Secure Internet Voting in Ireland Using the Open Source Kiezen op Afstand (KOA) Remote Voting System

MSc. Dissertation Report

Dermot Cochran

A dissertation report submitted in part fulfillment of the degree of
MSc. in Advanced Software Engineering
under the supervision of Dr. Joseph Kiniry.

School of Computer Science and Informatics

University College Dublin

11 March 2006

Abstract

This dissertation explores whether the Open Source Kiezen op Afstand Remote Voting System can contribute to meeting the needs of and comply with Secure Internet Voting in Ireland. This is an important question since the early attempts to introduce electronic voting using voting machines has met with considerable opposition, especially in relation to its security and integrity. The history of the Irish voting system is described. Considerable trust has been built up over many years use of the manual paper system.

The KOA system is then described, as is the work of the SoS group in the Netherlands on the vote counting subsystem for KOA. The use of Java Modeling Language (JML) is discussed with regard to how it can be used to specify and verify the accuracy and security of software using extended static checking (ESC/Java2). Applied Formal Methods were used by the SoS group to develop a vote counting subsystem for the Netherlands. This dissertation describes how this approach can be used to specify a vote counting subsystem for Ireland.

The functional requirements for the Irish election counting algorithm have been extracted from the official documents. JML along with ESC/Java2 has then been used to specify a vote counting algorithm for Irish elections. The design of this specification and the associated Java class models are outlined. Questions about the influence of random transfers of votes in manual counting procedures are considered, as are the ways in which automated vote counting can give a more accurate result. A glossary of voting system terms and definitions is also included. Finally, a JML specification for Irish election vote counting, with Java class and method signatures, but without implementation code, is included.

Acknowledgments

The author gratefully acknowledges the invaluable help and support of the following people:

My dissertation supervisor, Dr. Joseph Kiniry, who has been unfailingly positive, enthusiastic and encouraging about this project.

Dr. Lorcan Coyle and Alan Morkan of the Systems Research Group, University College Dublin, for taking time to meet with me at the early stages of this work.

My father, Robert Cochran, with his interest in the Irish electoral system and for his many helpful comments on my dissertation.

My wife, Margaret Gibney, who made sure that I had the time to complete the dissertation.

Table of Contents

1 Introduction

The terms used in this report are defined in the glossary.

1.1 Voting in Ireland

The Dáil, Ireland's lower house of parliament, is composed of 166 members representing 41 constituencies. Each constituency elects multiple members to parliament. The average constituency elects four representatives; every constituency elects at least three representatives.

1.1.1 Proportional Representation with Single Transferable Vote

The Irish electoral system uses proportional representation and transferable votes. This combination increases the representativeness of the Dáil. Irish voters, by ranking the candidates, give instructions as to who should receive their support should the first choice candidate be eliminated or elected.

Recalling that, the Dáil uses multi-member districts, the winners of the election are those candidates who exceed the quota for election. The quota for election is the minimum number of votes need to be elected or is equal to

```
 (Valid Votes Cast / (Number of Seats + 1)) +1
```

For example, in a district with three seats the quota for election would be 25% of the votes plus one.

Surplus votes are the number of votes in excess of the threshold of election a candidate receives. Surplus votes are transferred proportionally to the remaining candidates according to the indicated second preference of the voters.

If the election is undecided after counting the first preferences and transferring surplus votes, then the lowest polling candidate is eliminated. The ballots cast initially in support of this candidate are now counted according to their indicated second preference.

1.1.2 Tipperary North 2002: An Example

Tipperary North is represented by three members in the Dáil. In the 2002 elections six candidates ran to fill three seats. 40,964 valid votes were cast, resulting in a quota of 10,242 (25%) votes required to win.

Candidate	Party	First Count	Second Count	Third Count	Last Count
Noel Coonan	Fine Gael	6,108 (14.9%)	6,436	9,085	9,437
Bill Dwan	Progressive Democrats	1,446 (3.5%)	0	0	0
Maire Hoctor	Fianna Fáil	8,949 (21.8%)	9,320	11,040*	11,040

Candidate	Party	First Count	Second Count	Third Count	Last Count
Michael Lowry	Independent	10,400* (25.4%)	10,400	10,400	10,400
Kathleen O'Meara	Labour	5,537 (13.5%)	5,877	0	0
Michael Smith	Fianna Fáil	8,526 (20.8%)	8,842	9,642	10,088*

*indicates Count where the candidate crossed the threshold. **Bold** indicates secured seat.

Count 1: One candidate, Lowry, exceeded the quota in the first round.

Count 2: After transferring the surplus vote, the lowest polling candidate Dwan was eliminated. Each vote for Dwan was then transferred to its next preference. This was not enough though to elect any candidate.

Count 3: O'Meara was subsequently eliminated. Following this second elimination, Hoctor secured a seat in the third round.

Count 4: Most of the surplus vote from Hoctor’s election were transfered to Smith, ensuring his election instead of Coonan.

The use of multi-member districts and transferable vote minimized the number of voters unrepresented by the election’s outcome. In Tipperary North, only Coonan supporters, 23% of the voters, do not have their views represented in parliament. The Irish system permits voters that support less popular candidates to still be represented [1].

1.1.3 Electronic Voting in Ireland

In June 2000, the Irish Department of Environment and Local Government issued tenders for supply of an electronic voting system. In December 2000, Nedap was selected as the supplier for voting machines, and the Integrated Election Software (IES) system from Powervote was selected as the vote counting software.

The system was first piloted in the May 2002 general election in three constituencies (Meath, Dublin North and Dublin West). For the Nice II referendum in October 2002, a further pilot was carried out in seven constituencies (Dublin Mid-West, Dublin North, Dublin South, Dublin South-West, Dublin West, Dun Laoghaire and Meath).

The Nedap voting machine is a suitcase-sized, portable computer-based voting machine. It can handle up to 5 simultaneous ballots, with up to 30 selections (candidates) per ballot.

A ballot module is a removable memory device used to store the voter selections in random sequence. Each ballot module can hold up to 30,000 voter selections, and is pre-programmed with a unique identification number. Each voting machine has a backup ballot module and a battery backup.

The Integrated Election Software (IES) system supplied by Powervote Ltd. runs on a standard PC, using a Microsoft Access database. This is used to load details of the election onto the ballot modules in advance of the poll. At the end of the poll, the votes are loaded from the ballot modules into the IES system, mixed into a random sequence and counted. The IES system then prints and displays the results on screen.

Details of the elections and candidates are keyed into the IES system. Each ballot module is then connected to the IES system via a programming unit and details of the election are downloaded onto the ballot module.

The ballot module is then installed into the voting machine at the polling station. A printout is generated to verify that the ballot module is empty of votes at the start of the poll. Voters are authenticated using the electoral register as per the traditional system. Once a voter is authenticated, they are issued with a token and they proceed to a voting machine. They give the token to the control operator, who will then enable the machine for voting. The voter records their preference for each election by pressing buttons in the relevant column. When the voter is complete, they must press the ‘Cast Vote’ button to record their preferences.

The control operator can see on their control unit when the ‘Cast Vote’ button and each preference is stored on the ballot module. If the voter leaves the voting machine with pressing the ‘Cast Vote’ button, they control operator may advise them that their vote has not been completed. The control operator must log each uncompleted vote on a paper form before resetting the voting machine for the next voter.

At the end of the poll, a printout is generated on the voting machine to show the number of votes recorded. The control operator and presiding officer reconcile the number of tokens issues, the number of votes recorded and the number of uncompleted votes for each machine. The voting machine copies all votes on the ballot module onto the backup ballot module. The ballot module is removed and sent to the count centre.

At the count centre, after verifying the unique identification number on the ballot module, each ballot module is connected to the IES system and the votes are loaded into the database. If more than one PC is being used for loading of ballot modules, then votes can be transferred between PCs on disk. When all ballot modules have been downloaded, the IES system mixes the votes received into a random sequence and then proceeds to count the votes via the PR-STV rules. The count is only interrupted if the drawing of lots in case of a tie is required. When the count has completed, details of the count can be printed and viewed on screen.

The random selection used when distributing a surplus in the Irish system presents a particular challenge for auditing the count. Where two counts are carried out (one electronic and one manual) using two different random selections, the results of the counts may well be different. This is particularly likely in constituencies where the last seat may be decided on a handful of votes.

Cochran and Hogan have suggested that the correct way to deal with this is to undertake a statistical analysis of the results. A legal procedure should therefore be established so that when the outcome of an election, following randomization, is decided by a margin less than the expected randomization margin, a full statistical analysis be undertaken.. So if that margin is determined to be say 5%, then if a result less that 5% from the next candidate, the system is instructed to repeat the analysis at least 30 times, and to do an analysis of the results. Comparing these results should (if the randomization is done correctly) show a ‘bell curve’ from which the statistical mean (average) can be calculated. This mean value should then determine the formal result of that count [2].

1.2 Security and Remote Voting

The question of secure remote voting is quite controversial. Many people question if the Internet is secure enough for something as important as a general election. Nevertheless the Dutch government were willing to take the risk of introducing a remote voting system for the European parliamentary elections. The Kiezen op Afstand (KOA) system is the first Open Source Internet voting system to be developed in the world [3]. The Dutch ministry of internal affairs outsourced the original project to LogicaCMG, a Dutch consulting firm, who designed, developed tested and deployed the KOA system in the Netherlands. Part of the KOA system was tested by the Security of Systems (SoS) group at the Radboud University Nijmegen.

The Dutch Government released the source code for the KOA system under the GNU General Public License in July 2004 after the system was trailed in the European parliamentary elections [4], but part of the original system is proprietary and owned by LogicaCMG. Thus the code released under GPL was incomplete. Alan Morkan, working under the supervision of Dr. Joseph Kiniry of the Systems Research Group at University College Dublin, Ireland, has been working on completing the Open Source version of the KOA system [5].

1.3 KOA Remote Voting System

The KOA system is designed for remote voting in the Netherlands. It does not comply with Irish electoral law and does not include special cases such as postal/special voting or the needs of blind or illiterate voters.

1.3.1 Polling

The KOA system can serve multiple electoral areas, each with multiple voting machines. All votes cast are encrypted and stored. Before the start of the elections the list of authorized voters is stored. The system checks if a voter has already cast his/her vote.

1.3.2 Voter Registration

To ensure that each voter casts at most one vote, and to reduce the likelihood of personation, each voter is assigned a voter code and an access code.

The voter code is a unique 9-digit number generated by the KOA system. The secrecy of the ballot requires that the voter code cannot be used to identify the voter.

The access code is a 5-digit number chosen by the voter and stored in encrypted form using a one-way encryption algorithm. For security reasons the system does not differentiate between the situations "access code not present" and "access code incorrect".

1.3.3 Audit Trail

The following information is recorded for every attempt at voting:

- an indication if this voter has already voted or not,
- the date and time at which the vote was cast,
- the number of failed attempts made,
- the total number of attempts made,
- the time of the last failed attempt.

After a certain number of attempts to login all further attempts are blocked for a certain period. The parameters for this function can be changed. The default is a maximum of three attempts and a blocking period of one hour. The total counter per voter will never be returned to zero – it is used to detect a systematic attempt at personation.

1.3.4 Votes

The following information is recorded for each vote cast:

- the candidate code of the candidate that received the vote,
- the name and initials of the candidate,
- the position on the candidate list,
- the list number,
- the political party of the candidate.

1.3.5 Web Server Interface

The KOA system is designed to be accessed by voters using the Internet or from a touch tone telephone.

For Internet users, the web server interface was been kept as simple as possible, so that a minimum of data has to be transferred, which ensures that voters with a slow connection can still use this facility. The most important requirement is that the browser support the secure socket layer (SSL) protocol and session cookies.

A verification screen is displayed which requires the voter to enter the voter code and access code. If the data is incorrect the voters can retry a predetermined number of times. If verification continues to fail an error message is displayed. Further attempts are not possible.

A separate web server interface is used for systems administration.

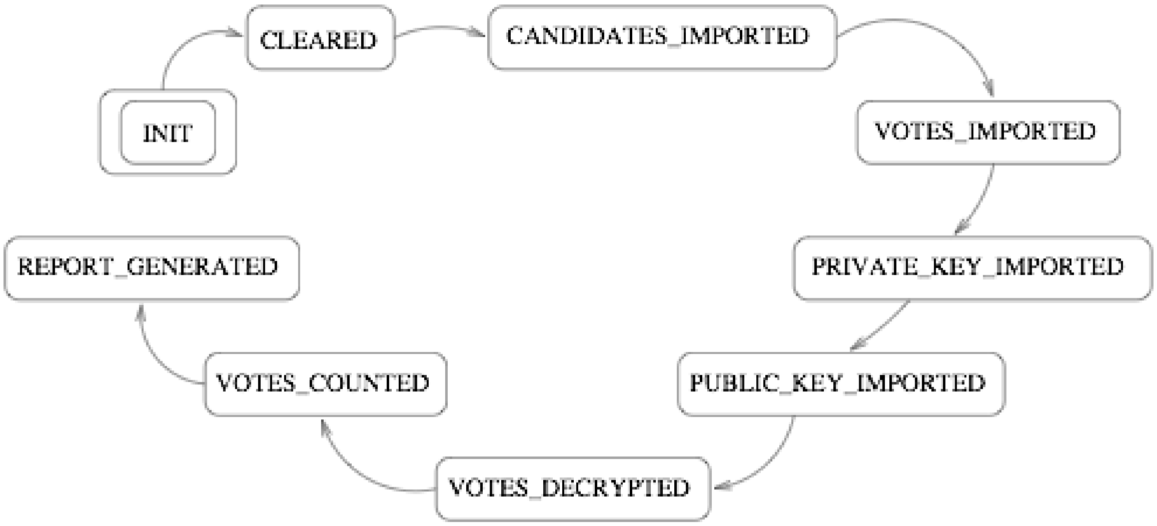


Figure 1: The flowchart for the KOA system

The system manages voter login by first checking if the system is open for voting and whether the voter code and access code are not empty and are of the correct length and format. If the password is incorrect, the number of invalid login attempts is incremented and if it equals the maximum number of invalid logins the voter account is locked and a time-stamp is set for the next available opportunity for the voter to try and login.

1.3.6 Evaluation of KOA in the Netherlands

The Security of Systems (SoS) group at Radboud University Nijmegen, was involved in the KOA system in several ways. Prof. Bart Jacobs, head of the SoS group, was part of an external expert panel that reviewed the system's requirements and design, but performed no review of the software. That panel wrote a report critical of many aspects of the system, and as a result many of its recommendations have been incorporated in the later versions of the system.

Three of the key recommendations were: 1) The system should not be designed, implemented, tested, and administered by the same company. 2) The source code of the system should be made available to any parties interested in reviewing it. 3) The system should log all raw data in the voting process so that an independent recount is possible by any interested third party.

