**
openAudit() : std::ofstream	This function opens the audit file using the fname attribute and returns the file stream.
writeMetadata(file : std::ofstream, election : ElectionObject) : void	This is an abstract method that is not implemented in this class. The child classes (CPLAudit and IRVAudit) implement this function to write their appropriate metadata to the audit file.

Abstract Class: ElectionEntity	
Attribute	**Description**
std::string name	The name of the entity.
Method	**Description**
getName() : std::string	Returns the name attribute.
setName(newName : std::string) : void	Sets the name attribute.

Abstract Class: ElectionObject	
Attribute	**Description**
int numBallots	Number of ballots for the election.
ElectionEntity[] entities	The entities associated with this election (either Party or Candidate objects depending on the election type).
ElectionAudit audit	The audit object for the election.
Method	**Description**
calculateWinners() : void	This is an abstract method not implemented here. This is implemented in IRV and CPL to determine the winner(s) of the election.
breakTie(entities : ElectionEntity[], winners : int, isRandom : bool = true) : ElectionEntity[]	This method will randomly determine the winner or winners of a tie given the entities that are tied. The isRandom flag is used for debugging purposes, and is by default set to true.
getAudit() : ElectionAudit	This method simply returns the audit member of this class.
setAudit(auditObject : ElectionAudit) : void	Sets the audit attribute.
setNumBallots(num : int) : void	Sets the numBallots attribute.

Concrete Class: FileParser	
Attribute	**Description**
std::string fname	Name of .csv file to read into the program.
std::ifstream file	ifstream object that the FileParser uses to take in information from the .csv
Method	**Description**
getFileName() : void	Gets the file name of the input .csv.

openFile(fileName : std::string) : int	Opens fname for reading.
getMetadata() : ElectionObject	Gets metadata for the election, returning the correct election object depending on the .csv file's first line.
ballotToArray(ballot : std::string) : int[]	A helper method for IRV use only, translates the ballot formatting into a format that makes it easier to make linked lists.
readBallots(election : CPL) : void	Reads the ballots and adds the information to the CPL election.
readBallots(election : IRV) : void	Reads the ballots and adds the information to the IRV election.

Concrete Class: IRV	
Attribute	**Description**
int numCandidates	The number of candidates involved in the election.
int remainCandidates	The number of candidates that are still in the running at any given time (in other words, the number of candidates not yet eliminated).
Candidate[] candidates	An array of candidate objects; these are the candidates involved in the election.
BallotLinkedList[] ballotList	This is an array of BallotLinkedLists. Each entry in the array is a linked list that represents a ballot.
Method	**Description**
calculateWinners() : void	This method determines the winner of the IRV election.
printResults(winner : Candidate, redists: int) : void	This method prints the results of the IRV election to the screen.
getCandidates() : Candidate[]	This returns the candidates attribute.

setCandidates(candidateArray : Candidate[]) : void	Sets the candidates attribute.
setBallotList(linkedList : BallotLinkedList[]) : void	Sets the ballotList attribute.
setNumCandidates(num : int) : void	Sets the numCandidates attribute.

Concrete Class: IRVAudit	
Attribute	**Description**
None	N/A
Method	**Description**
IRVAudit(day : int, month : int, year : int)	Constructor for the IRVAudit class. Takes in the day, month, and year of the election.
writeMetadata(file : ofstream, election : ElectionObject) : void	Writes the metadata for the IRV election (number of candidates, candidate names, etc) to the audit file.
writeFinalResults(file : ofstream, winner : Candidate, percentage : float) : void	Writes the final results of the election to the audit file.
writeBallot(votes : int[], file : ofstream, candidates : Candidate[]) : void	Writes the candidates that a particular person voted for in their ranking to the audit file.
writeVoteTransfer(file : ofstream, candidates : Candidate[], running : bool[], eliminated : Candidate) : void	Writes out how votes were transferred in an IRV vote redistribution to the audit file.
writeTieLead(file : ofstream, original : Candidate[], winner : Candidate) : void	Writes to the audit file the candidates that were tied and who won when the tie is concerned with tying for the most votes.
writeTieLose(file : ofstream, original : Candidate[], winner : Candidate) : void	Writes to the audit file the candidates that were tied and who won when the tie is for least number of votes.

