

Team 18  
**Voting System**  
Software Design Document

Name (s): Noreen Si,  
Jonathan Haak, Pyrenees  
Gavois, Lane Enget  
Section: 001

Date: Mar 2, 2023

## TABLE OF CONTENTS

<b>1. INTRODUCTION</b>	<b>2</b>
1.1 Purpose	2
1.2 Scope	2
1.3 Overview	2
1.4 Reference Material	2
1.5 Definitions and Acronyms	3
<b>2. SYSTEM OVERVIEW</b>	<b>3</b>
<b>3. SYSTEM ARCHITECTURE</b>	<b>4</b>
3.1 Architectural Design	4
3.2 Decomposition Description	5
3.3 Design Rationale	10
<b>4. DATA DESIGN</b>	<b>11</b>
4.1 Data Description	11
4.2 Data Dictionary	11
<b>5. COMPONENT DESIGN</b>	<b>15</b>
<b>6. HUMAN INTERFACE DESIGN</b>	<b>22</b>
6.1 Overview of User Interface	22
6.2 Screen Images	23
6.3 Screen Objects and Actions	24
<b>7. REQUIREMENTS MATRIX</b>	<b>25</b>
<b>8. APPENDICES</b>	<b>26</b>

# 1. INTRODUCTION

## 1.1 Purpose

This Software Design Document provides the details for the Voting System project from CSCI 5801. The audience of this document are the four developers who will be completing it (Noreen, Jon, Lane, Pyrenees) from the University of Minnesota and the professor and graders of this project.

## 1.2 Scope

This document contains a complete description of the voting system. This system will be created using Java and will be primarily accessed via the terminal. The goal of the system is to calculate the winners of different types of elections given a file with all the data of all the ballots and other information. This benefits the users as the process of actually counting and distributing votes among candidates is automated, saving a lot of time and preventing potential human errors.

## 1.3 Overview

- Section 2 gives an overall review of what the system will do, the context it will operate under, and the overall design.
- Section 3 describes a detailed design of the system and how all the components work with each other.
- Section 4 describes how collected data will be stored and processed. It elaborates on the data structures used throughout the system and how they work.
- Section 5 takes a look at each component in the system and elaborates on them one at a time. Unlike section 3, this does not focus on how they interact with each other, but rather what they do on their own.
- Section 6 describes how the user interface will work and how to operate the system.
- Section 7 shows a requirements matrix that traces the requirements to how they will be addressed.

## 1.4 Reference Material

The [SDD Document](#) on Canvas was consulted.

The [SRS Document](#) was also consulted for guidance.

## 1.5 Definitions and Acronyms

Term	Definition
IR Voting/IR/IRV	A system of voting in which voters rank candidates on a ballot by preference. In this system, if a candidate has a majority of the votes, the candidate wins the election; if not, the candidate with the least number of votes is eliminated, and their votes are added to the next choice indicated by each voter. This repeats until there is a majority, or – if all votes have been handed out and there’s still a tie – the winner is declared through popularity.
Closed Party Voting/CP Voting/CPL	Voters choose a party and – based on the total votes that each party receives and the total seats available in a given election – allocates seats among parties.
Audit File	A file containing all information necessary to verify the election process and results.
Voting Official	An official who oversees an election and is involved with calculating the results of an election.

## 2. SYSTEM OVERVIEW

The system will be able to perform both closed party elections and instant-runoff elections. For both these elections, “edge cases” or special scenarios will be handled, such as when there are ties. The system will be operated via the terminal by voting officials, and it is assumed that a properly formatted voting .txt file is provided, one containing all the data of ballots and candidates and parties. The system will have a tree/graph structure to store data for IR elections and a double array for CPL elections. There will be many classes to accommodate these data structures, including node and tree objects as well as some other classes to help around with other functionality, such as calculating winners, breaking ties, and writing to the audit files.

## **3. SYSTEM ARCHITECTURE**

### **3.1 Architectural Design**

The voting system contains 3 primary classes: IR, CPL, and RunElection. The IR and CPL classes are built off of an abstract 'Election' parent class and contain field members and methods necessary for implementation of IR and CPL elections, respectively.