The Dutch ministry of internal affairs requested that the SoS group audit the remote voting system and attempt to break into, or otherwise disrupt, a trial vote that took place in late 2003. For a one week period the group analyzed the remote voting system's network, servers, communication, web site, etc., just as a black hat hacker would do during the actual vote.

This analysis and experience helped the system designers and administrators learn how to react to a real-world attack. It also pointed out subtle network and physical attacks possible by determined hackers. In particular, the SoS group was unsurprisingly able to effect a denial of service attack on the service with little effort. As a result, the remote voting system was "tightened down" even further and protocols were put in place to deal with denial of service attacks.

1.3.7 Vote Counting Subsystem for KOA in the Netherlands

The SoS group also became directly involved in the implementation of a small but critical part of the remote voting system. Due to recommendation 1) above, the Ministry decided to open up a bid for a third party construction of the vote counting subsystem. It is thought that if a third party designs and implements the tally software in isolation then the likelihood of fraud is vanishingly small. The SoS group bid on this project and won it by virtue of their radical proposal: the vote counting system should be formally verified.

In early 2004, three members of the SoS group designed, implemented, and verified a vote counting subsystem. Leveraging their past experience with verification of Java, the vote counting system was constructed in Java and specified with the Java Modeling Language. The system was extensively tested, as thousands of unit tests were generated from the specification. Additionally, key components were verified using ESC/Java2, a static verification tool. As a result, they had extremely high confidence in the correctness and accuracy of the vote counting system.

1.4 Applied Formal Methods

Applied Formal Methods were used by the SoS group to develop a vote counting subsystem for the Netherlands. This dissertation describes how they can be used to specify a vote counting subsystem for Ireland.

1.4.1 Java Modeling Language

The Java Modeling Language (JML) is a formal behavioral specification language for Java [6]. It is a small extension to the Java programing language and uses annotations to express statements in mathematical logic. JML can be used to describe both

- a mathematical specification for the functional behavior of the software, and

- a technical design for the data model and class diagrams.

JML also acts as a test specification language and the JML tools can be used to automatically generate JUnit test code.

1.4.2 Extended Static Checker for Java, Version 2

ESC/Java2 is a static verification tool with the ability to issue warnings and errors based on the additional information in JML specifications that describe the intended behavior of the source code [7].

'Extended Static Checking' is the translation of the program source and its specifications into verification conditions; these are passed to a theorem prover, which in turn either verifies that no problems are found or generates a counterexample indicating a potential bug. The tool and its built-in prover operate automatically with reasonable performance and need only program annotations against which to check a program's source code. The annotations needed are easily read, written and understood by those familiar with Java and are partially complete, but wholly consistent, with respect the syntax and semantics of the Java Modeling Language.

The use of extended static checking has advantages over the current practice in the software industry. In the author's experience a typical software process consists of type checking, followed by manual testing, followed by a code review, followed by more manual testing, followed by quality assurance testing. Extended static checking is a means to automate the time consuming and labour intensive process of manual testing and detailed code reviews.

Design specifications written in JML with Java method prototypes can be checked using ESC/Java2 before the implementation code is written. If each method body contains the annotation '//@ assert false', then the specification can be checked for soundness. When there is a logical contradiction within the specification, it follows that 'assert false' evaluates to 'true'.

ESC/Java2 has been used to typecheck and check the soundness of the JML specification and Java files with JML annotations, that are listed in the Appendices.

2 Specification for Irish vote counting system (Votáil)

2.1 Functional Requirements

The functional requirements of the Dail election algorithm and dependent data-structures are identified by section, item number and page within the first of the source documents, mentioned below. The item number in the following table refers to the subsection number within each section. The page number is the page which contains the text from which the requirement can be identified.

The '@see requirement' documentation tag in the JML specification indicates a reference to one or more of these functional requirements, for example;

```
@see requirement 1, section 3, item 2, page 12.
```

Note that pages 16 and 17 of the first source document appear to contain typographic errors. Item 4 is incorrectly labeled as item 3 and item 7 is incorrectly labeled as item 1.

2.1.1 Source Documents

The following documents were taken as being the complete functional specification for the Dáil election count algorithm [8,9]:

- Department of Environment and Local Government, Commentary on Count Rules, sections 3-16, pages 12-65, http://www.cev.ie/htm/tenders/pdf/1_2.pdf

- Update No. 7 dated 14 April 2002, http://www.cev.ie/htm/tenders/pdf/1_4_7.pdf

The other updates contain implementation notes only. Update No. 7 clarifies the treatment of candidates with zero votes.

2.1.2 Overview of Vote Counting Algorithm

The number of votes for each candidate are counted. If any candidate has more than a quota of votes then he or she is elected and his or her surplus votes are transferred to the next preference candidate. If there are more candidates than seats and all surpluses have been transfered, then exclude the candidate with least votes and transfer those votes to the their next preferences. This process is repeated until the number of candidates remaining equals the number of seats remaining.

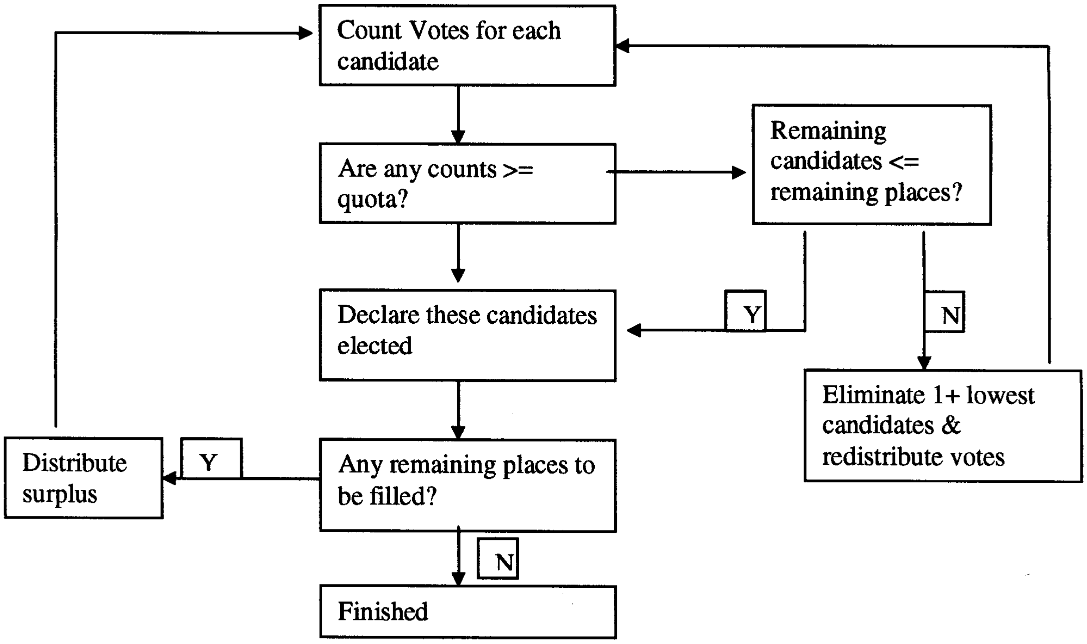

Count Votes for each candidate
Are any counts >= quota?
Remaining candidates <= remaining places?
Y
N
Declare these candidates elected
Eliminate 1+ lowest candidates & redistribute votes
Distribute surplus
Y
Any remaining places to be filled?
N
Finished

Figure 2: The Vote Counting Process

The following tables list all of the functional requirements of interest in the source documents. A requirement is of interest when it expresses a mathematical invariant, constraint, precondition, postcondition, or any combination of such, and is applicable to Dáil elections.

2.1.3 First Count Calculations

Requirements 1–7 describe the criteria for election and for saving a deposit.

ID	Functional Requirement	Section	Item	Page
1	Counting does not begin until all votes are loaded.	3	2	12
2	The total number of first preference votes must remain the same after each count.	3	3	12
3	The quota is equal to ((Number of valid votes cast)/(Number of seats being filled + 1)) + 1, ignoring any remainder.	3	3	13
4	Any candidate with a quota or more than a quota of votes, is deemed to be elected.	3	3	13

ID	Functional Requirement	Section	Item	Page
5	The surplus of an elected candidate is the difference between the quota and his/her total number of votes.	3	3	13
6	The minimum number of votes required for a candidate to secure the return of his/her deposit is one plus one-quarter of a quota based on the total number of seats in the constituency.	3	3	13
7	Any elected candidate automatically saves his/her deposit.	3	3	13

2.1.4 Decision Making

Requirements 8–13 describe the overall procedure for counting of votes.

ID	Functional Requirement	Section	Item	Page
8	If the number of continuing candidates is equal to the number of seats remaining unfilled, or the number of continuing candidates exceeds by one the number of unfilled seats or there is one unfilled seat, then do not distribute any surplus unless it could allow one or more candidates with at least one vote to save their deposits.	4	2	15
9	Not more than one surplus is distributed in any one count.	4	3	16
10	Where there are seats remaining to be filled, but no surpluses available for distribution, the lowest continuing candidate or candidates must be excluded.	4	4	16
11	There must be at least one continuing candidate for each remaining seat.	4	4	16
12	Two or more lowest continuing candidates must be excluded together if the total of their votes plus the sum of any undistributed surpluses is less than the votes of the next highest candidate and the either the second lowest candidate has already saved his/her deposit or the exclusion of candidates individually would not save any of their deposits.	4	4	16

ID	Functional Requirement	Section	Item	Page
13	The sum of the votes of the set of lowest continuing candidates must be greater than zero.	4	4	17

2.1.5 Surplus Distribution

Requirements 14–27 describe the procedure for distribution of surplus votes., when candidates are elected by exceeding the quota — as described in requirement 3. The order in which votes are selected for distribution is described in requirements 37–39.

ID	Functional Requirement	Section	Item	Page
14	The distribution of the largest available surplus is mandatory if the sum of surpluses could save the deposit of any continuing candidate with at least one vote.	5	2	18
15	The distribution of the largest available surplus is mandatory if the sum of surpluses could elect a continuing candidate, unless the last seats are being filled.	5	2	18
16	The distribution of the largest available surplus is mandatory, unless the last seats are being filled, if the sum of surpluses could save the lowest candidate, with at least one vote, from exclusion.	5	2	18
17	Where two or more candidates have equal surpluses arising from the same count, the surplus of the candidate with the largest number of votes at the earliest count at which two or more of such candidates had unequal votes is distributed first. Where all such candidates had equal votes at all counts, random selected is used.	6	3	21
18	Where two or more candidates have equal surpluses arising from different counts, the surplus that arose at the earliest count is distributed first.	6	4	22

ID	Functional Requirement	Section	Item	Page
19	Only the set of votes received by the elected candidate at the count at which he/she was elected (last set of votes) is used to distribute the surplus votes among the continuing candidates. Thus, all the votes of a candidate are taken into account in a surplus distribution only when a candidate is elected on the first count.	7	2	23
20	The number of votes in each sub-set of transferable votes is calculated and summed to determine the total number of transferable votes in the last set of votes.	7	2	24
21	When the total number of transferable votes in the last set of votes is greater than the surplus, only a proportion of the transferable votes can be transferred.	7	3.1	24
22	The number of votes in the surplus is divided by the total number of transferable votes in the last set of votes. This transfer factor is multiplied in turn by the total number of votes in each sub-set of next available preferences for continuing candidates.	7	3.2	25
23	If the total number of units in all the quotients is less than the surplus, the remainders in the quotients must be examined. In this situation, the number of votes for inclusion in the surplus distribution from particular sub-sets is increased by one each based on the highest remainders in the quotients, until the difference between the total number of units and the surplus is made up.	7	3.2	25
24	In cases where two or more quotient remainders are equal and not all of them can attract an additional vote in the surplus distribution, the remainder in the largest sub-set of transferable votes (i.e. next available preferences) is deemed to be the largest and an additional vote is included in the surplus distribution from that subset.	7	3.2	25

ID	Functional Requirement	Section	Item	Page
25	Where the sub-sets of transferable votes of two or more continuing candidates are equal, the remainder of the candidate with the largest number of votes at the earliest count at which two or more of the candidates had unequal votes is deemed to be the largest. Where all such candidates had equal votes at all counts, random selection is used.	7	3.2	25
26	When the total number of transferable votes in the last set of votes is less than the surplus, all the transferable votes are transferred, plus that number of non-transferable votes which makes up the difference between the number of transferable votes and the surplus.	7	5	26
27	The set of nontransferable votes not effective is kept separate from all other sets of votes and, where they arise, the number of such votes is included in the sum of recorded votes calculated at the second and subsequent counts.	7	5	27

2.1.6 Exclusion of Lowest Candidates

Requirements 28–36 describe the exclusion of lowest continuing candidates, when there are no surpluses available for distribution.

ID	Functional Requirement	Section	Item	Page
28	Where the two or more lowest candidates have equal votes, the candidate with the smallest number of votes at the earliest count at which two or more of such candidates had unequal votes is deemed to be the lowest candidate and is excluded. Where all such candidates had equal votes at all counts, random selection is used.	8	2	28
29	All candidates with zero votes and the lowest candidate with any votes must be excluded together in one operation. Lowest Candidate means all candidates with zero votes and the lowest candidate with any votes.	9	1	30

ID	Functional Requirement	Section	Item	Page
30	When the lowest continuing candidates are excluded and their votes distributed in the next count, their votes are examined and the transferable votes are distributed to continuing candidates according to the next available preferences marked.	10	2	40
31	Each continuing candidate is credited with an additional number of votes equal to the total number of votes transferred to him/her.	10	2	40
32	Any votes of the excluded candidates which have no next preferences marked on them for continuing candidates are non-transferable and are placed in a set of votes entitled non-transferable votes not effective.	10	2	40
33	When the number of continuing candidates equals the number of seats remaining unfilled, the continuing candidates are deemed to be elected to fill the remaining seats.	11	2	42
34	When the number of continuing candidates is one greater than the number of seats remaining unfilled and the total of the votes credited to the lowest continuing candidate together with any surplus not transferred is less than the number of votes credited to the next highest continuing candidate, the continuing candidates, with the exception of the lowest candidate, are deemed to be elected.	11	3	43
35	When only one seat remains unfilled and the votes of the highest continuing candidate exceed the total of the votes of all other continuing candidates together with any surpluses not transferred, the highest continuing candidate is deemed to be elected.	11	4	44
36	Any available surplus which could possibly save the deposit of a continuing candidate must be distributed after all seats are deemed to be filled.	11	5	44

2.1.7 Random Ordering of Votes

Requirements 37–39 describe the way in which votes are mixed randomly, which has implications for the way surplus votes are distributed.