Concrete Class: Party

Attribute	**Description**
int numVotes	This is the number of votes for the party.
int firstSeats	This is the number of seats given to the party in the first allocation of seats.
int secondSeats	This is the number of seats given to the party in the second allocation of seats.
std::string[] candidates	This is a list of candidate names for this party.
int leftoverVotes	This is the number of votes that are leftover after the initial allocation of seats.
Method	**Description**
getFirstSeats(seats : int) : void	Returns firstSeats attribute.
getSecondSeats(seats : int) : void	Returns secondSeats attribute.
setFirstSeats(seats: int) : void	Sets the firstSeats attribute.
setSecondSeats(seats : int) : void	Sets the secondSeats attribute.
getCandidates() : std::string[]	Returns the candidates attribute.
getNumVotes() : int	Returns the numVotes attribute.
getLeftoverVotes() : int	Returns the leftoverVotes attribute.
setNumVotes(num : int) : void	Sets the numBotes attribute.
setLeftoverVotes(num : int) : void	Sets the leftoverVotes attribute.
incrementVotes() : void	Increments numVotes by 1.
Party()	Constructor that initializes the integer attributes in Party object when constructed.

5. Component Design

In this section, we take a closer look at what each component does in a more systematic way. If you gave a functional description in section 3.2, provide a summary of your algorithm for each function listed in 3.2 in procedural description language (PDL) or pseudocode. If you gave an

OO description, summarize each object member function for all the objects listed in 3.2 in PDL or pseudocode. Describe any local data when necessary.

Abstract Class: ElectionObject

Method 1: getAudit() : ElectionAudit

```
return audit
```

Method 2: breakTie(entities : ElectionEntity[], winners : int, isRandom : bool = true) : ElectionEntity[]

```
if isRandom is true:
        tallies = new array with length of entities length
        win = new array with size winners
        numActive = length of entities array
        for i = 0, i < winners, i++:
                for i = 0, i < 1000, i++
                        j = random integer from 0 to (number of entities - 1)
                        tallies[j]++
                win[i] = entities[index of max element in tallies]
                remove winning entity from entities array
                numActive = numActive - 1
                tallies = new array with size numActive
        return win
else:
        win = new array of size winners
        for i = 0, i < winners, i++
                win[i] = entities[i]
        return win
```

Method 3: setAudit(auditObject : ElectionAudit) : void

```
audit = auditObject
```

Method 4: setNumBallots(num : int) : void

```
numBallots = num
```

Concrete Class: BallotLinkedList

Method 1: removeHead() : void

```
if list.head is null:
        return
else:
        temp = pointer to head.next
```

```
            free head
            head = temp
            length- -
            return
```

Method 2: appendNode(candidate : int) : void

```
    if head is null:
            new node = new BallotNode
            head = new node
            tail = new node
            length++
            return
    else:
            new node = new BallotNode
            tail.setNext(new node)
            tail = new node
            length++
            return
```

Method 3: getHead() : int

```
    if head is null:
            return -1
    else:
            return getIndex(head)
```

Method 4: populateList(votes : int[]) : void

```
    for i = 0, i < length of votes, i++:
            if votes[i] is -1
                    break
            else
                    appendNode(votes[i])
    return
```

Method 5: cleanup() : void

```
    ptr = head
    while ptr is not null
            temp = ptr.getNext()
            free ptr
            ptr = temp
    return
```

Method 6: getLength() : int

```
    return length
```

Concrete Class: BallotNode

Method 1: getNext() : BallotNode*
return next

Method 2: getIndex() : int
return index

Method 3: setNext(newNode : BallotNode*) : void
next = newNode
return

Concrete Class Candidate:

Method 1: incWorkingVotes(amt : int = 1) : void
workingVotes += amt
return

Method 2: getWorkingVotes() : int
return workingVotes

Method 3: getIndex() : int
return index

Method 4: setIndex(new : int) : void
index = new
return

Method 5: incGainedVotes(index : int) : void
gainedVotes[index]++
return

Method 6: getInitialFirstVotes() : int
return initialFirstVotes

Method 7: incInitialFirstVotes(votes : int) : void
initialFirstVotes += votes

Method 8: getGainedVotes() : int[]
return gainedVotes

Method 9: getParty() : std::string

return party

Concrete Class IRV:

Method 1: getCandidates() : Candidate[]
return candidates

Method 2: printResults(winner : Candidate, redists : int) : void
```
print ("Candidates and parties")
print ("Original first place votes")
for i = 0, i < redists, i++
        print("Transfer 'i'")
        print ("New totals")
for i = 0, i < numCandidates, i++
        if candidates[i] == winner
                print("*")
        print(candidates[i].getName())
        print(candidates[i].getParty())
        print(candidates[i].getInitialFirstVotes())
        for i = 0, i < redists, i++
                if candidates[i].getGainedVotes[i] = -1:
                        print("-------")
                else:
                        print ("+" candidates[i].getGainedVotes[i])
                        candidates[i].incWorkingVotes(candidates[i].getGainedVotes[i])
                        print(candidates[i].getWorkingVotes)
return
```

Method 3: calculateWinners() : void
```
fileStream = openAudit()
winner = null
stillIn = numCandidates
numLoop = 0
stillRunning = new bool array, all entries true, length of numCandidates
for each ballot in ballotList:
        if ballot.getHead() is not null:
                candidates[ballot.getHead()].incWorkingVotes()
                candidates[ballot.getHead()].incIntitialFirstVotes()
while winner is not null
        if stillIn is 2:
                votes1, votes2 = 0
```

```
                idx1, idx2 = 0
                for i = 0; i < numCandidates, i++
                        if inRunning[i] and votes1 is not 0:
                                votes2 = candidates[i].getWorkingVotes()
                                idx2 = i
                        else votes1 = candidates[i].getWorkingVotes()
                        idx1 = i
                if votes1 equals votes2:
                        tieArr = new array with candidates[idx1] and candidates[idx2]
                        tieWin = breakTie(tieArr, 1)
                        audit.writeTieLead(fileStream, tieArr, tieWin[0]
                        winner = tieWin[0]
                        break
                else:
                        if votes1 > votes2:
                                winner = candidates[idx1]
                                break
                        else winner = candidates[idx2], break
        else:
                max = 0
                maxCand = null
                for i = 0, i < numCandidates, i++
                        if candidates[i].getWorkingVotes() > max
                                max = candidates[i].getWorkingVotes()
                                maxCand = candidates[i]
                totalVotes = 0
                for i = 0, i < numCandidates, i++
                        totalVotes += candidates[i].getWorkingVotes()
                if (maxCand.getWorkingVotes() / totalVotes) >= .5
                        winner = maxCand
                        break
                else:
                        losers = new Candidate vector
                        min = MAX_INT
                        minCand = null
                        for i = 0, i < numCandidates, i++
                                if candidates[i].getWorkingVotes() < min
                                        losers.clear()
                                        min = candidates[i].getWorkingVotes()
                                        minCand = candidates[i]
                                        losers.add(candidates[i])
                                else if candidates[i].getWorkingVotes() = min
                                        losers.add(candidates[i])
```

```
                    if losers.size() > 1
                        tieWinner = new empty candidate array of length 1
                        tieWinner = breakTie(losers, 1)
                        audit.writeTieLose(fileStream, losers, tieWinner[0])
                        losers.clear()
                        losers.add(tieWinner[0])
                    loser = losers[0]
                    stillRunning[loser.getIndex()] = false
                    for i = 0, i < ballotList.getLength(), i++
                        if ballotList[i].getHead() == loser.getIndex()
                            done = false
                            while !done
                                ballotList[i].removeHead()
                                if ballotList[i].getHead == null
                                else if stillRunning[ballotList[i].getHead]
                                    done = true
                                    candidates[ballotList[i].getHead].inc
                                                                WorkingVotes()
                                    candidates[ballotList[i].getHead].inc
                                                        GainedVotes(numLoop)


                    writeVoteTransfer(fileStream, candidates, stillRunning, loser)
                    numLoop++
        printResults(winner, numLoop)
        for i = 0, i < numCandidates, i++
            total += candidates[i].getWorkingVotes()
        audit.writeFinalResults(fileStream, winner, winner.getWorkingVotes() / total)
        for i = 0, i < ballotList.length(), i++
            ballotList[i].cleanup()
        close(fileStream)
        return
```