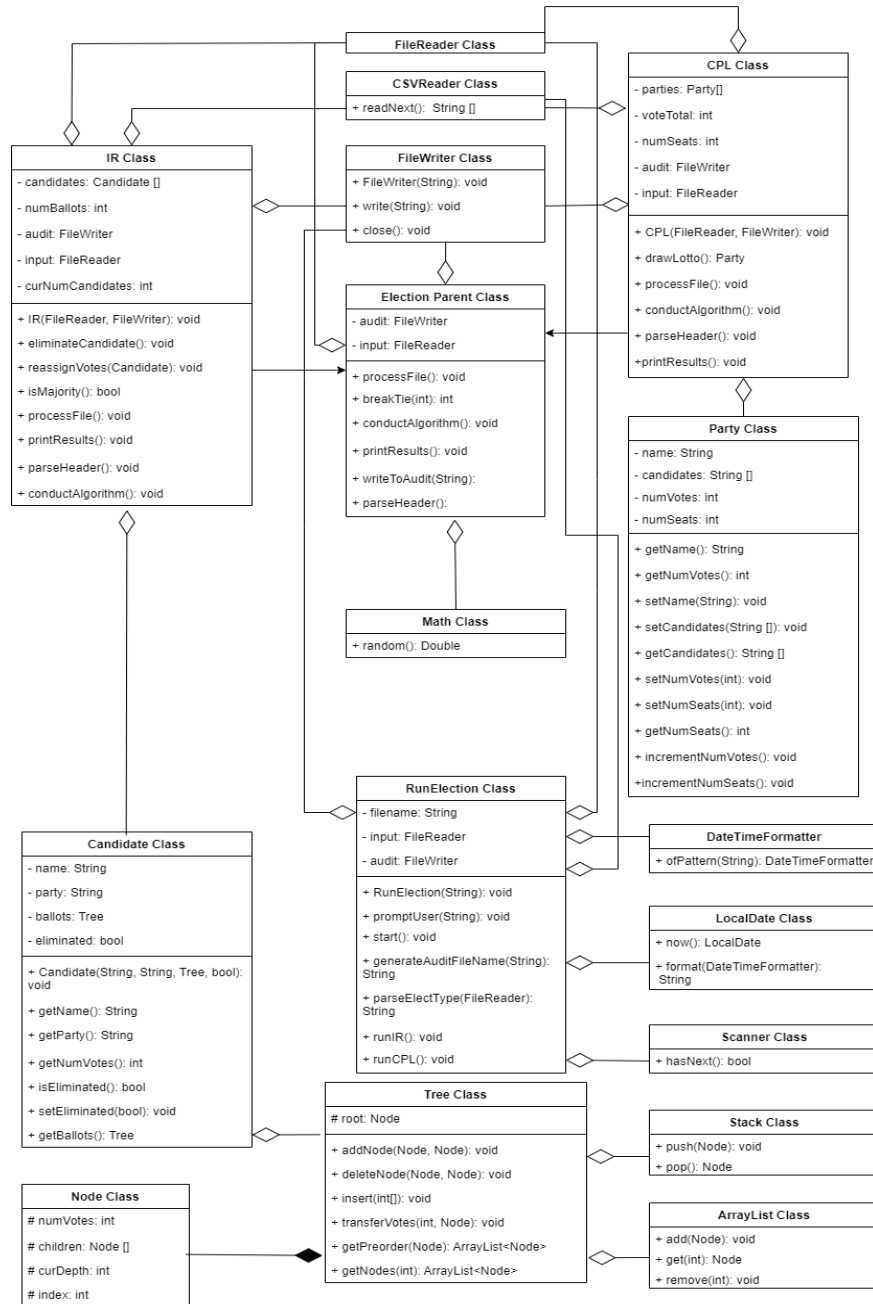
The RunElection functions as a facade, simplifying the interactions that the Main method in the Main class has with the rest of the program. The RunElection class is responsible for orchestrating every method call and object instantiation necessary to meet the use cases defined in the software requirements document.

The only function that the Main interaction has is to initialize a RunElection object, and to call the start() method on that object.

The RunElection class then steps through each process of the election, getting the input, opening input file, determining election type, creating an object for IR or CPL, and calling appropriate methods within the IR and CPL classes to orchestrate the fulfillment of all functional requirements. When complete, the Main method inside the main program resumes control, and the program is exited.

### 3.2 Decomposition Description

### 3.2.1 UML Class Diagram



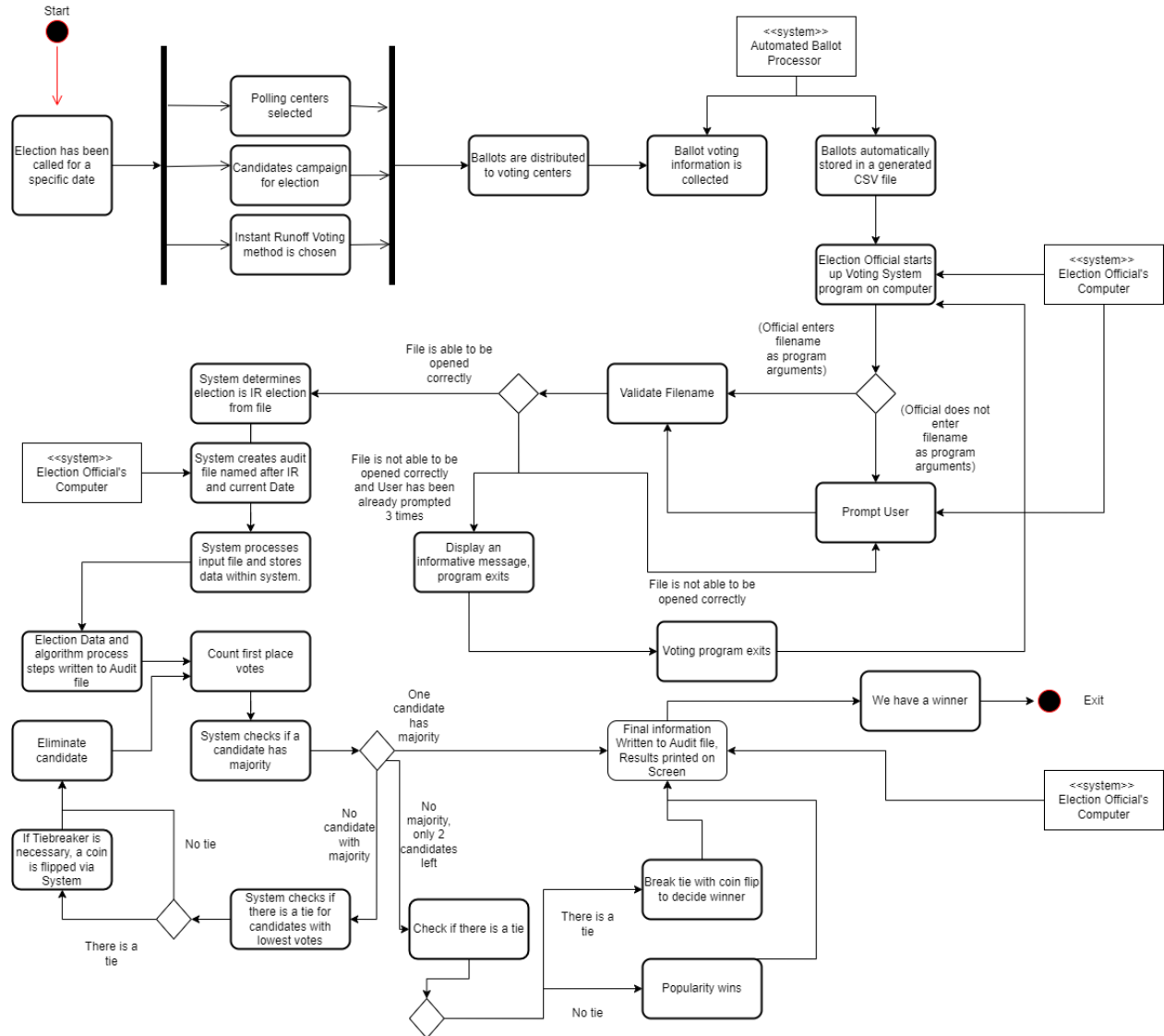
This UML Class Diagram shows the associations between various classes and the details about each class, such as the attributes and operations. Other classes, such as Stack, ArrayList, and Scanner, are included as these Java libraries are integral to the implementation of this system. The methods we intend to use and any other related information are included