ID	Functional Requirement	Section	Item	Page
37	All the votes recorded at an election in a constituency must be thoroughly mixed together before counting to ensure that vote transfers on the distribution of a surplus are representative.	16	1	57
38	The votes within each sub-set of next available preferences for a particular candidate are sorted in the same relative order as the votes were mixed and numbered before the count began.	16	5.2	61
39	The particular votes to be transferred from each sub-set are the votes with the highest random numbers.	16	5.4	63

2.2 Methodology

2.2.1 Overall Process: Converting Semi-Formal English into Formal Specifications

The process of developing formal specifications has been used for purposes other than vote counting [10]. Early examples of this include the use of Z.

Legal documents or functional specifications written by a business analyst are an example of semi-formal English. Requirements analysis is used to produce a numbered list of requirements in a structured format. Each requirement is a set of sentences or paragraphs describing one property of the system. The statement of each requirement might include diagrams and additional information such as the relationship to other requirements.

It is then possible to identify all the assertions and then to translate those into a formal language at the same level of abstraction. The abstract specification is then refined by a number of steps into more detailed design and then into a design than can be expressed in a programming language.

The source documents are transformed into a formal specification using JML and refined into an implementation in Java.

2.2.2 Methodology Example

This is a concrete example of how the methodology was applied.

Section 7 item 3.2 on page 25 of the first source document states:

> As a first step, a transfer factor is calculated, viz. the number of votes in the surplus is divided by the total number of transferable votes in the last set of votes. This transfer factor is multiplied in turn by the total number of votes in each sub-set of next available preferences for continuing candidates (note that the transfer factor is not applied to the sub-set of non-transferable votes in the set of votes).

The requirement is identified as follows:

> The number of votes in the surplus is divided by the total number of transferable votes in the last set of votes. This transfer factor is multiplied in turn by the total number of votes in each sub-set of next available preferences for continuing candidates.

This requirement is specified in one of the JML postconditions for the `getActualTransfers` method, the javadoc for which is shown below:

```
/**
 * Determine actual number of votes to transfer to this candidate, excluding
 * rounding up of fractional transfers
 *
 * @see requirement 25 from section 7 item 3.2 on page 25
 *
 * @design The votes in a surplus are transfered in proportion to
 * the number of transfers available throughout the candidates ballot stack.
 * The calculations are made using integer values because there is no concept
 * of fractional votes or fractional transfer of votes, in the existing manual
 * counting system. If not all transferable votes are accounted for the
 * highest remainders for each continuing candidate need to be examined.
 *
 * @param fromCandidate Candidate from which to count the transfers
 * @param toCandidate Continuing candidate eligible to receive votes
 * @return Number of votes to be transfered, excluding fractional transfers
 */
```

```
       ensures
       \result == (getSurplus (fromCandidate) *
                    getPotentialTransfers (fromCandidate,
                      toCandidate.getCandidateID()) /
                  getTotalTransferableVotes (fromCandidate);
```

2.2.3 Soundness Checking using ESC/Java2

Each of the JML and Java files listed in the Appendices have been typechecked and shown to be sound using the ESC/Java2 tool. ESC/Java2 does type checking on all JML and Java statements in the specification.

ESC/Java2 attempts to automatically prove that a given method implementation satisfies its specification. Each method of our system has an implementation containing the JML assertion "assert false", which essentially says "false should always be true at this point in the program." Obviously false is never true, therefore ESC/Java2 should always *fail* to prove these assertions. If ESC/Java2 *is* able to prove one of these assertions, then this means that the associated specifications, either or both the contract for the method in question or its related invariants, are unsound. That is to say, the specifications contain a contradiction, and because of that contradiction causes ESC/Java2 to be able to prove anything; in particular, it can prove that false is true

2.2.4 Completeness

Even if the specification is logical sound, it could be incomplete.

The specification can be said to complete if all the functional requirements have been modeled, if the specification is sound and if there is an overall element which links the specification together i.e. the specification works as a unified whole and not just independent sets of constraints or invariants. In the Votail module this is mainly achieved by the use of an abstract state machine.

2.2.5 Validation

Validation of the specification requires an independent review by a JML expert with an expert in Irish electoral law, and will take a significant amount of time to complete. Firstly the list of functional requirements would need to be confirmed. Secondly the JML specification would need to be shown to be consistent with the legal and functional requirements.

Note that the JML tools have the ability to generate unit tests, but such tests are valid only if the JML specification is valid. Likewise, manual testing can only cover a finite number of test cases. For example, in an election with 5 candidates and 40,000 voters there would be 5! = 120 ways each elector could vote and therefore 120 raised to the power of 40,000 combinations to consider. So, although manual testing would suffice for checking the layout of reports and to establish the usability of the system, it could never be used to validate the correctness of the algorithm.

Systems and specifications of this complexity are within the realm of modern model checkers like Bogor, even though full functional testing is impossible [11].

2.3 Technical Design

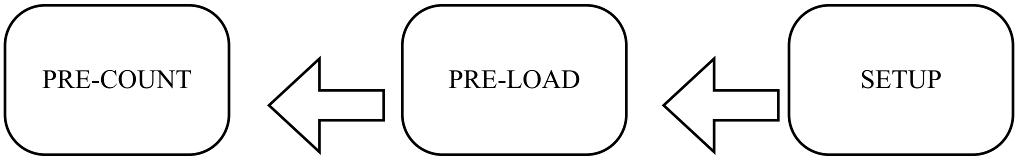
In additional to the functional requirements, there are design decisions which follow from the functional requirements by implication. These include the use of an abstract state machine, the usage of primitive arrays rather than abstract data types such as lists, and using random numbers.

2.3.1 Abstract State Machine

When the word ‘state’ is used in this document or in the specification it is always used in the technical sense of ‘abstract state machine’, not in the political sense of ‘nation state’, unless otherwise indicated.

The election count algorithm is modeled as an abstract state machine with state values and transitions between those states. There is no error state because the specification is written in the positive, so that error handling is not part of the high level design. This ASM is similar to the ASM in figure 1. The PRE-LOAD state corresponds to CANDIDATES_IMPORTED, and PRE-COUNT corresponds to VOTES_DECRYPTED.

There is a linear sequence of activity from one state to another. In the PRE-LOAD state all candidate details have been imported. In the PRE-COUNT state all votes have been loaded. In the FINISHED state all seats have been filled and there are no continuing candidates. In the REPORT state the results have been tabulated.



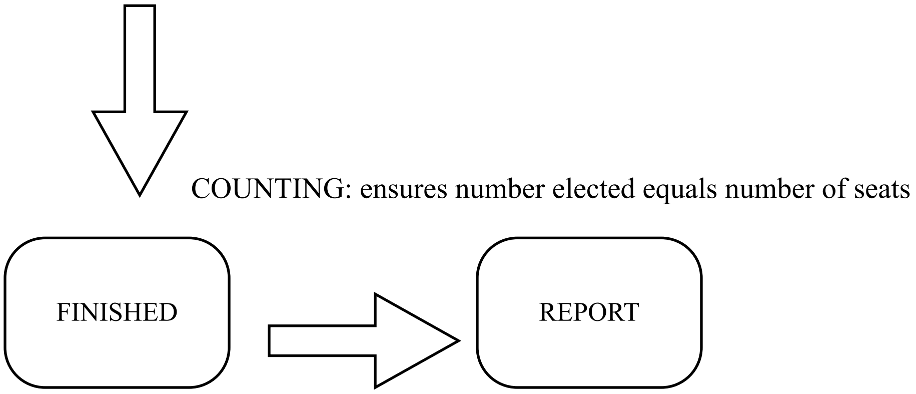


Figure 3. Abstract State Machine for Vote Counting Sub-System

2.3.2 Primitive Arrays

Primitive arrays are used in the specification instead of abstract data types like lists or hash-maps. ESC/Java2 is able to do more complete static analysis when primitive arrays are used instead of abstract data types.

2.3.3 Random Numbers

When a random selection between equal candidates is required, the system can ask the returning officer to draw lots, or else attempt to simulate the process using random numbers assigned to each candidate.

The Java class library is called to determine the values of the random numbers. There is a legal requirement to actually simulate the mixing and numbering of votes, as described in functional requirements 37–39, although the use of unique, uniformly distributed, randomly generated numbers has the same result.

The procedure for Dáil elections requires that if more votes are available for transfer than are contained in the surplus, then ballots are selected randomly in proportion to the number of transfers to each candidate. This introduces a random effect into the distribution of subsequent transfers.

2.3.4 Architecture

The `ElectionAlgorithm` object will be invoked from within the KOA system using the following four method calls:

- `setup` - define election parameters such as candidate list and number of seats
- `load` - load all valid ballots and then calculate quota and deposit saving threshold
- `count` - assign votes to candidates, distribute surpluses and exclude candidates until finished
- `report` - report the election results

These methods must be called in the order shown. Only the `report` method is called more than once for each instance of the `ElectionAlgorithm` object.

2.3.5 No Exceptions

Although JML has the facility to specify Java exception handling, no exception handling has been specified, as Java exceptions are not required as part of the design. In particular, Java exceptions are not used for flow of control within the algorithm. Avoiding exceptions simplifies the specification by removing the need to specify exceptions for each method. The design is simplified because the flow of control is simplified by the avoidance of exceptions.

2.3.6 Integration Issues

The original KOA system was designed for use with a party-list system with a single national constituency. The user interface may need to be extended in line with the guidelines for the Irish voting system. The KOA system allows the voter to select a list of candidates. In the Irish system each candidate is in a list of one. The KOA system allows only one selection by the voter. In the Irish system. the voter makes multiple selections in order of preference.

The ASM for KOA is similar to the ASM for the Irish vote counting algorithm. The data structures for votes and candidate lists are similar but not identical. Not all voters will use the Internet, some will use postal votes, paper ballots, voting machines or telephone voting. The system needs to be able to authenticate and load votes from many sources, if required by legislation.

2.4 Design of Java classes used in the specification

The `ElectionAlgorithm` class is central to the design. It contains objects of type `Ballot`, `Candidate`, `Decision`, `BallotBox`, `ElectionDetails` and `ElectionResults`. The following class diagram shows the relationships between the classes. Only a subset of the model fields are shown.

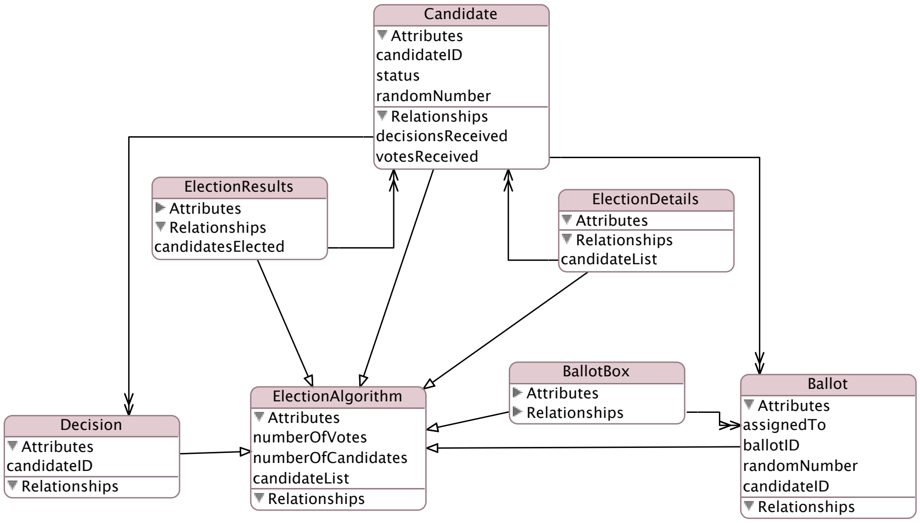


Figure 4. Class Diagram for the Irish Vote Counting Sub-System

2.4.1 ElectionAlgorithm

The `ElectionAlgorithm` class represents the election count process. There is one instance of this object for each constituency.

The count starts when the `count` method is called by the KOA system. It is a requirement that the `setup` and `load` methods have already been called.

The official guidelines suggest that the returning officer be asked to draw lots each time a random selection is required. This is simulated by having random numbers assigned to the candidates, so that process of drawing lots is repeatable for testing purposes. This means that the count results are deterministic for any given set of random numbers, so that the vote count can be audited or repeated.

2.4.2 Candidate

The `Candidate` class represents the status of each candidate and the number of votes received at each count. This is one instance of this object for each candidate. The `Candidate` object records the number of votes received during each round of counting.

Votes can only be added to the candidate's stack while the candidate is neither elected nor excluded. It is the client object, of type `ElectionAlgorithm`, which must ensure that votes added to one candidate do not exceed votes transfered away from another. In particular the `ElectionAlgorithm` object needs to verify that the ballot records agree with the candidate records.

2.4.3 Decision

This object represents any decision which might influence the order in which votes are counted of transfered, especially a random selection between two or more equal candidates. Since the counting of votes is a deterministic process, there is no need to present a user interface each time a decision is made.

2.4.4 Ballot

This object represents a single vote cast by a voter.

A special candidate ID value is used to indicate "non transferable not effective" votes i.e., when the list of preferences has been exhausted and none of the continuing candidates are in the preference list, then the ballot is deemed to be "non transferable not effective".

There must always be a first preference vote in each ballot, otherwise the vote is not included and need not be loaded. The quota is calculated from the number of first preference votes, so that empty ballots are not included.

2.4.5 BallotBox

There is one instance of the `BallotBox` object. It contains the ballot details that are to be loaded by the `load` method of `ElectionAlgorithm`.

2.4.6 ElectionDetails

There is one instance of the `ElectionDetails` object. It contains the candidate details, number of seats and other election parameters. It is loaded by the `setup` method of `ElectionAlgorithm`.

2.4.7 ElectionResults

This object contains the election results returned from the `report` method of `ElectionAlgorithm` to the KOA system.

2.4.8 Overview and Summary

The Java classes described above, which represent a vote counting sub-system for Irish elections, are placed in a Java package `ie.koa.` The overall specification is in the file called `ElectionAlgorithm.jml` in the Appendices. The implementation will be written using Design by Contract, which means writing each method body according to the JML specification and then checking each method with ESC/Java2.

3 Conclusions

The use of JML has enabled the election count algorithm to be specified in a mathematical language without the ambiguities of natural language, while retaining the ability to separate design from implementation. ESC/Java2 acts as a semantic compiler and not just a syntactic compiler which only typechecks the code. The use of '`//@ assert false`' with ESC/Java2 allows the specification to be checked without the write any implementation code until the specification is complete. This should reduce the risk of writing the specification to match the code, rather than changing the code to match the specification.

Clearly many questions remain to be answered in detail. For example, it remains unclear to what extent the drawing of lots and random numbering of votes has or could have on the outcome of an election. The use of electronic voting would allow for fractional transfers of votes instead of rounding up or down the number of votes transfered. This is different than the present manual rules, but it is not clear without further work what the full implications of this would be in practice.