Method 4: setNumCandidates(num : int) : void

```
    numCandidates = num
```

Method 5: setCandidates(candidateArray : Candidate[]) : void

```
    candidates = candidateArray
```

Method 6: setBallotList(linkedList : BallotLinkedList) : void

```
    ballotList = linkedList
```

Concrete Class: FileParser

Method 1: getFileName() : void

```
fname = “”;
std::ifstream file
while(fname == “”)
        if arguments <= 1
                print “Not enough arguments! Normal usage is ./programName
                <filename>” to screen
        else if arguments > 2
                print “Too many arguments! Normal usage is ./programName <filename>”
                to screen
        else
                file.open(filename)
                if(file.failedToOpen())
                        print “Input file failed to open.” to screen
                        fname = “”
                file.close()
                fname = filename
```

Method 2: openFile() : int

```
std::ifstream file
file.open(fname)
if(file.failedToOpen())
        return -1
else
        return 0
```

Method 3: getMetadata() : ElectionObject

```
print “Please give the current day in dd format.” to screen
day = input()
while(day invalid)
        print “Wrong format! Please give the current day in dd format.” to screen
        day = input()
print “Please give the current month in mm format.” to screen
month = input()
while(month invalid)
        print “Wrong format! Please give the current month in mm format.” to
        screen
        month = input()
print “Please give the current year in yyyy format.” to screen
year = input()
while(year invalid)
        print “Wrong format! Please give the current year in yyyy format.” to
```

```
        screen
        year = input()
file >> stringStore
if(stringStore == “IR”)
        IRV IRElection
        IRVAudit audit(day, month, year)
        file >> stringStore
        numCandidates = toInt(stringStore)
        IRElection.setNumCandidates(numCandidates)
        Candidate candidates[numCandidates]
        for(int i = 0; i < numCandidates; i++)
                file >> stringStore
                if(stringStore[stringStore.length] == “,”)
                        stringStore = stringStore.substr(0, stringStore.length)
                candidates[i] = stringStore
        IRElection.setCandidates(candidates)
        file >> stringStore
        numBallots = toInt(stringStore)
        IRElection.setNumBallots(numBallots)
        ofstream auditFile = audit.openAudit()
        audit.writeMetadata(auditFile, IRElection)
        IRElection.setAudit(audit)
        auditFile.close()
if(stringStore == “CPL”)
        CPL CPLElection
        CPLAudit audit(day, month, year)
        CPLElection.setAudit(audit)
        file >> stringStore
        numParties = toInt(stringStore)
        CPLElection.setNumParties(numParties)
        Party parties[numParties]
        for(int i = 0; i < numParties; i++)
                file >> stringStore
                if(stringStore[stringStore.length] == “,”)
                        stringStore = stringStore.substr(0, stringStore.length)
                parties[i] = stringStore
        std::string candidates[0]
        for(int i = 0; i < numParties; i++)
                file >> stringStore
                if(stringStore[stringStore.length] == “,”)
                        stringStore = stringStore.substr(0, stringStore.length)
                candidates.push(stringStore)
        CPLElection.setParties(parties)
```

```
            file >> stringStore
            numSeats = stringStore
            CPLElection.setNumSeats(numSeats)
            file >> stringStore
            numBallots = toInt(stringStore)
            CPLElection.setNumBallots(numBallots)
            ofstream auditFile = audit.openAudit()
            audit.writeMetadata(auditFile, CPLElection)
            CPLElection.setAudit(audit)
            auditFile.close()
        return null // this should only be reached if no election type in the input file was found
```