as a part of the library class' descriptions. To show where these classes are used, there are has-a relationships depicted in the diagram. For example, the IR Class has a FileWriter.

One notable point is how the IR and CPL classes inherit from the abstract Election class, represented by the arrows that connect the IR and CPL classes with their parent. This shows how both IR and CPL share many similarities, such as how these classes both take in the election input file and the audit file. They also perform similar functions such as processing the file, conducting the algorithm, parsing the header, etc. Some methods, like `writeToAudit()` and `breakTie()` are implemented in the Election class, while others remain abstract. This is so the IR and CPL classes can implement their own implementations for these methods.

One other feature of this diagram is that it shows compositional relationships. For instance, a Tree contains Nodes. This is represented by a pointer with a black diamond to symbolize this relationship.

## 3.2.2 UML Activity Diagram, IR



The diagram above is an activity diagram for the IR Election. Notably, this diagram proceeds through the major steps of what it takes for an IR Election to take place and briefly touches upon associated logic for determining a winner. The process begins when an election is called. From there, there is a fork along with synchronization bars as the selection of polling centers, choosing IR as the voting method of choice, and having candidates campaign for the election do not occur one after the other or necessarily in order. From there, the process is straightforward; the automated ballot processor system can automatically record the ballot slips and convert the information into a CSV format file. The election official can then interact with