More work is also needed to develop the Java code for this specification and to integrate it with KOA. The user interface of KOA will need to be examined for compliance with Irish law, and/or proposals made for changes to legislation. An end-to-end specification for the complete KOA system is also needed. In addition, the full system needs to be fully validated and checked for system integrity and security.

Finally, it is clear that the relative merits of an Open Source versus a closed system approach need to be fully considered. It would appear that there is a political need to ensure that the source code is open and available for inspection (in order to generate public trust) but that is not the same thing as making it Open Source. Issues like this will also need to be teased out in future work.

Many of the electronic voting systems in use are proprietary systems, the source code for which is not publicly available. The unavailability of source code means that the election results cannot be easily verified unless the votes are printed on paper and counted manually. The inability to audit an election result could invalidate the whole election process.

References

1. Ireland's Department of Environment and Local Government, http://www.environ.ie/elections/dailelect.html#sys

2. Electronic Voting in Ireland Report to the Labour Parliamentary Party Meeting – 16/17 September 2003 By Shane Hogan & Robert Cochran

3. Security of Systems, Kiezen op Afstand, http://www.cs.ru.nl/sos/research/koa/

4. Kiniry, J. (2004), Electronic and Internet Voting in The Netherlands, http://kind.cs.kun.nl/~kiniry/papers/NL_Voting.html

5. The KOA project home page, http://secure.ucd.ie/products/opensource/KOA

6. The Java Modeling Language (JML) home page, http://www.jmlspecs.org/

7. The ESCJava/2 home page, http://secure.ucd.ie/products/opensource/ESCJava2/

8. Department of Environment and Local Government, Count Requirements and Commentary on Count Rules, http://www.cev.ie/htm/tenders/pdf/1_2.pdf

9. Department of Environment and Local Government, Count Requirements and Commentary on Count Rules Update No. 7 dated 14 April 2002, http://www.cev.ie/htm/tenders/pdf/1_4_7.pdf

10. J.C.P. Woodcock, Editors: D. Craigen and K. Summerskill, ‘The Applicability of Formal Methods’, Springer-Verlag, 1990, pages 63–67, ‘Formal Methods for Trustworthy Systems’

11. Bogor Software Model Checking Framework, http://bogor.projects.cis.ksu.edu/

Glossary of Vote Counting Terms and Definitions

These terms are as defined in the Electoral Act 1992 and elsewhere.

Personation

attempting to vote using someone else's identity

First Preference

casting a '1' vote for a candidate

Quota

The minimum number of votes to be normally elected, calculated so that no more people can be elected than there are seats to be filled

Candidate

Person nominated to stand for election, whose names appears of the voting list for selection by voters

Surplus

The number of votes cast for a candidate in excess of the quota

Count

(1) The process of counting votes cast in an election, and determining those candidates to be elected

(2) The number of votes received by a specific candidate

Seats Remaining

The number of vacancies needing to be filled in a multi-seat constituency during an election count

Lowest Continuing Candidates

Those candidates not yet elected or eliminated with the smallest numbers of votes cast for them

Deposit

Amount of money which must be paid by each candidate when being nominated to stand in an election

Exclusion

Removal or elimination of a candidate from further consideration in the count process, due to the low number of votes received

Transferable Votes

Votes cast in a multi-seat constituency are marked '1', '2' etc to indicate the voter's order of preference for candidates. If the first preference cannot be used, due to a candidate receiving either too few votes and being eliminated, or receiving too many votes (i.e. more than the quota, and thus not needed in order to elect the person), then the vote can be transferred to the candidate with the next highest preference marked on the vote.

Ballot

(1) The process of casting votes in an election

(2) A vote

Continuing Candidate

A candidate not yet elected or eliminated

Equal Candidates

Candidates with the same number of votes

Non Transferable Not Effective

Votes which can no longer be used in later stages of a count, due to no further preference being indicated for any continuing candidate

Preference List

The list of candidates in an election for whom a voter has indicated a preference by recording a preference number ('1', '2', etc) against their names

Appendices

A. JML specifications

The Java files are also part of the JML specification in that they define the method signatures to which the JML specification applies.