Method 4: ballotToArray(ballot : std::string, numCandidates : int) : int[]

```
        int[numCandidates] initialArray = {-1}
        int currentCandidate = 0
        int i
        while(i  < ballot.length)
            if(ballot[i] == “,”)
                currentCandidate++
            else
                initialArray[currentCandidate] += toInt(ballot[i])
            i++
        int[numCandidates] translatedArray = {-1}
        for(int i = 0; i < numCandidates; i++)
            if(initialArray[i] != -1)
                translatedArray[initialArray [i]] = i
        return translatedArray
```

Method 5: readBallots(election : CPL) : void

```
        numBallots = 0
        parties = election.getParties()
        audit = election.getAudit()
        while(file >> stringStore)
            int i = 0
            currentParty = 0
            while(i  < parties.length)
                if(stringStore[i] == “1”)
                    parties[currentParty].incrementVotes()
                    audit.writeBallot(parties[currentParty])
                    break()
                else
                    currentParty++
```

```
                    i++
            numBallots++
        election.setNumBallots(numBallots)
        file.close()
```

Method 6: readBallots(election : IRV) : void

```
        BallotLinkedList[] listOfLinkedLists
        election.setBallotList(listOfLinkedLists)
        audit = election.getAudit()
        while(file >> stringStore)
                ballotArray = ballotToArray(stringStore)
                audit.writeBallot(ballotArray)
                election.populateList(ballotArray)
```

Concrete Class: CPL

Method 1: calculateWinners() : void

```
        bool[numParties] allowed = {true}
        auditFile = audit.openAudit()
        audit.writeInitialVotes(auditFile, parties)
        totalVotes = 0
        for(int i = 0; i < numParties; i++)
                totalVotes += parties[i].getNumVotes()
        quota = totalVotes / numSeats  // make sure this is integer division
        auditFile << "Quota is:" << quota << "\n"
        numCandidates = parties[i].getCandidates().length
        for(int i = 0; i < numParties; i++)
                int seats = parties[i].getNumVotes() / quota // make sure this is integer division
                parties[i].setLeftoverVotes(parties[i].getNumVotes() % quota)
                if (seats > numCandidates )
                        parties[i].getCandidates.setFirstSeats(numCandidates)
                        allowed[i] = false
                        seatLottery(allowed, parties[i], seats-numCandidates, auditFile)
                else
                        parties[i].getParties.setFirstSeats(seats)
                numSeats -= seats
                audit.writeFirstAllocation(auditFile, parties, quota)
        if(numSeats > 0)
                int[numParties] remainingVotes = {0}
                for(int i = 0; i < numParties; i++)
                        remainingVotes[i] = parties[i].getLeftoverVotes()
```

```
            Party[0] needsTiebreaking
            while(numSeats > 0)
                int maxRemain = 0
                int index = 0
                for(int i = 0; i < numParties; i++)
                    if(allowed[i]
                        if(remainingVotes[i] > maxRemain)
                            maxRemain = remainingVotes[i]
                            index = i
                        else if(remainingVotes[i] == maxRemain)
                            needsTiebreaking.push(parties[i])
                needsTiebreaking.push(parties[index])
                if(needsTiebreaking.length > 1)
                    Party[] winners = breakTie(needsTiebreaking, numSeats)
                    audit.writeTiebreaker(auditFile, needsTiebreaking, winners)
                    for(int i = 0; i < winners .length; i++)
                        numSeats--
                        winners[i].setSecondSeats(1)
                else
                    numSeats--
                    parties[index].setSecondSeats(1)
            audit.writeSecondAllocation(auditFile, parties)
        audit.writeFinalResults(auditFile, parties, quota)
```

Method 2: getParties() : void

```
    return this.parties
```

Method 3: printResults() : void

```
    print("Parties")
    for(int i = 0; i < numParties; i++)
        print("Candidates")
        for(int j =0; j < parties[i].getCandidates.length; j++)
            print(parties[i].getCandidates[j])
        print("Votes")
        print(party.getNumVotes)
        print("First Allocation of Seats")
        print(party.getFirstSeats)
        print("Remaining Votes")
        print(party.getLeftoverVotes)
        print("Second Allocation of Seats")
        print(party.getSecondSeats)
        print("Final Seat Total")
```

```
        int total = party.getFirstSeats+party.getSecondSeats
        print(total)
```