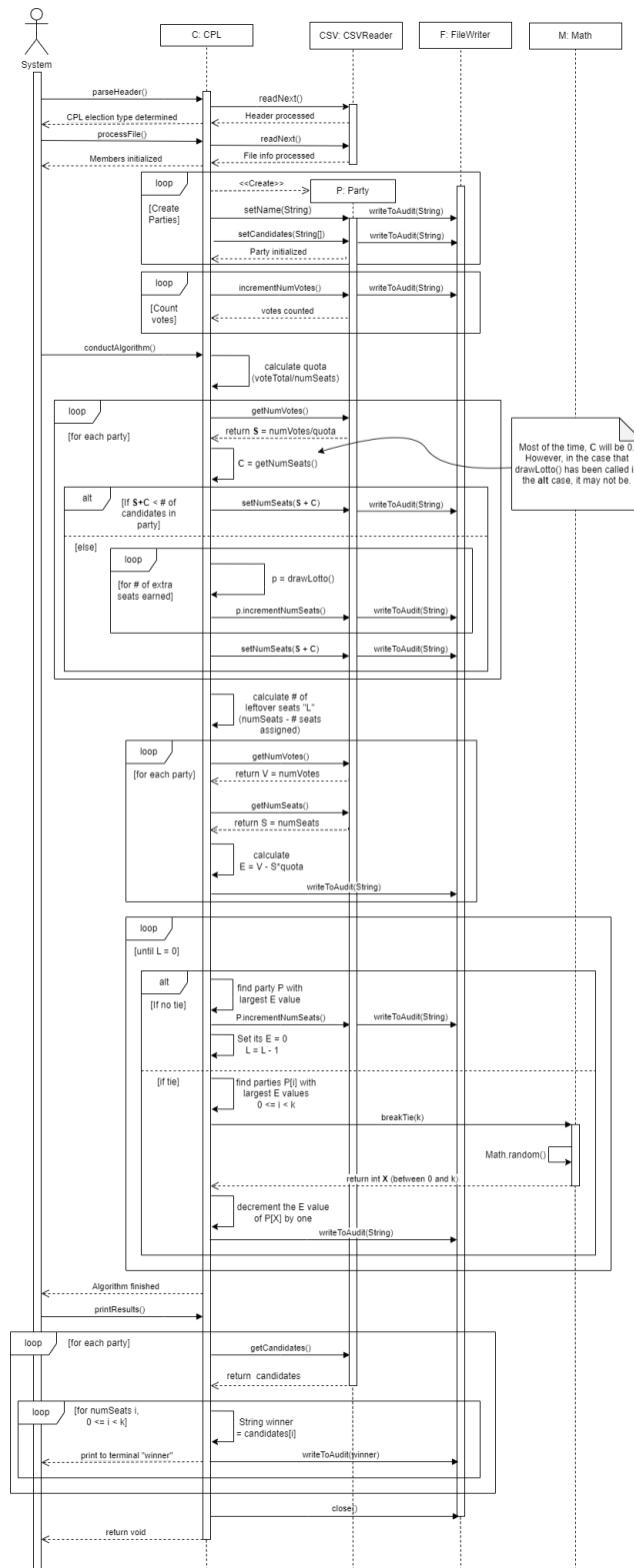


the voting program to start the program in order to eventually determine a winner. The election official's computer is a system that interacts with the IR Election process.

There are several different decisions that can be made in the course of determining the winner for the IR election. These decisions are represented by diamond symbols within the diagram. For instance, if there is a tie, then a tiebreaker shall be entered; otherwise, the program can proceed as normal.

The IR Process concludes when a winner is decided, where the process can safely exit.

## 3.2.3 Sequence Diagram, CPL



The diagram above is a sequence diagram for the CPL Election. This diagram proceeds through the logic starting from when the file is first being read to when the election process is complete and results have been fully calculated.

The process starts when the system calls `parseHeader()` from the controller. This essentially sets up the CPL object so that when `processFile()` is called by the system, the CPL object knows and is ready to read in and store all the data from the file. It will create the parties, initialize them, and tally up the votes for each party.

After this, `conductAlgorithm()` is called. For the first phase of this algorithm, it will allocate the amount of seats each party gets according to the quota, as well as handling the `drawLotto()` case (when a party earns more seats than it has candidates)

In the second phase of this algorithm, we distribute the remaining seats that have not been filled yet to the parties with the most left over votes (those that did not count towards getting seats in the first phase). This is when we handle the `breakTie()` case – when there is a tie among parties and a winner must be decided by coin flip(s).

When the algorithm is finished, each Party should have their respective `numSeats` value updated fully and `printResults()` is called by the system. The candidates of each party are printed based on the `numSeats` they have.

Throughout the process, we are constantly calling `writeToAudit()` (from the `FileWriter` class) to keep track of every step of the process, from both the Party objects and the CPL object.

### 3.3 Design Rationale

The system design is intended to optimize modularity and ease of extension while still meeting all functional requirements outlined in the SRS document. The current design allows for minimal code refactoring to add increased election type handling functionality and to add functionality for the system to handle multiple elections at the same time.

The design pattern also allows threading to be very easily implemented, if such a decision were to be made down the road regarding the processing of multiple elections.

A possible weakness with the current design is the high coupling involving the `RunElection` class, as it accesses both the `IR` and `CPL` classes. This issue can be addressed by extensive testing of the `RunElection` class and its components to ensure its functionality.

Data only enters the system in specifically defined points - program start, getting file name from user, and reading file information - and this reduces the risk of program failure caused by any single component of the system, as extensive error checking can be done whenever the program receives input. Which minimizes the possibility of errant data input into critical system components.

## **4. DATA DESIGN**

### **4.1 Data Description**

Information is read into the system from a file stored in .csv format, and it is then stored within the system where operations can be performed on the data. When party and candidate data is read into the program for an IR, it is used to initialize Candidate objects. For CPL elections, Party objects are instantiated instead. Candidate objects contain the name, and party of a candidate stored as strings. The ballots tied to that candidate, stored in a Tree data object, and whether the candidate is eliminated or not, represented by a bool. The Candidate class also provides several methods to assist with inputting data into the Candidate object, and retrieving that Data.

Party objects contain a string specifying the name of the party, a string array containing all candidates within the party, and integers specifying the number of votes the party has received, and the number of seats they have been assigned. The Party class also contains several helper methods to assist in data input and retrieval.

For IR elections, the Tree expands proportional to the number of candidates in the election. The first node will contain first place votes for the candidate, and the next level of nodes will have a node for each other candidate in the election, and hold the amount of 2nd place votes for each other candidate where the 1st place vote was for candidate who owns the tree.

This process continues for each level, but with 2nd place votes replacing 1st place votes, and so on until there is a level in the tree for every single rank of vote in the election. This data structure is needed to store all information necessary to successfully transfer votes when candidates are eliminated in IR elections.

The total number of nodes in the tree is proportional to the number of candidates in the election but storing the ballot information in this way allows for faster retrieval, significantly improving the efficiency of the algorithm.

### **4.2 Data Dictionary**

Candidate Object - Data object to store candidate information, contains a String identifying the Candidate's party, a String identifying the Candidate's name, and a Tree containing vote information necessary to the candidate.

CPL Object - Object that stores methods and variables necessary to run a CPL election.