A1. ElectionAlgorithm.jml

```
package ie.koa;
/**
* This is a Java Modeling Language (JML) design specification the
* vote counting algorithm for Dail elections
*
* @author Dermot Cochran
* @copyright Systems Research Group and
* University College Dublin, Ireland.
* @reviewer Joe Kiniry
*
* @design This JML specification and associated Java code is
* intended to be checkable using ESC/Java2
* (the Extended Static Checker for Java, version 2).
*
*/
public class ElectionAlgorithm {
/**
* Abstract State Machine for Election Algorithm.
*/
/*@ public model byte state;
```

```
@ public invariant EMPTY < SETUP;
@ public invariant SETUP < PRELOAD;
@ public invariant PRELOAD < LOADING;
@ public invariant LOADING < PRECOUNT;
@ public invariant PRECOUNT < COUNTING;
@ public invariant COUNTING < FINISHED;
@ public invariant FINISHED < REPORT;
@ public initially state == EMPTY;
@ public constraint \old (state) <= state;
@ public invariant (state == EMPTY) || (state == SETUP) ||
@   (state == PRELOAD) ||
@   (state == LOADING) || (state == PRECOUNT) ||
@   (state == COUNTING) ||
@   (state == FINISHED) || (state == REPORT);
@*/
/** List of decisions made */
/*@ public model Decision[] decisionsMade;
@ public invariant
@ (\forall int i; 0 <= i && i < numberOfDecisions;
@   decisionsMade[i].decisionTaken != Decision.NODECISION &&
@   decisionsMade[i].atCountNumber < countNumber &&
@   (\exists int k; 0 <= k && k < totalCandidates;
@     decisionsMade[i].candidateID ==
@     candidateList[k].getCandidateID()));
@*/
/** Number of decisions made */
/*@ public model int numberOfDecisions;
@ public initially numberOfDecisions == 0;
@ public invariant 0 <= numberOfDecisions;
@ public constraint
@   \old (numberOfDecisions) <= numberOfDecisions;
@ public constraint (state != COUNTING) ==>
@   numberOfDecisions == \old (numberOfDecisions);
@*/
/** List of details for each candidate.
* @constraint There are no duplicates in the list of candidate
* IDs and, once the counting starts, there must be a ballot paper
* associated with each vote held by a candidate.
*/
//@ public model non_null ie.koa.Candidate[] candidateList;
/*@ public invariant (state == PRELOAD || state == LOADING ||
  @   state == PRECOUNT)
  @ ==>
  @ (\forall int i; 0 <= i && i < totalCandidates;
  @   candidateList[i].getStatus() == Candidate.CONTINUING &&
  @   candidateList[i].getTotalVote() == 0 &&
  @   candidateList[i].getOriginalVote() == 0);
  @
  @ public invariant (state == PRELOAD || state == LOADING ||
  @   state == PRECOUNT ||
  @   state == COUNTING || state == FINISHED || state == REPORT)
  @ ==>
  @   (\forall int i, j;
  @     0 <= i && i < totalCandidates &&
  @     i < j && j < totalCandidates;
  @     candidateList[i].getCandidateID() !=
  @     candidateList[j].getCandidateID());
  @
  @ public invariant (state == PRELOAD || state == LOADING ||
  @   state == PRECOUNT ||
  @   state == COUNTING || state == FINISHED || state == REPORT)
  @ ==>
  @   (\forall int i;
  @     0 <= i && i < totalCandidates;
  @     candidateList[i].getCandidateID() !=
  @     Ballot.NONTRANSFERABLE);
  @
  @ public invariant (state == COUNTING || state == FINISHED ||
  @   state == REPORT)
  @ ==>
  @  (\forall int i; 0 <= i && i < totalCandidates;
  @    candidateList[i].getTotalVote() ==
  @    getNumberOfVotes (candidateList[i].getCandidateID()));
  @*/

/** List of contents of each ballot paper that will be counted. */
//@ public model non_null ie.koa.Ballot[] ballotsToCount;
/*@ public invariant (state >= PRECOUNT)
  @ ==>
  @   (\forall int i, j;
  @     0 <= i && i < totalVotes && i < j && j < totalVotes;
  @     ballotsToCount[i].getBallotID() != ballotsToCount[j].getBallotID());
  @*/
/** Total number of candidates for election */
//@ public model int totalCandidates;
//@ public invariant 0 <= totalCandidates;
//@ public invariant totalCandidates <= candidateList.length;
/*@ public constraint (state >= LOADING) ==>
@ totalCandidates == \old (totalCandidates);
@ public invariant (state == FINISHED) ==> totalCandidates ==
@ numberElected + numberEliminated;
@ public invariant (state == COUNTING) ==> 1 <= totalCandidates;
@*/
/** Number of candidates elected so far */
//@ public model long numberElected;
//@ public invariant 0 <= numberElected;
//@ public invariant numberElected <= seats;
/*@ public invariant (state <= PRECOUNT) ==> numberElected == 0;
  @ public invariant (COUNTING <= state && state <= REPORT)
  @ ==>
  @   numberElected == (\num_of int i; 0 <= i && i < totalCandidates;
  @   isElected(candidateList[i]));
  @ public invariant (state == FINISHED || state == REPORT) ==>
  @   numberElected == seats;
  @ public constraint (state == COUNTING) ==>
  @   \old(numberElected) <= numberElected;
  @*/
/** Number of candidates excluded from election so far */
```

```
//@ public model long numberEliminated;
//@ public invariant 0 <= numberEliminated;
//@ public invariant numberEliminated <= totalCandidates - seats;
/*@ public invariant (state == COUNTING || state == FINISHED ||
  @   state == REPORT)
  @ ==>
  @  numberEliminated == (\num_of int i; 0 <= i && i < totalCandidates;
  @  candidateList[i].getStatus() == Candidate.ELIMINATED);
  @*/

/** Number of seats to be filled in this election */
//@ public model long seats;
//@ public invariant 0 <= seats;
//@ public invariant seats <= totalSeats;
/*@ public constraint (state == LOADING || state == COUNTING) ==>
  @   seats == \old (seats);
  @ public invariant (state == COUNTING) ==> 1 <= seats;
  @*/

/** Total number of seats in this constituency
 * @design The constitution and laws of Ireland do not allow less than three or
 * more than five seats in each Dail constituency, but this could change in
 * future and is not an essential part of the specification.
 */
//@ public model long totalSeats;
//@ public invariant 0 <= totalSeats;
/*@ public constraint (state == LOADING || state == COUNTING) ==>
  @   totalSeats == \old (totalSeats);
  @*/

/** Total number of valid votes in this election */
//@ public model long totalVotes;
//@ public invariant 0 <= totalVotes;
//@ public invariant totalVotes <= ballotsToCount.length;
/*@ public invariant (state == EMPTY || state == SETUP ||
  @   state == PRELOAD)
  @ ==>
  @   totalVotes == 0;
  @ public constraint (state == LOADING)
  @ ==>
  @   \old (totalVotes) <= totalVotes;
  @ public constraint (state == PRECOUNT || state == COUNTING ||
  @   state == FINISHED || state == REPORT)
  @ ==>
  @  totalVotes == \old (totalVotes);
  @*/

/** Number of votes so far which did not have a transfer to
 * a continuing candidate */
//@ public model long nonTransferableVotes;
//@ public invariant 0 <= nonTransferableVotes;
//@ public invariant nonTransferableVotes <= totalVotes;
/*@ public invariant (state == COUNTING || state == FINISHED ||
  @   state == REPORT)
  @ ==> nonTransferableVotes ==
  @   (\num_of int i; 0 <= i && i < totalVotes;
  @     ballotsToCount[i].candidateID == Ballot.NONTRANSFERABLE);
  @*/

/** Minimum number of votes needed to guarantee election */
//@ public model long quota;
//@ public invariant 0 <= quota;
//@ public invariant quota <= totalVotes;
/*@ public invariant (state == COUNTING) ==>
  @   quota == 1 + (totalVotes / (seats + 1));
  @*/

/** Minimum number of votes needed to save deposit unless elected */
//@ public model long depositSavingThreshold;
//@ public invariant 0 <= depositSavingThreshold;
//@ public invariant depositSavingThreshold <= totalVotes;
/*@ public invariant (state == COUNTING)
  @ ==> depositSavingThreshold ==
  @   (((totalVotes / (totalSeats + 1)) + 1) / 4) + 1;
  @*/

/** Number of rounds of counting so far */
//@ public model int countNumber;
//@ public initially countNumber == 0;
//@ public invariant 0 <= countNumber;
//@ public invariant countNumber < Candidate.MAXCOUNT;
/*@ public constraint (state == COUNTING) ==>
  @   \old(countNumber) <= countNumber;
  @
  @ public constraint (state == COUNTING) ==>
  @   countNumber <= \old (countNumber) + 1;
  @*/

/** Number of candidates with surplus votes */
//@ public model long numberOfSurpluses;
//@ public invariant 0 <= numberOfSurpluses;
//@ public invariant numberOfSurpluses <= numberElected;

/** Total number of undistributed surplus votes */
//@ public model long sumOfSurpluses;
//@ public invariant 0 <= sumOfSurpluses;
//@ public invariant sumOfSurpluses <= totalVotes;
/*@ invariant (state == COUNTING)
  @ ==> sumOfSurpluses == (\sum int i;
  @   0 <= i && i < totalCandidates; getSurplus(candidateList[i]));
  @*/

/** Number of seats which remain to be filled */
//@ public model long remainingSeats;
//@ public invariant 0 <= remainingSeats;
//@ public invariant remainingSeats <= seats;
/*@ public invariant (state <= PRECOUNT) ==>
  @   remainingSeats == seats;
  @ public invariant (state == FINISHED || state == REPORT) ==>
  @   remainingSeats == 0;
  @ public invariant (state == COUNTING) ==>
  @   remainingSeats == (seats - numberElected);
  @*/

/** Number of candidates neither elected nor excluded from election */
//@ public model long numberOfContinuingCandidates;
//@ public invariant 0 <= numberOfContinuingCandidates;
/*@ public invariant
  @   numberOfContinuingCandidates <= totalCandidates;
  @
  @ public invariant (state == FINISHED) ==>
  @   numberOfContinuingCandidates == 0;
  @ invariant (state == COUNTING) ==>
  @   numberOfContinuingCandidates ==
  @   (totalCandidates - numberElected) - numberEliminated;
  @*/

/** There must be at least one continuing candidate for each remaining seat
 * @see requirement 11, section 4, item 4, page 16
 */
/*@ invariant (state == COUNTING) ==>
  @   remainingSeats <= numberOfContinuingCandidates;
  @*/

/** The lowest non-zero number of votes held by a continuing candidate
*/
//@ public model int lowestContinuingVote;
//@ public invariant 0 < lowestContinuingVote;
/*@ public invariant
  @   (\exists int k; 0 <= k && k < totalCandidates;
  @     candidateList[k].getStatus() == Candidate.CONTINUING &&
  @     0 < candidateList[k].getTotalVote())
  @ ==> lowestContinuingVote ==
  @   (\min int i; 0 <= i && i < totalCandidates &&
  @     candidateList[i].getStatus() == Candidate.CONTINUING &&
  @     0 < candidateList[i].getTotalVote();
  @     candidateList[i].getTotalVote());
  @*/

/** The second lowest non-zero number of votes held by a continuing
candidate */
//@ public model int nextHighestVote;
//@ public invariant lowestContinuingVote < nextHighestVote;
/*@ public invariant
  @   (\exists int k; 0 <= k && k < totalCandidates;
  @     candidateList[k].getStatus() == Candidate.CONTINUING &&
  @     lowestContinuingVote < candidateList[k].getTotalVote())
  @ ==> nextHighestVote ==
  @   (\min int i; 0 <= i && i < totalCandidates &&
  @     candidateList[i].getStatus() == Candidate.CONTINUING &&
  @     lowestContinuingVote < candidateList[i].getTotalVote();
  @     candidateList[i].getTotalVote());
  @*/

/** The highest number of votes held by a continuing candidate */
//@ public model long highestContinuingVote;
//@ public invariant highestContinuingVote < quota;
/*@ public invariant (0 < numberOfContinuingCandidates)
  @ ==> highestContinuingVote ==
  @   (\max int i; 0 < i && i < totalCandidates &&
  @     candidateList[i].getStatus() == Candidate.CONTINUING;
  @     candidateList[i].getTotalVote());
  @*/

/** Highest available surplus for distribution */
//@ public model long highestSurplus;
//@ public invariant 0 <= highestSurplus;
//@ public invariant highestSurplus <= sumOfSurpluses;
/*@ invariant (state == COUNTING) ==> highestSurplus ==
  @   (\max int i; 0 < i && i < totalCandidates;
  @   getSurplus(candidateList[i]));
  @*/

/** Sum of continuing votes other than the highest */
//@ public model long sumOfOtherContinuingVotes;
//@ public invariant 0 <= sumOfOtherContinuingVotes;
//@ public invariant sumOfOtherContinuingVotes <= totalVotes;
/*@ invariant (state == COUNTING) ==> sumOfOtherContinuingVotes ==
  @ (\sum int i; 0 <= i && i < totalCandidates &&
  @ candidateList[i].getStatus() == Candidate.CONTINUING
  @ && candidateList[i].getTotalVote() < highestContinuingVote;
  @ candidateList[i].getTotalVote());
  @*/

/** Number of candidates with equal highest continuing votes */
//@ public model long numberOfEqualHighestContinuing;
//@ public invariant 0 <= numberOfEqualHighestContinuing;
/*@ public invariant
@ numberOfEqualHighestContinuing <= numberOfContinuingCandidates;
@ public invariant (state == COUNTING) ==>
numberOfEqualHighestContinuing ==
@ (\num_of int i; 0 <= i && i < totalCandidates &&
@ candidateList[i].getStatus() == Candidate.CONTINUING;
@ candidateList[i].getTotalVote() == highestContinuingVote);
@*/
/** Number of candidates with equal lowest non-zero votes */
//@ public model long numberOfEqualLowestContinuing;
//@ public invariant 0 <= numberOfEqualLowestContinuing;
/*@ public invariant
@ numberOfEqualLowestContinuing <= numberOfContinuingCandidates;
@ public invariant (state == COUNTING) ==>
numberOfEqualLowestContinuing ==
@ (\num_of int i; 0 <= i && i < totalCandidates &&
@ candidateList[i].getStatus() == Candidate.CONTINUING;
@ candidateList[i].getTotalVote() == lowestContinuingVote);
@*/
```

```
/**
* Determine if the candidate has received enough votes for election
*
* @param candidate This candidate
* @return True if the candidate has at least a quota of votes
* @see "http://www.cev.ie/htm/tenders/pdf/1_1.pdf, page 79, paragraph
120 (2)"
*/
/*@ requires candidate != null;
@ requires countNumber >= 1;
@ requires state == COUNTING;
@ ensures \result == (candidate.getTotalVote() >= quota);
@*/
protected boolean hasQuota (ie.koa.Candidate candidate);
/**
* Determine if the candidate has been elected
*
* @design It is possible for a candidate without having reached the quota
* to be elected in the final round of counting by virtue of being the
* highest continuing candidate when one seat remains.
*
*
* <p> The quota is the minimum number of votes needed for election, except
* when any of the following shortcuts apply.
* <ul>
* <li>The number of continuing candidates is equal to the number of remaining
* seats.
* <li>The number of continuing candidates is one more than the number of
* remaining seats.
* <li>There is one remaining seat.
* </ul>
*
* @see <a href="http://www.cev.ie/htm/tenders/pdf/1_1.pdf">Department
* of Environment and Local Government, Electronic Vote Counting System,
* Appendix E</a>
* @see <a href="http://www.cev.ie/htm/tenders/pdf/1_2.pdf">Department
* of Environment and Local Government, Count Requirements and Commentary on
* Count Rules, sections 3-14</a>
* @see <a href="http://www.irishstatutebook.ie/1992_23.html">Sections
* 48, 118 and 124 of the Electoral Act, 1992</a>
*
* @param candidate This candidate
* @return True if the candidate has been elected
*/
/*@ requires candidate != null;
@ requires countNumber >= 1;
@ requires state == COUNTING;
@ ensures (\result == true) <==>
@ (candidate.getStatus() == Candidate.ELECTED || hasQuota
(candidate));
@*/
protected boolean isElected (ie.koa.Candidate candidate);
/**
* Determine how many surplus votes a candidate has
```

```
*
* @design The surplus is the maximum number of votes available for transfer
* @param candidate The candidate record
* @return The undistributed surplus for that candidate, or zero if the
* candidate has less than a quota of votes
* @see < a href="http://www.cev.ie/htm/tenders/pdf/1_1.pdf">Department
* of Environment and Local Government, Electronic Voting and Counting
* System, Appendix E</a>
*/
/*@
@ requires candidate != null;
@ requires countNumber > 1;
@ ensures (hasQuota(candidate) == true) ==> \result ==
@ (candidate.getTotalVote() - quota);
@ ensures (hasQuota(candidate) == false) ==> \result == 0;
@ ensures \result >= 0;
@*/
protected int getSurplus (ie.koa.Candidate candidate);
/**
* Determines if a candidate has saved his or her deposit.
*
* @design The deposit saving threshold is one plus one quarter of the full
* quota.<p>
* It is possible for a candidate without the deposit saving threshold
* to be elected in the final round of counting by virtue of being the
* highest continuing candidate when one seat remains. This could happen,
* for example, if the majority of ballots contained only first preferences.
*
* @param candidate This candidate
* @return <code>true</code> if candidate has had enough votes to save deposit
* or has been elected
*/
/*@ requires (state == COUNTING) || (state == FINISHED) || (state == REPORT);
@ ensures \result == (candidate.getOriginalVote() >= depositSavingThreshold) ||
@ (isElected (candidate) == true);
@*/
protected boolean isDepositSaved (/*@ non_null @*/ ie.koa.Candidate
candidate);
/**
* Distribution of surplus votes
*
* @param candidateWithSurplus The candidate whose surplus is to be distributed
* @design The highest surplus must be distributed if the total surplus
* could save the deposit of a candidate or change the relative position
* of the two lowest continuing candidates, or would be enough to elect the
* highest continuing candidate.
* @see "http://www.cev.ie/htm/tenders/pdf/1_2.pdf, sections 5 and 11"
*/
/*@ requires getSurplus (candidateWithSurplus) > 0;
@ requires state == COUNTING;
@ requires numberOfContinuingCandidates > remainingSeats;
@ requires (numberOfContinuingCandidates > remainingSeats + 1) ||
@ (sumOfSurpluses + lowestContinuingVote > nextHighestVote) ||
@ (numberOfEqualLowestContinuing > 1);
@ requires remainingSeats > 0;
@ requires (remainingSeats > 1) ||
@ ((highestContinuingVote < sumOfOtherContinuingVotes + sumOfSurpluses) &&
@ (numberOfEqualHighestContinuing == 1));
@ requires getSurplus (candidateWithSurplus) == highestSurplus;
@ requires (sumOfSurpluses + highestContinuingVote >= quota) ||
@ (sumOfSurpluses + lowestContinuingVote > nextHighestVote) ||
@ (numberOfEqualLowestContinuing > 1) ||
@ ((sumOfSurpluses + lowestContinuingVote >= depositSavingThreshold) &&
@ (lowestContinuingVote < depositSavingThreshold));
@ ensures getSurplus (candidateWithSurplus) == 0;
@ ensures countNumber == \old (countNumber) + 1;
@ ensures (state == COUNTING) || (state == FINISHED);
@ ensures totalVotes == nonTransferableVotes +
@ (\sum int i; 0 <= i && i < totalCandidates;
@ candidateList[i].getTotalVote());
@*/
protected void distributeSurplus(/*@ non_null @*/
ie.koa.Candidate candidateWithSurplus);

/**
* Elimination of one or more candidates and transfer of votes
*
* @param candidate This candidate
*
* @design More than one candidate could be eliminated in the same round if the
* combined vote held by the group is not enough to elect a candidate or to
* save a deposit, and if there are no surplus votes available for distribution.
*
* @see <a href="http://www.cev.ie/htm/tenders/pdf/1_2.pdf">Count Requirements
* and Commentary on Count Rules, section 10</a>
* @see <a href="http://www.cev.ie/htm/tenders/pdf/1_4_7.pdf">Update No. 7 dated
* 14 April 2002; Available surpluses and candidates with zero votes</a>
*/
/*@ requires 1 <= numberToEliminate;