Method 4: seatLottery(allowed : bool[], source : std::string, numSeats : int, file : ofstream) : void

```
    random randomGen
    while(numSeats > 0)
        num = randomGen.randInt() % allowed.length
        if(allowed[num])
            parties[num].setFirstSeats(parties[num].getFirstSeats()+1)
            numSeats--
            file.writeLottery(file, source, parties[num])
```

Method 5: setNumParties(num : int) : void

```
    numParties = num
```

Method 6: setParties(partyArray : Party[]) : void

```
    parties = partyArray
```

Method 7: setSeats(num : int) : void

```
    numSeats = num
```

Concrete Class: IRVAudit

Method 1: writeMetadata(file : std::ofstream, election : ElectionObject) : void

```
    file << election << endl;
    file.close();
```

Method 2: writeFinalResults(file : ofstream, winner : Candidate, percentage : float) : void

```
    file << "The final results have " << winner <<" as the winning candidate with a vote majority of " << percentage << endl;
    file.close();
```

Method 3: writeBallot(votes : int[], file : std::ofstream, candidates : Candidates[])

```
    for(int i =0; i< votes.length(); i++)
        file << "Voted for " << candidates[votes[i]] << " as their " << i+1 << "th choice." << endl
    file.close()
```

Method 4: writeVoteTransfer(file : ofstream, candidates : Candidate[], running bool[], distributed int[], eliminated : Candidate) : void

```
    for(int i = 0; i < candidates.length(); i++)
        if running[i] = false
            file << eliminated << " was eliminated and their votes distributed " << endl
```

```
        for(int i = 0; i < distributed.length(); i++)
                file << candidates[i] << " had " << distributed[i] << " votes distributed to them "
<< endl;
```

Method 5: writeTieLead(file : ofstream, original : Candidate[], winner : Candidate) : void

```
        for (int i = 0; i < original.length(); i++)
                if winner == original[i]
                        file << winner << " won in a tie breaker for winning the election against "
<< rest of original;
```

Method 6: writeTieLose(file : ofstream, original : Candidate[], winner : Candidate) : void

```
        for (int i = 0; i < original.length(); i++)
                if winner == original[i]
                        file << winner << " won in a tie breaker for lowest votes against " << rest
of original;
```

Method 7: IRVAudit(day : int, month : int, year : int)

```
        fname = "IRVAudit_" + day + "/" + month + "/" + year + ".txt
        ofstream(fname);
```

Abstract Class: ElectionAudit

Method 1: openAudit() : std::ofstream

```
        stream = open(fname)
        return stream
```

Concrete Class: CPLAudit

Method 1: CPLAudit(day : int, month : int, year : int)

```
        fname = "CPLAudit_" + day + "/" + month + "/" + year + ".txt
        ofstream(fname);
```

Method 2: writeMetadata(file : ofstream, object : ElectionObject) : void

```
        file << object.getMetadata();
```

Method 3: writeInitialVotes(file : ofstream, parties : Party[]) : void

```
        for(int i=0; i< parties.length(); i++)
                file << parties[i].getVotes() << " had " << parties[i].getVotes() << " total votes"
<< endl;
```

Method 4: writeFinalResults(file : ofstream, parties : Party[]) : void

```
        for i = 0, i < numParties, i++
                write(ofstream, parties[i].getName())
```

```
seats = parties[i].getFirstSeats() + parties[i].getSecondSeats()
write(ofstream, “seats:” seats)
write(ofstream, “winners:”)
for i = 0, i < seats, i++
        write(ofstream, candidates[i])
write(ofstream, newline)
return
```

Method 5: writeFirstAllocation(file : ofstream, parties : Party[], quota : int) : void

```
write(ofstream, “quota is: “ quota)
for i = 0, i < numParties, i++
        write(ofstream, parties[i].getName())
        write(ofstream, “received” parties[i].getNumVotes “votes”)
        write(ofstream, “received” parties[i].getFirstSeats “seats”)
        return
```