#### Variables

- Private Party[] parties
- Private int voteTotal
- Private int numSeats
- Private FileWriter audit
- Private FileReader input

#### Methods

- Public CPL(FileReader, FileWriter) - void
- Public drawLotto(): Party
- Public processFile() - void
- Public conductAlgorithm() - void
- Public parseHeader() - void
- Public printResults() - void

IR Object - Object that stores methods and variables necessary to run an IR election

#### Variables

- Private Candidate [] candidates
- Private int numBallots
- Private File audit
- Private File input
- Private int curNumCandidates

#### Methods

- Public IR(FileReader, FileWriter) - void
- Public eliminateCandidate() - void
- Public reassignVotes(Candidate) - void
- Public isMajority() - bool
- Public processFile() - void
- Public printResults() - void
- Public parseHeader() - void
- Public conductAlgorithm() - void

Node Object - Data object to store a vote total, a depth, an index, and an array of children nodes.

#### Variables

- Protected int numVotes
- Protected Node [] children
- Protected int curDepth
- Protected int index

Tree Object - Data Object to store a collection of linked nodes, along with methods to manipulate and search the tree.

#### Variables

- Protected Node Root

#### Methods

- Public addNode(Node,Node) - void
- Public deleteNode(Node,Node) - void
- Public insert(int[]) - void
- Public transferVotes(int, Node) - void
- Public getPreorder(Node) - ArrayList<Node>

- `getNodes(int)` - `ArrayList<Node>`

Party Object - Data Object is used to store Party information

#### Variables

- Private String name
- Private String [] candidates
- Private int numVotes
- Private int numSeats

#### Methods

- Public `getName()` - String
- Public `getNumVotes()`: int
- Public `setName(String)`: void
- Public `setCandidates(String [])` - void
- Public `getCandidates()` - String []
- Public `setNumVotes(int)` - void
- Public `setNumSeats(int)` - void
- Public `getNumSeats()` - int
- Public `incrementNumVotes()` - void
- Public `incrementNumSeats()` - void

RunElection Object - Object is instantiated to conduct election

#### Variables

- Private string filename
- private FileReader input
- private FileWriter audit

## Methods

- Public RunElection(String) - void
- Public promptUser(String) - void
- Public start() - void
- Public generateAuditFileName(String) - String
- Public parseElectType(FileReader) - String
- Public runIR() - void
- Public runCPL() - void

## 5. COMPONENT DESIGN

### Election Class (Abstract)

#### Fields:

- audit: A FileWriter object that should represent the audit file.
- input: A FileReader object that should represent the input election file.

#### Methods:

- processFile(): An abstract method that is meant to represent processing the election file and storing the information in data structures. The implementation for this method in the IR and CPL classes should involve using OpenCSV to parse and store information into data structures.
- breakTie(int): Takes in an int that represents a number from 0 to N - 1, inclusive, where N is the number of candidates or parties tied. Breaks ties until one number from 0 to the integer passed in is fairly chosen through a series of unbiased coin flips. This method uses Math.random() to simulate unbiased coin flips.
- parseHeader(): An abstract method that uses the election input file and the audit file. This method is meant to parse the header of the election input file and return an array of objects with information such as name and party. The information is then recorded in the audit file through a call to writeToAudit().
- writeToAudit(String): This method writes the String passed in to the audit file in the audit file field.
- printResults(): An abstract method that is meant to represent the code that gets key information to display to the screen for the IR Class and the CPL Class.
- conductAlgorithm(): An abstract method that is meant to represent the code that runs an election to completion in the IR Class and CPL Class.

### RunElection Class



**Fields:**

- filename: A String that represents the filename. If we determined in main that the name of the file was passed in with the start of the program, then this field holds the file name, regardless of if it was valid or not. If the name of the file was not passed in with the start of the program, then this field is simply empty.
- input: A FileReader object that represents the input file.
- audit: A FileWriter object that represents the audit file.

**Methods:**

- RunElection(String): A constructor for RunElection. If the name of the file was directly passed in with the start of the program, then initializes the filename field to this name.
- start(): This method is used for the initial setup of the program. When main calls start on a RunElection object, start will first call promptUser() to obtain a valid file name only if the file name was not passed in the terminal. The prompt passed into promptUser() should ask the user to enter the name of the election file. Repeat this process until a valid file name of a file that can be opened is obtained or the user has been prompted three times. If a valid file name is able to be obtained, then a FileReader object is created. The FileReader object is then passed into parseElectType, which determines what type of election it is by returning either "IR" or "CPL". Depending on the type of election, either runIR() or runCPL() is called.
- parseElectType(FileReader): Takes in the input file and returns a string (either "IR" or "CPL") that represents the type of the election.
- promptUser(String): Takes in a prompt that represents the prompt that will be displayed to the user on the screen. Returns the user's input by using a Scanner object to parse user input.
- generateAuditFileName(String): Takes in a String that represents the type of election ( either "IR" or "CPL"). Gets the current date using DateTimeFormatter and LocalDate. Generates an audit file name formatted "MM\_DD\_YYYY\_VOTINGTYPE.txt" where VOTINGTYPE is "IR" or "CPL" respectively. Returns this String.
- runIR(): When an IR election needs to be called, first creates a FileWriter object that represents the audit file by calling upon generateAuditFile("IR"). Creates an IR object with the FileReader object input and the new FileWriter audit File. Calls parseHeader() and processFile() on the IR object. Lastly, calls conductAlgorithm() on the IR object.
- runCPL(): When a CPL election needs to be called, first creates a FileWriter object that represents the audit file by calling upon generateAuditFile("CPL"). Creates a CPL object with the FileReader object input and the new FileWriter audit File.. Calls parseHeader() and processFile() on the IR object. Lastly, calls conductAlgorithm() on the CPL object.