@ requires numberToEliminate <= numberOfContinuingCandidates;
@ requires (\forall int i;
@   0 <= i && i < numberToEliminate;
@   candidatesToEliminate[i].getTotalVote() == 0 ||
@   depositSavingThreshold <= candidatesToEliminate[i].getTotalVote() ||
@   candidatesToEliminate[i].getTotalVote() +
@   sumOfSurpluses + (\sum int j;
@   0 <= j && j != i && j < numberToEliminate;
@   candidatesToEliminate[i].getTotalVote()) < depositSavingThreshold);
@ requires (\forall int i;
@   0 <= i && i < numberToEliminate;
@   candidatesToEliminate[i].getStatus() == Candidate.CONTINUING);
@ requires sumOfSurpluses + (\sum int i;
@   0 <= i && i < numberToEliminate;
@   candidatesToEliminate[i].getTotalVote()) < quota;
@ requires remainingSeats < numberOfContinuingCandidates;
@ requires (state == COUNTING);
@ ensures (\forall int i;
@   0 <= i && i < numberToEliminate;
@   candidatesToEliminate[i].getStatus() == Candidate.ELIMINATED &&
@   candidatesToEliminate[i].getTotalVote() == 0);
@ ensures numberEliminated == \old (numberEliminated) + numberToEliminate;
@ ensures remainingSeats <= numberOfContinuingCandidates;
@ ensures numberElected <= seats;
@ ensures \old(lowestContinuingVote) <= lowestContinuingVote;
@*/
protected void eliminateCandidates(
  /*@ non_null @*/ Candidate[] candidatesToEliminate,
  long numberToEliminate);

/**
* Declare election results
*/
/*@ requires state == FINISHED;
@ assignable state;
@ ensures state == REPORT;
@*/
public ElectionResults report();
/**
* Set up candidate details and number of seats
*/
/*@ requires state == EMPTY;
@ ensures state == PRELOAD;
@ ensures totalCandidates == electionDetails.numberOfCandidates;
@ ensures seats == electionDetails.numberOfSeatsInThisElection;
@ ensures totalSeats == electionDetails.totalNumberOfSeats;
@ assignable state, seats, totalSeats, totalCandidates, candidateList;
@*/
public void setup (ElectionDetails electionDetails);

/** Loads all valid ballot papers */
// @design All ballot papers must be assigned to a valid candidate ID
/*@ requires state == PRELOAD;
@ assignable state, totalVotes, ballotsToCount, quota,depositSavingThreshold;
@ ensures state == PRECOUNT;
@ ensures totalVotes == ballotBox.numberOfBallots;
@ ensures (\forall int i; 0 <= i && i < totalVotes;
@ (\exists int j; 0 <= j && j < totalCandidates;
@ ballotsToCount[j].isAssignedTo(candidateList[i].getCandidateID())));
@*/
public void load(BallotBox ballotBox);

/**
* Count the votes
*/
/*@ requires remainingSeats == seats;
@ requires numberElected == 0;
@ requires numberEliminated == 0;
@ requires nonTransferableVotes == 0;
@*/
/** @see requirement 1, section 3, item 2, page 12 */
//@ requires state == PRECOUNT;
/*@
```

```
@ requires numberOfContinuingCandidates == totalCandidates;
@ requires countNumber == 0;
@ ensures state == FINISHED;
@ assignable state, countNumber, numberElected, numberEliminated,
@ numberOfContinuingCandidates, remainingSeats,
@ numberOfSurpluses, candidateList,
@ sumOfSurpluses, highestContinuingVote, lowestContinuingVote,
@ nextHighestVote, nonTransferableVotes, ballotsToCount,
@ numberOfContinuingCandidates;
@ ensures remainingSeats == 0;
@ ensures numberElected == seats;
@ ensures numberEliminated == totalCandidates - numberElected;
@ ensures numberOfContinuingCandidates == 0;
@*/
public void count();

/**
* Get the status of the algorithm in progress
*/
/*@ ensures \result == state;
@*/
public /*@ pure @*/ byte getStatus();

/**
* Gets the current number of votes for this candidate ID
*
* @design This method an also be used to check the number of nontransferable
* votes.
*
* @param candidateID Candidate ID for which to check the votes.
* @return Number of votes currently assigned to this candidate
*/
/*@ requires (state == COUNTING || state == FINISHED || state == REPORT);
@ requires 0 < candidateID || candidateID == Ballot.NONTRANSFERABLE;
@ ensures \result== (\num_of int j; 0 <= j && j < totalVotes;
@ ballotsToCount[j].isAssignedTo(candidateID));
@*/
protected /*@ pure @*/ long getNumberOfVotes (long candidateID);

/**
* Gets the potential number of transfers from one candidate to another.
*
* @design This method is needed to get the proportions to use when transferring
* surplus votes. If the candidate was elected on the first count then all
* votes are examined, otherwise only the last set of votes received are
* examined.
*
* @param fromCandidate Candidate ID from which to check the transfers
* @param toCandidateID Candidate ID to check for receipt of transfered votes
* @return Number of votes potentially transferable from this candidate to that
* candidate
*/
/*@ requires (state == COUNTING);
@ requires 0 < toCandidateID;
@ requires toCandidateID != Ballot.NONTRANSFERABLE;
@ ensures \result== (\num_of int j; 0 <= j && j < totalVotes;
@ (ballotsToCount[j].isAssignedTo(fromCandidate.getCandidateID())) &&
@ (ballotsToCount[j].getCountNumberAtLastTransfer() ==
@ fromCandidate.getLastSetAddedCountNumber()) &&
@ (getNextContinuingPreference(ballotsToCount[j]) == toCandidateID));
@*/
protected /*@ pure @*/ int getPotentialTransfers (
/*@ non_null @*/ Candidate fromCandidate, long toCandidateID);

/**
* Get the maximum number of votes transferable to continuing candidates.
* @param fromCandidate Candidate from which to check the transfers
* @return Number of votes potentially transferable from this candidate
*/
/*@ requires (state == COUNTING);
@ ensures \result == (\sum int i; 0 <= i && i < totalCandidates;
@ getPotentialTransfers (fromCandidate, candidateList[i].getCandidateID()));
@*/
protected /*@ pure @*/ long getTotalTransferableVotes (
/*@ non_null @*/ Candidate fromCandidate);

/**
* Gets the next preference continuing candidate.
*
* @design This is the _nearest_ next preference i.e.
* filter the list of preferences to contain continuing candidates and then
* get the next preference to a continuing candidate, if any.
*
* @param ballot Ballot paper from which to get the next preference
*
* @return Candidate ID of next continuing candidate or NONTRANSFERABLE
*/
/*@ requires state == COUNTING;
@ ensures (\result == Ballot.NONTRANSFERABLE) <=!=>
@ (\exists int k; 1 <= k && k <= ballot.remainingPreferences();
@ (\result == ballot.getNextPreference(k)) &&
@ (\forall int i; 1 <= i && i < k;
@ isContinuingCandidateID(ballot.getNextPreference(i)) == false)
@ );
@*/
protected long getNextContinuingPreference(/*@ non_null @*/ Ballot
ballot);

/**
* Determine actual number of votes to transfer to this candidate, excluding
* rounding up of fractional transfers
*
* @requirement 25 from section 7 item 3.2 on page 25
*
* @design The votes in a surplus are transfered in proportion to
* the number of transfers available throughout the candidates ballot stack.
* The calculations are made using integer values because there is no concept
* of fractional votes or fractional transfer of votes, in the existing manual
* counting system. If not all transferable votes are accounted for the
* highest remainders for each continuing candidate need to be examined.
*
* @param fromCandidate Candidate from which to count the transfers
* @param toCandidate Continuing candidate eligible to receive votes
* @return Number of votes to be transfered, excluding fractional transfers
*/
/*@ requires (state == COUNTING);
@ requires (fromCandidate.getStatus() == Candidate.ELECTED) ||
@ (fromCandidate.getStatus() == Candidate.ELIMINATED);
@ requires toCandidate.getStatus() == Candidate.CONTINUING;
@ ensures ((fromCandidate.getStatus() == Candidate.ELECTED) &&
@ (getSurplus(fromCandidate) < getTotalTransferableVotes(fromCandidate)))
@ ==>
@ (\result ==
@ (getSurplus (fromCandidate) *
@ getPotentialTransfers (fromCandidate,
@ toCandidate.getCandidateID()) /
@ getTotalTransferableVotes (fromCandidate)));
@ ensures ((fromCandidate.getStatus() == Candidate.ELIMINATED) ||
@ (getTotalTransferableVotes(fromCandidate) <= getSurplus(fromCandidate)))
@ ==>
@ (\result ==
@ (\num_of int j; 0 <= j && j < totalVotes;
@ ballotsToCount[j].isAssignedTo(fromCandidate.getCandidateID()) &&
@ getNextContinuingPreference(ballotsToCount[j]) ==
@ toCandidate.getCandidateID()));
@*/
protected /*@ pure @*/ int getActualTransfers (
/*@ non_null @*/ Candidate fromCandidate,
/*@ non_null @*/ Candidate toCandidate);
/**
* Determine the rounded value of a fractional transfer.
*
* @design This depends on the shortfall and the relative size of the other
* fractional transfers.
*
* @param fromCandidate
* Elected candidate from which to distribute surplus
*
* @param toCandidate
* Continuing candidate potentially eligible to receive transfers
*
* @return <code>1</code> if the fractional vote is to be rounded up
* <code>0</code> if the fractional vote is to be rounded down
*/
/*@ requires state == COUNTING;
@ requires isElected (fromCandidate);
@ requires toCandidate.getStatus() == ie.koa.Candidate.CONTINUING;
@ requires getSurplus(fromCandidate) < getTotalTransferableVote(fromCandidate);
@ ensures (getCandidateOrderByHighestRemainder (fromCandidate,toCandidate) <=
@ getTransferShortfall (fromCandidate))
@ ==> \result == 1;
@ ensures (getCandidateOrderByHighestRemainder (fromCandidate,toCandidate) >
@ getTransferShortfall (fromCandidate))
@ ==> \result == 0;
@*/
protected /*@ pure @*/ long getRoundedFractionalVote (
/*@ non_null @*/ Candidate fromCandidate,
/*@ non_null @*/ Candidate toCandidate);
/**
* Determine the number of continuing candidates with a higher remainder in
* their transfer quotient, or deemed to have a higher remainder.
*
* @design There must be a strict ordering of candidates for the purpose of
* allocating the transfer shortfall. If two or more candidates have equal
* remainders then use the number of transfers they are about to receive and if
* the number of transfers are equal then look at the number of votes already
* received.
*
* @param fromCandidate
* Elected candidate from which to distribute surplus
*
* @param toCandidate
* Continuing candidate potentially eligible to receive transfers
*
* @return The number of continuing candidates with a higher quotient remainder
* than this candidate
*/
/*@ requires state == COUNTING;
@ requires isElected (fromCandidate);
@ requires toCandidate.getStatus() == ie.koa.Candidate.CONTINUING;
@ requires getSurplus(fromCandidate) < getTotalTransferableVote(fromCandidate);
@ ensures \result == (\num_of int i; i <= 0 && i < totalCandidates &&
@ candidateList[i].getCandidateID() != toCandidate.getCandidateID()&&
@ candidateList[i].getStatus() == ie.koa.Candidate.CONTINUING;
@ (getTransferRemainder(fromCandidate, candidateList[i]) >
@ getTransferRemainder(fromCandidate, toCandidate)) ||
@ ((getTransferRemainder(fromCandidate, candidateList[i]) ==
@ getTransferRemainder(fromCandidate, toCandidate)) &&
@ (getActualTransfers(fromCandidate, candidateList[i]) >
@ getActualTransfers(fromCandidate, toCandidate))) ||
@ ((((getTransferRemainder(fromCandidate, candidateList[i]) ==
@ getTransferRemainder(fromCandidate, toCandidate)) &&
@ (getActualTransfers(fromCandidate, candidateList[i]) ==
@ getActualTransfers(fromCandidate, toCandidate)))) &&
@ isHigherThan (candidateList[i], toCandidate)));
@*/
protected /*@ pure @*/ long getCandidateOrderByHighestRemainder (
/*@ non_null @*/ Candidate fromCandidate,
/*@ non_null @*/ Candidate toCandidate);
/**
* Determine if one continuing candidate is higher than another, for the purpose
* of resolving remainders of transfer quotients.
*
* @design This is determined by finding the earliest round of counting in which
* these candidates had unequal votes. If both candidates are equal at all
* counts then random numbers are used to draw lots.
```

```
*
* @see <a href="http://www.cev.ie/htm/tenders/pdf/1_2.pdf">Department of
* Environment and Local Government, Count Requirements and Commentary on Count
* Rules, section 7, page 25</a>
*
* @param firstCandidate
* The first of the two candidates to be compared
*
* @param secondCandidate
* The second of the two candidates to be compared
*
* @return <code>true</code> if first candidate is deemed to have received more
* votes than the second.
*/
/*@ requires firstCandidate.getStatus() == Candidate.CONTINUING;
@ requires secondCandidate.getStatus() == Candidate.CONTINUING;
@ ensures \result == (\exists int i; 0 <= i && i < countNumber;
@ (firstCandidate.getVoteAtCount(i) > secondCandidate.getVoteAtCount(i)) &&
@ (\forall int j; 0 <= j && j < i;
@ firstCandidate.getVoteAtCount(j) == secondCandidate.getVoteAtCount(j))) ||
@ ((randomSelection (firstCandidate, secondCandidate) ==
@ firstCandidate.getCandidateID()) &&
@ (\forall int k; 0 <= k && k < countNumber;
@ firstCandidate.getVoteAtCount(k) == secondCandidate.getVoteAtCount(k)));
@*/
public /*@ pure @*/ boolean isHigherThan (
/*@ non_null @*/ Candidate firstCandidate,
/*@ non_null @*/ Candidate secondCandidate);
/**
* Draw lots to choose between two continuing candidates.
*
* @design The official guidelines suggest that the returning officer
* be asked to draw lots each time a random selection is required. This
* is simulated by having random numbers assigned to the candidates, so that
* process of drawing lots is repeatable for testing purposes. <p>
* This means that the count results are deterministic for any given set of
* random numbers.
*
* @param firstCandidate
* The first of the two candidates to be compared
*
* @param secondCandidate
* The second of the two candidates to be compared
*
* @return The candidate ID of the chosen candidate
*/
/*@ requires state == COUNTING;
@ requires firstCandidate.getStatus() == Candidate.CONTINUING;
@ requires secondCandidate.getStatus() == Candidate.CONTINUING;
@ requires firstCandidate.randomNumber != secondCandidate.randomNumber;
@ ensures (\result == firstCandidate.candidateID) <==>
@ (firstCandidate.randomNumber < secondCandidate.randomNumber);
@ ensures (\result == secondCandidate.candidateID) <==>
@ (secondCandidate.randomNumber < firstCandidate.randomNumber);
```

```
@*/
public /*@ pure @*/ long randomSelection (
/*@ non_null @*/ Candidate firstCandidate,
/*@ non_null @*/ Candidate secondCandidate);
/**
* Determine the indivisible remainder after integer division by the transfer
* factor for surpluses.
*
* @design This can all be done with integer arithmetic; no need to use
* floating point numbers, which could introduce rounding errors.
*
* @param fromCandidate Elected candidate from which to count the transfers
* @param toCandidate Continuing candidate eligible to receive votes
*
* @return The size of the quotient remainder
*/
/*@ requires state == COUNTING;
@ requires isElected (fromCandidate);
@ requires toCandidate.getStatus() == ie.koa.Candidate.CONTINUING;
@ requires getSurplus(fromCandidate) <
@   getTotalTransferableVotes(fromCandidate);
@ requires 0 <= getTransferShortfall (fromCandidate);
@ ensures \result ==
@ getPotentialTransfers(fromCandidate, toCandidate.getCandidateID()) -
@ ((getActualTransfers(fromCandidate, toCandidate) *
@ getTotalTransferableVotes (fromCandidate)) /
@ getSurplus(fromCandidate));
@*/
protected /*@ pure @*/ long getTransferRemainder (
/*@ non_null @*/ Candidate fromCandidate,
/*@ non_null @*/ Candidate toCandidate);

/**
* Determine shortfall between sum of transfers rounded down and the size of
* surplus.
*
* @param fromCandidate
* Elected candidate from which to distribute surplus
*
* @return The shortfall between the sum of the transfers and the size
* of surplus
*/
/*@ requires state == COUNTING;
@ requires isElected (fromCandidate);
@ requires getSurplus(fromCandidate) <
@   getTotalTransferableVotes(fromCandidate);
@ ensures \result == getSurplus (fromCandidate) -
@ (\sum int i; 0 <= i && i < totalCandidates &&
@ candidateList[i].getStatus() == Candidate.CONTINUING;
@ getActualTransfers (fromCandidate, candidateList[i]));
@*/
protected /*@ pure @*/ long getTransferShortfall (
/*@ non_null @*/ Candidate fromCandidate);
```

```
/**
* Determine if a candidate ID belongs to a continuing candidate.
*
* @param candidateID
* The ID of candidate for which to check the status
*
* @return <code>true</code> if this candidate ID matches that of a
* continuing candidate
*/
/*@ requires 0 < candidateID;
@ ensures \result == (\exists int i;
@ 0 <= i && i < totalCandidates;
@ candidateID == candidateList[i].getCandidateID() &&
@ candidateList[i].getStatus() == Candidate.CONTINUING);
*/
public /*@ pure @*/ boolean isContinuingCandidateID (long candidateID);

/**
* List each ballot ID in order by random number used to show how the votes have
* been mixed and numbered.
*
* @param ballot Ballot for which to get order of
* @return Order of this ballot in numbered list of ballots
*/
/*@
@ requires state == REPORT;
@ ensures 1 <= \result;
@ ensures \result <= ballotsToCount.length;
@ ensures (\forall Ballot a, b; a != null && b != null;
@ (getOrder (a) < getOrder (b)) <==> (b.isAfter(a)));
@*/
protected /*@ pure @*/ long getOrder(/*@ non_null @*/ Ballot ballot);

/**
* List each candidate ID in order by random number to show how lots would have
* been chosen.
*
* @param candidate Candidate for which to get the order of
* @return Order of this candidate for use when lots are chosen
*/
/*@
@ requires state == REPORT;
@ ensures 1 <= \result;
@ ensures \result <= candidateList.length;
@ ensures (\forall Candidate c, d; c != null && d != null;
@ (getOrder (c) < getOrder (d)) <==> (d.isAfter(c)));
@*/
public /*@ pure @*/ long getOrder(
  /*@ non_null @*/ Candidate candidate);

/**
* Transfer votes from one candidate to another.
* @param fromCandidate Elected or excluded candidate
* @param toCandidate Continuing candidate
* @param numberOfVotes Number of votes to be transfered
*/
/*@ requires fromCandidate.getStatus() != Candidate.CONTINUING;
@ requires toCandidate.getStatus() == Candidate.CONTINUING;
@ requires numberOfVotes == getActualTransfers (fromCandidate,toCandidate) +
@ getRoundedFractionalVote (fromCandidate, toCandidate);
@ ensures fromCandidate.getTotalVote() ==
@ \old (fromCandidate.getTotalVote()) - numberOfVotes;
@ ensures toCandidate.getTotalVote() ==
@ \old (toCandidate.getTotalVote()) + numberOfVotes;
@*/
protected void transferVotes(
/*@ non_null @*/ Candidate fromCandidate,
/*@ non_null @*/ Candidate toCandidate,
long numberOfVotes);

/**
* Complete one round of counting.
*
*/
/*@ requires state == COUNTING;
@
@*/
protected void completeRound();

/**
* Update list of decision events.
*/
/*@ requires state == COUNTING;
@ ensures (\forall int i; 0 <= i && i < totalCandidates;
@ isElected (candidateList[i]) ==> (\exists int k;
@ 0 <= k && k < numberOfDecisions;
@ (decisionsMade[k].candidateID == candidateList[i].getCandidateID()) &&
@ ((decisionsMade[k].decisionTaken == Decision.ELECTBYQUOTA) ||
@ (decisionsMade[k].decisionTaken == Decision.DEEMELECTED))) &&
@ (\forall int n; 0 <= n && n < numberOfDecisions;
@ (decisionsMade[n].candidateID == candidateList[i].getCandidateID())
@ ==> (decisionsMade[n].decisionTaken != Decision.EXCLUDE))
@ );
@*/
protected void updateDecisions();
}
```