Method 6: writeSecondAllocation(file : ofstream, parties : Party[]) : void

```
for i = 0, i < numParties, i++
        write(ofstream, parties[i].getName())
        write(ofstream, “had” parties[i].getleftoverVotes “remainder votes”)
        write(ofstream, “received” parties[i].getSecondSeats “more seats”)
        return
```

Method 7: writeLottery(file : ofstream, source : Party, destination : Party) : void

```
file << “Seat taken lotteried “ << source.getName() << “and given to ” <<
destination.getName()
```

Method 8: writeBallot(party : Party) : void

```
file << “Ballot for “ << party.getName()
```

Method 9: writeTiebreaker(file : ofstream, original : Party[], winners : Party[]) : void

```
file << “Seat lottery chose the following winners: ”
for(int i = 0; i < winners.length; i++)
        if(i != winners.length-1)
                file << winners[i] << “, “
file << “selected from the following parties: “
for(int i = 0; i < original.length; i++)
        if(i != original.length-1)
                file << original[i] << “, “
```

Abstract Class: ElectionEntity

Method 1: getName() : std::string
return name

Method 2: setName(newName : std::string)
name = newName

Concrete Class: Party

Method 1: incrementVotes() : void
numVotes++

Method 2: setFirstSeats(seats : int) : void
firstSeats = seats

Method 3: setSecondSeats(seats : int) : void
secondSeats = seats

Method 4: setNumVotes(num : int) : void
numVotes = num

Method 5: setLeftoverVotes(num : int) : void
leftoverVotes = num

Method 6: getFirstSeats() : int
return firstSeats

Method 7: getSecondSeats() : int
return secondSeats

Method 8: getCandidates() : std::string[]
return candidates

Method 9: getNumvotes() : int
return numVotes

Method 10: getLeftoverVotes() : int
return leftoverVotes

Method 11: Party()
numVotes = 0
firstSeats = 0
secondSeats = 0
leftoverVotes = 0

6. HUMAN INTERFACE DESIGN

6.1 Overview of User Interface

This voting system will be developed in such a way that it will enhance efficiency, intuitiveness, and most notably, user-friendliness. The system will be guiding its users throughout the overall process involved in inputting the required data, running elections, and analyzing or generating results. It is worth noting that the system users here will also be allowed to see election results and choose what must be shared with notable stakeholders such as the media personnel. It is worth noting that the system here is run in the terminal. The user is interacting with this system via the CLI (command line interface within the terminal). The user is first prompted for the input csv file name (if a file name was not provided on the command line). The user will be repromted if the file name is invalid. Next, the user will be prompted to enter in the day, month, and year of the election separately. After these inputs, the program will run and it will display the major results of the election to the terminal screen so that the user can see them and share them with the media if necessary. They will also have access to the audit file, which will have been created in the same directory as the input file.

6.2 Screen Images

Recall that this system is run entirely in the terminal and is text-based. Therefore, there is no user interface being developed for this program. Below we can see what a potential run of the program would look like from the user's perspective, with the box being the terminal window. Note that system output is colored black, while user input is red.

```
> ./voting_system
> Please enter the name of the election file.
exampl_irv.csv
> File was not found. Please give the name of the election file.
example_irv.csv
> Please enter the day of the election.
10
> Please enter the month of the election.
11
> Please enter the year of the election.
2023
> Candidates                First Place Votes
  *Jacob McIsaac                     50,000
   Ruolei Zeng                          200
   Caleb Otto                           100
   Daniel Kong                           10
```

Please note that this is only one possibility for IRV terminal result output. The result table may be much larger, depending on if there were any vote redistributions involved. The format of this table was outlined in depth in the SRS document, and psuedocode can be found in Section 5 of this document. Nevertheless, the general flow of the program in the terminal is the same.

6.3 Screen Objects and Actions

As previously mentioned, this program is entirely text-based and will be running in the terminal. Any required input to the program will be prompted for in the terminal and inputted by the user. The user will run the program in the terminal and respond to prompts when necessary. The results of the election will then be displayed in the terminal screen. Therefore, the only "screen object" that the user will be interacting with will be the terminal. There are no other screen objects in this program.