### **Math Class**

Methods:

- `random()`: Returns a random double greater than or equal to 0.0 and less than 1.0.

### **CSVReader Class**

Methods:

- `readNext()`: Reads the next line for the CSVReader and returns a String [].

### **Scanner Class**

Methods:

- `hasNext()`: Returns True if there is another token in the input for the Scanner.

### **FileWriter Class**

Methods:

- `FileWriter(String)`: This is the constructor for the FileWriter Class. It takes the name of a file to open.
- `write(String)`: Takes in a String to write to the file.
- `close()`: Closes the file.

### **LocalDate Class**

Methods:

- `now()`: Returns an instance of LocalDate that represents the current date.
- `format(DateTimeFormatter)`: Returns a String of the date formatted in the specified format.

### **DateTimeFormatter Class**

Methods:

- `ofPattern(String)`: Formats a date according to the format of the String passed in. Returns a DateTimeFormatter.

### **Stack Class**

Methods:

- `push(Node)`: Pushes a Node to the top of the Stack.
- `pop()`: Removes the Node item at the top of the Stack and returns this Node.

### **ArrayList Class**

Methods:

- add(Node): Adds a Node to the ArrayList.
- get(int): Retrieves the element at the index specified by the int passed in and returns this element.
- remove(int): Removes the element at the index specified by the int passed in from the ArrayList.

## Candidate Class

Fields:

- name: A string that represents the name of the candidate.
- party: A string that represents the name of the party the candidate belongs to.
- ballots: A tree structure that houses the ballots where the candidate was ranked first.
- eliminated: A bool that is True if the candidate has been eliminated and False otherwise.

Methods:

- Candidate(String, String, Tree, bool): The constructor for the Candidate. Takes in the name, party, ballots (initially an empty Tree), and a bool (initially false) for the eliminated field.
- getName(): Returns the name field.
- getParty(): Returns the party field.
- getNumVotes(): Returns the number of first-place votes by accessing the number of votes from the root node of the ballots tree.
- isEliminated(): Accesses the eliminated field to return whether or not the candidate has been eliminated already.
- setEliminated(bool): Sets the eliminated field to the bool value that is passed in.
- getBallots(): Returns the ballots tree associated with the candidate.

## IR Class

Fields:

- candidates: An array of Candidate objects.
- numBallots: An int that represents the total number of ballots in the election.
- curNumCandidates: An int that represents the current number of candidates remaining in the election.
- audit: The audit file that is used for recording the election progress.
- input: The input file that is used for obtaining voting information and information about the candidates and election overall.

Methods:

- IR(FileReader, FileWriter): This is the constructor for the IR Class where we pass in the input election file and the audit file.
- eliminateCandidate(): Determines the candidate(s) with the least amount of votes from the candidates field. If there is a tie, this method calls breakTie() until the

single candidate being eliminated is determined. This method calls `reassignVotes()` to handle reassigning the votes properly.

- `reassignVotes(Candidate)`: Takes in a candidate object and handles the reassignment of votes when a candidate is eliminated. Calls the vote transferring method, `transferVotes()`, on the eliminated candidate tree to transfer the votes to the other candidates accordingly. Handles the redistribution of the eliminated candidate's votes within the other candidates' trees as well with a similar call to `transferVotes()` to transfer the votes to other candidates.
- `isMajority(bool)`: Iterates through the candidates array to see if one candidate has majority. If so, writes the name of the candidate with the majority in the audit file and returns true. If not, writes that there is not yet a majority in the audit file and returns false.
- `parseHeader()`: Parses the header for information about the election. For example, this method stores the number of ballots in the `numBallots` field, the number of candidates in the `curNumCandidates` field, and initializes our candidates field by creating `Candidate` objects with the appropriate names and parties.
- `processFile()`: Throughout this method, this method will refer to the candidates field to access the array of `Candidate` objects, the audit field to refer to the audit file, and the input field to refer to the election's input file. Trees are initialized for each candidate where each tree stores the ballots where that candidate was ranked first. For each ballot, creates node paths in the trees or adds a vote for each candidate accordingly through the `Tree.insert()` method. To do this, this method calls upon `OpenCSV` methods to obtain int arrays of the ballots. For each of these ballots, identify the candidate who is ranked first. Then, call the `Tree.insert()` method on this candidate's tree. Repeat this for all the ballots.
- `conductAlgorithm()`: This is the code that runs an IR election to completion. While there is no majority and the number of candidates remaining (`curNumCandidates`) is greater than 2, keep eliminating candidates. If there are two candidates left and no majority, the one with the most votes (popularity) wins. Otherwise, the one with majority votes would win. Calls `writeToAudit()` to record key information once the winner has been determined. Lastly, calls `printResults()` to display the results to the screen by iterating over the candidates field array and reporting the percentage of the total `numBallots` the candidate received, the number of first place votes they received, and the winner of the election. Changes the permissions of the audit file to be read only at the very end.