B. JML annotations to Java files

B1. ElectionAlgorithm.java

```
package ie.koa;
/**
* Vote counting algorithm for elections to D‡il Eireann.
```

```
*
* @author Dermot Cochran
* @copyright Systems Research Group and University College Dublin, Ireland.
* @reviewer Joe Kiniry
*
* @design This JML specification and associated Java code is intended
* to be verifiable using the Extended Static Checking for Java
* version 2 tool (ESCJava2). <p>
* This Java package <code>ie.koa</code> is designed to be used as part of the
* KOA remote voting system, but does not assume anything about the KOA system
* other than that it takes care of the remote voting process, supplies a valid
* set of ballots and candidate IDs to be counted by this algorithm and takes
* care of system level issues such as security, authentication and data
storage.
*
* @see <a href="http://www.irishstatutebook.ie/1992_23.html">Part XIX of the
* Electoral Act, 1992</a>
* @see <a href="http://www.cev.ie/htm/tenders/pdf/1_2.pdf">Department of
* Environment and Local Government: Count Requirements and Commentary on Count
* Rules, sections 3-16</a>
* @see <a href="http://secure.ucd.ie/products/opensource/KOA/">The KOA Remote
* Voting System</a>
* @see <a href="http://secure.ucd.ie/products/opensource/ ESCJava2/">ESCJava/2
* Homepage</a>
* @see <a href="http://www.jmlspecs.org/">JML Homepage</a>
*/
//@ refine "ElectionAlgorithm.jml";
public class ElectionAlgorithm {
/**
* @design The election count algorithm is modeled as an abstract state
* machine with states and transitions between those states:
*
* <p> The normal path is:
* <p> EMPTY --> SETUP --> PRELOAD --> LOADING -->
* PRECOUNT --> COUNTING --> FINISHED --> REPORT
*
*/
protected /*@ spec_public @*/ byte status; //@ in state;
//@ public represents state <- status;
/*@ public invariant status == EMPTY || status == SETUP || status == PRELOAD ||
@ status == LOADING || status == PRECOUNT || status == COUNTING ||
@ status == FINISHED || status == REPORT;
@*/
/** Start state */
public static final byte EMPTY = 0;
/** Set up candidate list */
public static final byte SETUP = 1;
/** Ready to load ballots */
public static final byte PRELOAD = 2;
/** Load all valid ballots */
public static final byte LOADING = 3;
/** Ready to count votes */
public static final byte PRECOUNT = 4;
/** Count the votes */
```

```
public static final byte COUNTING = 5;
/** Finished counting */
public static final byte FINISHED = 6;
/** Declare election result */
public static final byte REPORT = 7;

/** Number of seats in this election */
protected long numberOfSeats;
//@ protected represents seats <- numberOfSeats;

/** Number of seats in this constituency */
protected long totalNumberOfSeats;
//@ protected represents totalSeats <- totalNumberOfSeats;

/** Total number of valid ballot papers */
protected int totalNumberOfVotes;
final static int MAXVOTES = 999999;
//@ protected represents totalVotes <- totalNumberOfVotes;

/** Number of rounds of counting */
protected int countNumberValue;
//@ protected represents countNumber <- countNumberValue;

/** Number of candidates on the ballot paper */
protected int totalNumberOfCandidates;
//@ public represents totalCandidates <- totalNumberOfCandidates;

/** List of ballots being counted */
protected Ballot[] ballots;
//@ public represents ballotsToCount <- ballots;

/** List of candidates for election */
protected Candidate[] candidates;
//@ public represents candidateList <- candidates;

/** Lowest continuing vote */
protected int lowestVote;
//@ public represents lowestContinuingVote <- lowestVote;

/**
* Default Constructor
*/
/*@ also
@ public normal_behavior
@ ensures state == EMPTY;
@ ensures countNumber == 0;
@ ensures numberElected == 0;
@ ensures numberEliminated == 0;
@*/
public /*@ pure @*/ ElectionAlgorithm () {
//@ assert false;
}
```

```
/**
* Determine if the candidate has enough votes to be elected.
*
* @param candidate
* The candidate record
* @return True if the candidate has at least a quota of votes
* @see http://www.cev.ie/htm/tenders/pdf/1_1.pdf, page 79, paragraph 120 (2)
*/
/*@ also
@ protected normal_behavior
@*/
protected boolean hasQuota(ie.koa.Candidate candidate) {
//@ assert false;
return false;
}

/**
* Determine if the candidate was elected in any previous round
*
* @param candidate
* The candidate record
* @return True if the candidate has already been elected
*/
/*@ also
@ protected normal_behavior
@*/
protected boolean isElected(ie.koa.Candidate candidate) {
//@ assert false;
return false;
}

/**
* Determine how many surplus votes a candidate has.
*
* @design The surplus is the maximum number of votes available for transfer
* @param candidate
* The candidate record
* @return The undistributed surplus for that candidate, or zero if the
* candidate has less than a quota of votes
* @see http://www.cev.ie/htm/tenders/pdf/1_1.pdf, page 79, paragraph 120 (2)
*/
/*@ also
@ protected normal_behavior
@*/
protected int getSurplus(ie.koa.Candidate candidate) {
//@ assert false;
return 0;
}
/**
* @design The deposit saving threshold is one plus one quarter of the full
* quota.
* @design This needs to be checked just before the candidate is eliminated
* @see http://www.cev.ie/htm/tenders/pdf/1_2.pdf, section 3 page 13, section 4
* page 17 and section 14
* @param candidate
* @return true if candidate has enough votes to save deposit
*/
/*@ also
@ protected normal_behavior
@*/
protected boolean isDepositSaved(ie.koa.Candidate candidate) {
//@ assert false;
return false;
}

/**
* Distribution of highest surplus.
*
* @param candidateWithSurplus
* @design At most one surplus may be distributed in each round
* @see "http://www.cev.ie/htm/tenders/pdf/1_2.pdf, section 12, page 47"
*/
/*@ also
@ protected normal_behavior
@*/
protected void distributeSurplus(ie.koa.Candidate candidateWithSurplus)
{
//@ assert false;
}

/**
* Elimination of a candidate and transfer of votes.
*
* @param candidatesToEliminate One or more candidates to be excluded from the
* election in this count
* @param numberToEliminate Number of candidates to eliminate in this count
*/
/*@ also
@ protected normal_behavior
@*/
protected void eliminateCandidates(
ie.koa.Candidate[] candidatesToEliminate,
long numberToEliminate) {
//@ assert false;
}

/**
* Declare results.
*
* @return This election result
*/
/*@ also
@ public normal_behavior
@*/
public ElectionResults report() {
//@ assert false;
return null;
}
```

```
/**
* Start the counting.
*/
/*@ also
@ public normal_behavior
@*/
public void count() {
//@ assert false;
}

/**
* Load candidate details and number of seats.
*
* @param electionDetails The candidate details and number of seats
*/
/*@ also
@ public normal_behavior
@*/
public void setup (ElectionDetails electionDetails) {
//@ assert false;
}

/**
* Load all valid vote details.
* @param ballotBox The complete set of valid votes
*/
/*@ also
@ public normal_behavior
@*/
public void load(BallotBox ballotBox) {
//@ assert false;
}

/**
* Gets the current number of votes for this candidate ID.
*
* @design This method an also be used to check the number of nontransferable
* votes.
*
* @param candidateID Candidate ID for which to check the votes.
* @return Number of votes currently assigned to this candidate
*/
/*@ also
@ public normal_behavior
@*/
protected /*@ pure @*/ long getNumberOfVotes (long candidateID) {
//@ assert false;
return 0;
}

/**
* Gets the potential number of transfers from one candidate to another.
*
* @design This method is needed to get the proportions to use when transferring
* surplus votes.
*
* @param fromCandidate Candidate from which to check the transfers
* @param toCandidateID Candidate ID to check for receipt of transfered
votes
* @return Number of votes potentially transferable from this candidate to that
* candidate
*/
/*@ also
@ protected normal_behavior
@*/
protected /*@ pure @*/ int getPotentialTransfers (Candidate
fromCandidate,
long toCandidateID) {
//@ assert false;
return 0;
}

/**
* Gets the status of the algorithm in progress.
* @return The state variable value {@link #EMPTY}, {@link #SETUP},
* {@link #PREOAD}, {@link #LOADING}, {@link #PRECOUNT},
* {@link #COUNTING}, {@link #FINISHED} or {@link #REPORT}.
*/
/*@ also
@ public normal_behavior
@*/
public /*@ pure @*/ byte getStatus() {
//@ assert false;
return 0;
}
/**
* Gets the next preference continuing candidate.
*
* @param ballot Ballot paper from which to get the next preference
*
* @return Candidate if next continuing candidate or NONTRANSFERABLE
*/
/*@ also
@ public normal_behavior
@*/
protected long getNextContinuingPreference(Ballot ballot) {
//@ assert false;
return 0;
}
/**
* Determine if a candidate ID belongs to a continuing candidate.
*
* @param candidateID
* The ID of candidate for which to check the status
*
* @return <code>true</code> if this candidate ID matches that of a
* continuing candidate
*/
```

```
/*@ also
@ public normal_behavior
@*/
protected /*@ pure @*/ boolean isContinuingCandidateID (long
candidateID) {
//@ assert false;
return false;
}

/**
* Determine actual number of votes to transfer to this candidate
*
* @design The number of votes in a surplus transfered is in proportion to
* the number of transfers available throughout the candidates ballot stack
*
* @param fromCandidate Candidate from which to count the transfers
* @param toCandidate Continuing candidate eligible to receive votes
* @return Number of votes available for transfer
*/
/*@ also
@ public normal_behavior
@*/
protected /*@ pure @*/ int getActualTransfers (
/*@ non_null @*/ Candidate fromCandidate,
/*@ non_null @*/ Candidate toCandidate) {
//@ assert false;
return 0;
}

/**
* Determine the rounded value of a fractional transfer.
*
* @design This depends on the shortfall and the relative size of the other
* fractional transfers.
*
* @param fromCandidate
* Elected candidate from which to distribute surplus
*
* @param toCandidate
* Continuing candidate potentially eligible to receive
transfers
*
* @return <code>1</code> if the fractional vote is to be rounded up
* <code>0</code> if the fractional vote is to be rounded down
*/
/*@ also
@ public normal_behavior
@*/
protected /*@ pure @*/ long getRoundedFractionalVote (
/*@ non_null @*/ Candidate fromCandidate,
/*@ non_null @*/ Candidate toCandidate) {
//@ assert false;
return 0;
}
```

```
/**
* Determine shortfall between sum of transfers rounded down and the size of
* surplus.
*
* @param fromCandidate
* Elected candidate from which to distribute surplus
*
* @return The shortfall between the sum of the transfers and the size of
surplus
*/
/*@ also
@ public normal_behavior
@*/
protected /*@ pure @*/ long getTransferShortfall (
/*@ non_null @*/ Candidate fromCandidate) {
//@ assert false;
return 0;
}

/**
* Draw lots to choose between two continuing candidates.
*
* @design This needs to be random but consistent, so that same
* result is always given for the same pair of candidates.
*
* @param firstCandidate
* The first of the two candidates to be selected from
*
* @param secondCandidate
* The second of the two candidates to be selected from
*
* @return The candidate ID of the chosen candidate
*/
/*@ also
@ public normal_behavior
@*/
protected long randomSelection (
/*@ non_null @*/ Candidate firstCandidate,
/*@ non_null @*/ Candidate secondCandidate) {
//@ assert false;
return 0;
}

/**
* List each ballot ID in order by random number used to show how the votes have
* been mixed and numbered.
*
* @param ballot Ballot for which to get order of
* @return Order of this ballot in numbered list of ballots
*/
/*@ also
@ public normal_behavior
@*/
protected /*@ pure @*/ long getOrder(Ballot ballot) {
```

```
//@ assert false;
return 0;
}

/**
* List each candidate ID in order by random number to show how lots would have
* been chosen.
*
* @param candidate Candidate for which to get the order of
* @return Order of this candidate for use when lots are chosen
*/
/*@ also
@ public normal_behavior
@*/
protected /*@ pure @*/ long getOrder(Candidate candidate) {
//@ assert false;
return 0;
}

/**
* Determine the indivisible remainder after integer division by the transfer
* factor for surpluses.
*
* @design This can all be done with integer arithmetic; no need to use
* floating point numbers, which could introduce rounding errors.
*
* @param fromCandidate Elected candidate from which to count the transfers
* @param toCandidate Continuing candidate eligible to receive votes
*
* @return The size of the quotient remainder
*/
/*@ also
@ public normal_behavior
@*/
protected /*@ pure @*/ long getTransferRemainder (
/*@ non_null @*/ Candidate fromCandidate,
/*@ non_null @*/ Candidate toCandidate) {
//@ assert false;
return 0;
}

/**
* Determine if one continuing candidate is higher than another, for the purpose
* of resolving remainders of transfer quotients.
*
* @design This is determined by finding the earliest round of counting in which
* these candidates had unequal votes. If both candidates are equal at
* all counts then random numbers are used to draw lots.
*
* @see <a href="http://www.cev.ie/htm/tenders/pdf/1_2.pdf">Department of
* Environment and Local Government, Count Requirements and Commentary on Count
* Rules, section 7, page 25</a>
*
* @param firstCandidate
* The first of the two candidates to be compared
*
* @param secondCandidate
* The second of the two candidates to be compared
*
* @return <code>true</code> if first candidate is deemed to have received more
* votes than the second.
*/
/*@ also
@ public normal_behavior
@*/
protected /*@ pure @*/ boolean isHigherThan (
/*@ non_null @*/ Candidate firstCandidate,
/*@ non_null @*/ Candidate secondCandidate) {
//@ assert false;
return false;
}
/**
* Determine the number of continuing candidates with a higher remainder in
* their transfer quotient, or deemed to have a higher remainder.
*
* @design There must be a strict ordering of candidates for the purpose of
* allocating the transfer shortfall. If two or more candidates have equal
* remainders then use the number of transfers they are about to receive and if
* the number of transfers are equal then look at the number of votes already
* received.
*
* @param fromCandidate
* Elected candidate from which to distribute surplus
*
* @param toCandidate
* Continuing candidate potentially eligible to receive transfers
*
* @return The number of continuing candidates with a higher quotient remainder
* than this candidate
*/
/*@ also
@ public normal_behavior
@*/
protected /*@ pure @*/ long getCandidateOrderByHighestRemainder (
/*@ non_null @*/ Candidate fromCandidate,
/*@ non_null @*/ Candidate toCandidate) {
//@ assert false;
return 0;
}

/**
* Get the maximum number of votes transferable to continuing candidates.
* @param fromCandidate Candidate ID from which to check the transfers
* @return Number of votes potentially transferable from this candidate
*/
/*@ also
@ public normal_behavior
@*/
```

```
protected /*@ pure @*/ long getTotalTransferableVotes (
/*@ non_null @*/ Candidate fromCandidate) {
//@ assert false;
return 0;
}
/**
* Transfer votes from one candidate to another.
* @param fromCandidate Elected or excluded candidate
* @param toCandidate Continuing candidate
* @param numberOfVotes Number of votes to be transfered
*/
/*@ also
@ public normal_behavior
@*/
protected /*@ pure @*/ void transferVotes(
/*@ non_null @*/ Candidate fromCandidate,
/*@ non_null @*/ Candidate toCandidate,
long numberOfVotes) {
//@ assert false;
}

/**
* Complete one round of counting.
*
*/
/*@ also
@ public normal_behavior
@*/
protected void completeRound() {
  //@ assert false;
  }

/**
* Update list of decision events.
*/
/*@ also
@ public normal_behavior
@*/
protected void updateDecisions() {
  //@ assert false;
  }
}
```