7. Requirements Matrix

The table below shows how each of the use cases specified in the SRS document are satisfied with this system design. For a full description of each use case, see the SRS document and use

cases document which can be found in the SRS directory. Please note as well that use case 4.3 has changed from the SRS. This is because the previous understanding of the required functionality was incorrect. This use case is now simply required to ask the user for the date that the election took place.

Use Case	**How it is Satisfied**
4.1: Prompting User for Election File Name	This functionality is satisfied in the getFileName() function in the FileParser class. This method will prompt the user for a CSV file name. It will also check to ensure that the file type is correct; if it is not, it will continue prompting the user until it gets a correct file name.
4.2: Getting Election Filename on Command Line	This functionality is also satisfied by the getFileName() function in the FileParser class. This function will check the command line arguments to the program. Specifically, it will check to see if the first non-program name command line argument is a valid CSV file name. If it is not, it will move along with use case 4.1.
4.3: Prompt User For Election Date	This functionality is satisfied in the getMetadata() function in the FileParser class. The user will be prompted for the day, month, and year of the election and this information will be used to set the fields of the ElectionAudit object in the newly instantiated ElectionObject object.
4.4: Opening the Election File	This functionality is handled in the openFile() function in the FileParser class. This function will attempt to open the file with the given file name. If it is not able to open the file, it will be called again with a new file name that will be obtained from an additional call to getFileName().
4.5: Parse File for Election Type	This functionality is handled in the getMetadata() function in the FileParser class. The program will have the information

	extracted from input files, the type of election be stored in memory, and then continue on to read the rest of the metadata based on this type.
4.6: Parse File for IRV Information	This functionality is handled in the getMetadata() function in the FileParser class. The program will have the IRV information extracted from input files, and have them utilized to carry out Instant-runoff Voting elections.
4.7: Parse File for CPL Information	This functionality is handled in the getMetadata() function in the FileParser class. The program will have the CPL information extracted from input files, and have them utilized to carry out Instant-runoff Voting elections.
4.8: Read in Ballot Data	This functionality is handled by the readBallots() function in the FileParser class. This function will read in the ballots from the input file and transform them into a form that is usable by the system. This process will of course depend on the type of election being processed.
4.9: Determine Winner in IRV Election	This functionality is handled by the calculateWinners() function in the IRV class. It will compare all of the vote counts out of those candidates remaining and then determine if one has a majority and the winner can be determined or if another round must commence.
4.10: Determine Winner When No Candidate Has Majority at End of IRV	This functionality is handled by the calculateWinners() function in the IRV class. It will check if there is no clear majority between two candidates in the IRV election and then compare their popularity votes if there isn't a clear winner.
4.11: Determine Winner(s) in CPL Election	This functionality is handled by the calculateWinners() function in the CPL class.

	It will compare the amounts of votes for different parties in order to determine how many seats each party has earned. It will also determine who those seats go to within the party by order of candidates per party.
4.12: Distribute Seats When Party Wins More Seats than They Have Candidates	This functionality is handled by the seatLottery() function in the CPL class. It will check how many extra seats a party has earned beyond their number of candidates and then perform a random raffle to determine which of the remaining parties the seats go to.
4.13: Break a Tie that has Occurred	This functionality is handled by the breakTie() function in the ElectionObject class. It will be used in both IRV and CPL elections to break any tie that has occurred by simulating a random coin toss as well as to assist in fair random judgment for functions such as the lottery.
4.14: Create/Open Audit File for Writing	This functionality is handled by the ElectionAudit class, specifically the openAudit() function, as it creates an audit file with the given metadata if one is not found, and opens the file for writing.
4.15: Populate Audit File	This functionality is handled by the concrete child classes of ElectionAudit (I.E. CPLAudit and IRVAudit), which allow for writing to the audit after the audit file has been opened.
4.16: Print IRV Results to Terminal	This functionality will be handled by the printResults() function in the IRV class. This function will print the appropriate information for IRV to the terminal once the winner has been determined.
4.17: Print CPL Results to Terminal	This functionality will be handled by the printResults() function in the CPL class. This function will print the appropriate information for CPL to the terminal once the winners have been determined.

8. APPENDICES

There are no necessary appendices for this document.