## CPL Class

Fields:

- `parties`: An array of `Party` objects
- `voteTotal`: An int representing the total number of votes casted in the election
- `numSeats`: An in representing the number of open seats that need to be filled in the election
- `audit`: A `FileWriter` object that CPL should write to throughout its methods

- input: A FileReader object representing the files from which CPL will get its data from

Methods:

- conductAlgorithm(): Updates the actual numSeats value of each **Party** (NOT numSeats for the CPL class). This function will sometimes call the drawLotto() and breakTie() functions, as this is supposed to allocate the appropriate number of seats to each party given all possible scenarios. Takes in no parameters and outputs void, but does use and modify its fields.
- drawLotto(): Returns a random party from the parties array. Used in the case where a party has earned more seats than it has candidates, so its unfilled seats must be given out to random parties.
- parseHeader(): Parses the header for information about the election. This will store the basic information, such as knowing how many parties there are and initializing an array of parties.
- processFile(): Follows parseHeader. This function is reliant on parseHeader successfully executing, and will create individual parties, count the votes, and fully store any information in the data structures setup from the parseHeader().
- printResults(): A simple function that prints the results of the election to the terminal. This should be called after conductAlgorithm has been completed

## Party Class

Fields:

- name: String representing the name of the party
- candidates: An array of strings holding the name of the candidates in the party (in order)
- numVotes: An int that is how many votes the party got
- numSeats: The number of seats the party has earned (int)

Methods:

- getName(): getter method for the name of the party. Returns string
- getNumVotes(): getter method for the numVotes of the party. Returns string
- setName(String): setter method for the name of the party
- setCandidates(String[]): setter method for for the candidates
- getCandidates(): returns the string array that is the candidates. getter method for candidates
- setNumVotes(int): setter method for numVotes
- setNumSeats(int): setter method for numSeats
- getNumSeats(): getter method for numSeats. returns an int
- incrementNumVotes(): increases numVotes by one
- incrementNumSeats(): increases numSeats by one

## Node Class

**Fields:**

- numVotes: an int that represents the number of votes stored within a particular node.
- children: a Node array that holds the children of the current Node.
- curDepth: an int that represents what level of the tree the Node is at, where the root begins at the 0th level.
- index: an int that represents this node's specific index identifier. That is, when this Node is a child, it shall be placed at this specific index within the parent's children array.
- parent: a Node that is the parent of the current Node.

**Tree Class****Fields:**

- root: A Node that is the root of the Tree.

**Methods:**

- addNode(Node, Node): Takes in two Nodes – one represents a parent Node and the other is the Node to be added. Adds the new Node to be in the parent Node's children array.
- insertNode(int[] ballot): Takes in an integer array that represents a single ballot, such as [1,,2,3]. If the Tree for the candidate ranked number one on a ballot is empty, then insertNode() will create a root node for the tree with a total count for the number of votes so far as one. Each node keeps track of exactly which index it appeared in the ballot that was passed in. Each node gets a children node array that is the length of the ballot. This method should loop over the number of possible rankings, and for each rank, search for the index at which the ranking occurred. If, within the current node's children array at the index where the next ranking occurred has null instead of node, create a new node and insert it into the proper index in the current node's children array. If there already is a node at the index, then simply add the votes to that node's vote total. Make this our new current node, and repeat the process for the rest of the rankings that exist in the ballot. If we are searching for a rank and cannot find a rank, that means that we have fully processed a ballot and can safely stop inserting nodes for that ballot.
- deleteNode(Node, Node): Takes in two Nodes – one represents a parent Node and the other is the Node to be deleted. Deletes the child Node from the parent's children array such that the Node to be deleted is no longer accessible in the tree. This means that the spot in the parent's child array where the child Node appeared should be set to null.
- getPreorder(Node): Takes in a Node that represents the root of the subtree to traverse in preorder fashion. Returns an ArrayList that represents the Nodes of the subtree with a preorder traversal.
- getNodes(int): Takes in an int that is the specific index identifier of the candidate to search the tree for. Calls upon getPreorder() and searches the resulting array for Nodes that have the candidate's specific index identifier. Returns an ArrayList of

Nodes that match the candidate's specific index identifier.

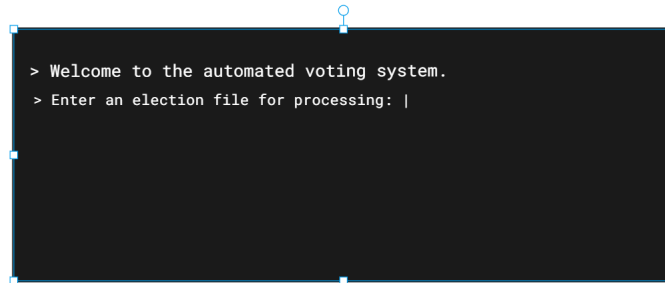
- **transferVotes(String, Node):** Takes in an index that represents the specific index identifier of the Candidate whose votes will be transferred. If transferring votes to a different candidate tree, then the Node parameter represents the root of another tree. Otherwise, it is null. Calls upon `getPreorder()`, `getNodes()`, `addNode()`, and `deleteNode()` for proper reassignment of votes. Firstly, call `getNodes()` for the eliminated Candidate's index identifier. Calls `getPreorder()` on each of these eliminated candidate nodes. If the parent of these traversed nodes is the eliminated candidate, get the parent of the eliminated node and make this the tracker node. Look in the tracker node's children. If there is a node that matches the traversed node at the traversed node's specific index identifier, add the votes to this node (outside the current subtree rooted at the eliminated candidate). If not, create a new Node and call `addNode()` to include this new Node in the parent's children array. Regardless, make this Node our new tracker, unless the traversed Node was a leaf node. In that case, do not change what the tracker node was referencing previously. Continue this process for all the traversed nodes. Do this for each subtree rooted at an eliminated candidate's node. Lastly, call `deleteNode()` on the eliminated candidate nodes. There is a similar process for when we transfer the votes from an eliminated candidate to other candidates' trees; in this case, the Node parameter would represent our first tracker Node.

## **6. HUMAN INTERFACE DESIGN**

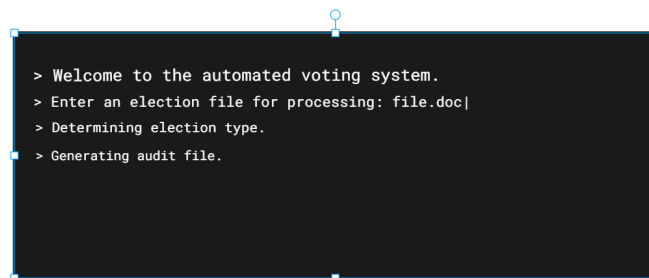
### **6.1 Overview of User Interface**

The system is run as a command line program. The system prompts the user to enter an election file. The system then updates the user on its process as the election is determined, and once a winner is called, the election results are displayed in the terminal.

## 6.2 Screen Images

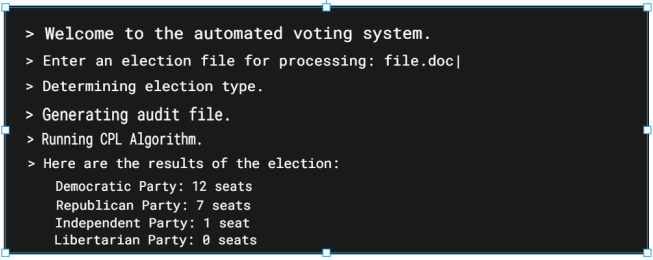


```
> Welcome to the automated voting system.  
> Enter an election file for processing: |
```

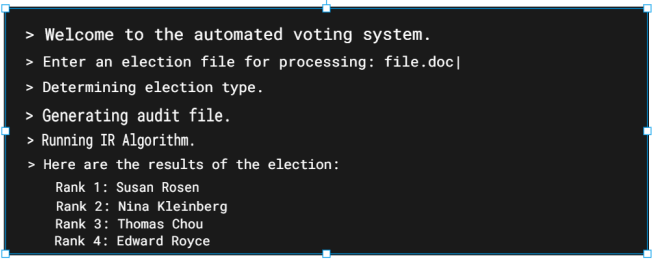


```
> Welcome to the automated voting system.  
> Enter an election file for processing: file.doc|  
> Determining election type.  
> Generating audit file.
```





```
> Welcome to the automated voting system.  
> Enter an election file for processing: file.doc|  
> Determining election type.  
> Generating audit file.  
> Running CPL Algorithm.  
> Here are the results of the election:  
    Democratic Party: 12 seats  
    Republican Party: 7 seats  
    Independent Party: 1 seat  
    Libertarian Party: 0 seats
```



```
> Welcome to the automated voting system.  
> Enter an election file for processing: file.doc|  
> Determining election type.  
> Generating audit file.  
> Running IR Algorithm.  
> Here are the results of the election:  
    Rank 1: Susan Rosen  
    Rank 2: Nina Kleinberg  
    Rank 3: Thomas Chou  
    Rank 4: Edward Royce
```

### 6.3 Screen Objects and Actions

The only screen object the user interacts with is the command line interface. The associated action with the interface is file inputting. As the system does the rest of the work, a series of process updates follow the file input.

## 7. REQUIREMENTS MATRIX

Functional Requirement	Description	Implementation
Pass a File Into the Program	User passes .csv file into program	main () in Main Class and start() in RunElection Class
Determine Election Type	System determines election type from input file header	start() in RunElection Class
Create an Audit File	System creates an audit file with the election type and date	start() in RunElection Class
Process File for CPL Election	System processes input file to store CPL election data	processFile() in CPL Class
Conduct CPL Algorithm	System allocates seats based on CPL election specs	allocateSeats() in CPL Class and conductAlgorithm() in CPL Class and Election Parent Class, and runCPL in RunElection Class
Assign Seats via Lottery	System draws lottery to assign unclaimed seats in CPL	drawLotto() in CPL Class
Process File for IR Election	System processes input file to store IR election data	processFile() in IR Class
Conduct IR Algorithm	System ranks candidates based on IR specs	eliminateCandidate(), reassignVotes(), conductAlgorithm(), and isMajority() in IR Class and runIR() in RunElection Class
Break a Tie	System breaks a tie between two or more candidates/parties	breakTie() in the Election Parent Class
Write to the Audit File	System writes information on elections to audit file	writeToAudit() in Election Parent Class
Display Results to Terminal	System prints results to the screen	printResults() in IR or CPL class Utility Class

## **8. APPENDICES**

There are currently no appendices.