B2. Ballot.java

```
package ie.koa;
/**
* The Ballot class represents a ballot paper in an Irish election,
* which uses the Proportional Representation Single Transferable Vote
* (PRSTV) system.
*
* @author Dermot Cochran
* @copyright Systems Research Group and University College Dublin,
Ireland.
* @reviewer Joe Kiniry
*
* @see <a href="http://www.cev.ie/htm/tenders/pdf/1_2.pdf">Department of
* Environment and Local Government, Count Requirements and Commentary on
Count
* Rules, sections 3-14</a>
*/
public class Ballot {
/**
* Candidate ID value to use for nontransferable ballot papers.
*
* @design A special candidate ID value is used to indicate
* non-transferable votes i.e., when the list of preferences has
* been exhausted and none of the continuing candidates are in the
* preference list, then the ballot is deemed to be nontransferable.
*
* @see <a href="http://www.cev.ie/htm/tenders/pdf/1_2.pdf">Department
of
* Environment and Local Government, Count Requirements and Commentary
on Count
* Rules, section 7, pages 23-27</a>
*/
public static final long NONTRANSFERABLE = 0;
/** @review kiniry 8 Feb 2006 - Each of these fields needs at least
* one invariant, typically. E.g., can ballotID be negative? If
* ballotID is unique per Ballot object, then this needs to be
* expressed in an invariant. These examples are below. */
/** Ballot ID number */
//@ public invariant 0 < ballotID;
/*@ public instance invariant (\forall Ballot a,b;
@ a != null && b != null;
@ a.ballotID == b.ballotID <==> a == b);
@*/
protected /*@ spec_public @*/ long ballotID;
/** Candidate ID to which this ballot is assigned */
//@ public invariant (0 < candidateID) || (candidateID ==
NONTRANSFERABLE);
/*@ public invariant (0 <= positionInList &&
@ positionInList < numberOfPreferences) ==>
@ candidateID == preferenceList[positionInList];
@ public invariant (positionInList == numberOfPreferences) ==>
@ candidateID == NONTRANSFERABLE;
@*/
protected /*@ spec_public @*/ long candidateID;
/** Preference list of candidate IDs */
/*@ public invariant (\forall int i;
@ 0 <= i && i < numberOfPreferences;
@ preferenceList[i] > 0 &&
@ preferenceList[i] != NONTRANSFERABLE);
@ public invariant (\forall int i, j;
@ 0 < i && i < numberOfPreferences && 0 <= j && j <
i;
@ preferenceList[i] != preferenceList[j]);
```

```
@*/
protected /*@ spec_public non_null @*/ long[] preferenceList;
/** Total number of valid preferences on the ballot paper */
//@ public invariant 0 < numberOfPreferences;
protected /*@ spec_public @*/ long numberOfPreferences;
/** Position within preference list */
//@ public initially positionInList == 0;
//@ public invariant 0 <= positionInList;
//@ public invariant positionInList <= numberOfPreferences;
//@ public constraint \old(positionInList) <= positionInList;
protected /*@ spec_public @*/ int positionInList;
/** @review kiniry 8 Feb 2006 - I presume that these arrays should
* be non_null. Note that in the latest JML tools release, and in
* the upcoming ESC/Java2 release, non_null is the default semantics
* of references. */
/** Candidate ID to which the vote is assigned at the end of each count
*/
protected /*@ spec_public non_null @*/ long[] candidateIDAtCount;
/** Last count number in which this ballot was transfered */
//@ public invariant 0 <= countNumberAtLastTransfer;
//@ public initially countNumberAtLastTransfer == 0;
protected /*@ spec_public @*/ long countNumberAtLastTransfer;
/** Random number used for proportional distribution of surplus votes */
/*@ public invariant (\forall Ballot a, b; a != null && b != null;
@ (a.randomNumber == b.randomNumber) <==> (a == b));
@ public constraint randomNumber == \old (randomNumber);
@*/
protected /*@ spec_public @*/ long randomNumber;
/**
* Default constructor
*/
/*@ public normal_behavior
@ ensures numberOfPreferences == 0;
@ ensures countNumberAtLastTransfer == 0;
@ ensures positionInList == 0;
@ ensures candidateID == NONTRANSFERABLE;
@ ensures ballotID > 0;
@*/
public /*@ pure @*/ Ballot() {
//@ assert false;
}
/**
* Copy constructor
*
* @return An exact copy of this ballot paper
*/
/*@ public normal_behavior
@ ensures this == \result;
@*/
public /*@ pure @*/ Ballot copy() {
//@ assert false;
return null;
}
/**
```

```
* Get the location of the ballot at each count
*
* @param countNumber
* @return The candidate ID or the NONTRANSFERABLE constant
* @review kiniry 8 Feb 2006 - Note the rewrite of precondition to
* make range more obvious. I have rewritten several other
* specifications in a similar fashion, but only in this class.
*/
/*@ public normal_behavior
@ requires 0 <= countNumber;
@ requires countNumber <= countNumberAtLastTransfer;
@ ensures \result == candidateIDAtCount[countNumber];
@*/
public /*@ pure @*/ long getPreferenceAtCount(int countNumber) {
//@ assert false;
return 0;
}
/**
* Get the count number for the last transfer of this ballot
* @return The last count at which this ballot was transfered
*/
/*@ public normal_behavior
@ ensures \result == countNumberAtLastTransfer;
@*/
public /*@ pure @*/ long getCountNumberAtLastTransfer() {
//@ assert false;
return 0;
}
/**
* Get first preference vote from this ballot
*
* @design There must always be a first preference vote in each
* ballot, otherwise the vote is not included and need not be
* loaded. The quota is calculated from the number of first
* preference votes, so that empty ballots are not included.
*
* @reference http://www.cev.ie/htm/tenders/pdf/1_2.pdf, section 3, page
12
* @return The candidate ID of the first preference for this ballot
*/
/*@ public normal_behavior
@ requires 0 < numberOfPreferences;
@ ensures \result != NONTRANSFERABLE;
@ ensures \result == preferenceList[0];
@*/
public /*@ pure @*/ long getFirstPreference() {
//@ assert false;
return 0;
}
/**
* Load the ballot details
*
* @param candidateIDList List of candidate IDs in order from first
* preference
```

```
*
* @param listSize Number of candidate IDs in the list
*
* @param uniqueID The serial number of the ballot or equivalent
*
* @design There should be at least one preference in the list.
* Empty or spoilt votes should neither be loaded nor counted.
* There should be no duplicate preferences in the list and none
* of the candidate ID values should match the special value for
* non transferable votes.
* <p> There should be no duplicates in the preference list; but
* there is no need to make this a precondition because duplicates
* will be ignored and skipped over.
*/
/*@ public normal_behavior
@ requires 0 <= listSize;
@ requires (\forall int i; 0 <= i && i < listSize;
@ (candidateIDList[i]) != NONTRANSFERABLE);
@ ensures numberOfPreferences == listSize;
@ ensures ballotID == uniqueID;
@ ensures (\forall int i; 0 <= i && i < listSize;
@ (preferenceList[i] == candidateIDList[i]));
@*/
public void load(long[] candidateIDList, long listSize, long uniqueID)
{
//@ assert false;
}
/**
* Get candidate ID to which this ballot is assigned
*
* @return The candidate ID to which this ballot is assigned
*/
/*@ public normal_behavior
@ requires 0 <= positionInList;
@ requires positionInList <= numberOfPreferences;
@ requires preferenceList != null;
@ ensures (positionInList == numberOfPreferences) ==>
@ (\result == NONTRANSFERABLE);
@ ensures (positionInList < numberOfPreferences) ==>
@ (\result == preferenceList[positionInList]);
@*/
public /*@ pure @*/ long getCandidateID() {
//@ assert false;
return 0;
}
/**
* Get next preference candidate ID
*
* @param offset The number of preferences to look ahead
*
* @return The next preference candidate ID
*/
/*@ public normal_behavior
@ requires 0 <= positionInList;
@ requires 1 <= offset;
@ requires positionInList <= numberOfPreferences;
@ requires preferenceList != null;
@ ensures (positionInList + offset >= numberOfPreferences) ==>
@ (\result == NONTRANSFERABLE);
@ ensures (positionInList + offset < numberOfPreferences) ==>
@ (\result == preferenceList[positionInList + offset]);
@*/
public /*@ pure @*/ long getNextPreference(int offset) {
//@ assert false;
return 0;
}
/**
* Transfer this ballot to the next preference candidate
*
* @design This method may be called multiple times during the same
* count until the ballot is nontransferable or a continuing
* candidate ID is found in the remainder of the preference list.
*
* @param countNumber The count number at which the ballot was
* transfered
*/
/*@ public normal_behavior
@ requires 0 <= positionInList;
@ requires positionInList <= numberOfPreferences;
@ requires countNumberAtLastTransfer <= countNumber;
@ assignable countNumberAtLastTransfer, positionInList;
@ ensures countNumberAtLastTransfer == countNumber;
@ ensures \old(positionInList) <= positionInList;
@ ensures (positionInList == \old(positionInList) + 1) ||
@ (positionInList == numberOfPreferences);
@*/
public void transfer(long countNumber) {
//@ assert false;
}
/**
* Get ballot ID number.
*
* @return ID number for this ballot
*/
/*@ public normal_behavior
@ ensures \result == ballotID;
@*/
public /*@ pure @*/ long getBallotID() {
//@ assert false;
return 0;
}
/**
* This method checks if this ballot paper is assigned to this
candidate.
*
* @design It is valid to use <code>NONTRANSFERABLE</code> as the ID
value to
* be checked. This ballot paper can only be assigned to one candidate
```

```
at a time;
* there is no concept of fractional transfers of votes in the Irish
electoral
* system.
*
* @param candidateIDToCheck The unique identifier for this candidate.
*
* @return <code>true</code> if this ballot paper is assigned to this
candidate ID
*/
/*@ public normal_behavior
@ requires (0 < candidateIDToCheck) ||
@ (candidateIDToCheck == NONTRANSFERABLE);
@ ensures (\result == true) <==> (candidateID ==
candidateIDToCheck);
@*/
public /*@ pure @*/ boolean isAssignedTo (long candidateIDToCheck) {
//@ assert false;
return false;
}
/**
* Gets remaining number of preferences.
*
* @return The number of preferences remaining
*/
/*@
@ public normal_behavior
@ requires positionInList <= numberOfPreferences;
@ ensures \result == numberOfPreferences - positionInList;
@*/
public /*@ pure @*/ long remainingPreferences() {
//@ assert false;
return 0;
}
/**
* Compares with another ballot paper's secret random number.
*
* @design It is intended to be able to compare random numbers without
* revealing the exact value of the random number, so that the random
* number cannot be manipulated in any way.
*
* @param other
* Other ballot to compare with this ballot
*
* @return <code>true</code> if other ballot has lower random number
*/
/*@ public normal_behavior
@ requires this.randomNumber != other.randomNumber;
@ ensures (\result == true) <==> (this.randomNumber >
other.randomNumber);
@*/
public /*@ pure @*/ boolean isAfter (Ballot other) {
//@ assert false;
return false;
}
}
```

B3. BallotBox.java

```
package ie.koa;
/**
* @author Dermot Cochran
* @copyright Systems Research Group and University College Dublin,
Ireland.
* @reviewer Joe Kiniry
*/
/** Data transfer structure for set of all valid votes */
public class BallotBox {
/** List of valid ballot papers */
// @review kiniry 1 Mar 2006 - This first invariant can be captured
// using the JML keyword \nonnullelements. I have changed the type
// annotation below on the ballots field to reflect such.
//@ public invariant \nonnullelements (ballots);
// @review kiniry 1 Mar 2006 - I would express each invariant that
// is approximately this complex in English as well as in JML.
// E.g., it is not clear to me that this invariant is capturing what
// you really meant to say because I do not know what high-level
// constraints are being expressed. For this particular invariant,
// is the constraint no two ballot IDs in a ballot box are the same?
// If so, I would express this invariant differently in a manner
// that is more mathematically clean but perhaps less "efficient"
// when viewing the expression from a computational viewpoint.
/**
* @constraint No two ballot IDs in a ballot box are the same.
*/
/*@ public invariant (\forall int i, j;
@ 0 <= i & i < numberOfBallots &
@ 0 <= j & j < numberOfBallots &
@ i != j;
@ ballots[i].ballotID != ballots[j].ballotID);
@*/
public /*@ non_null @*/ Ballot[] ballots;
// @review kiniry 1 Mar 2006 - There were missing '@' signs here, so
// these specs were not being interpreted.
// @review kiniry 1 Mar 2006 - You made a design decision in
// choosing an long for the numberOfBallots property of a ballot box.
// How do you justify that decision? Is it based upon law, the
// current population of Ireland, etc.? If there is no concrete
// restriction or assumption, is it not safer to make this
// representation a long at no appreciable loss of complexity?
// Obviously the type signature of the aforementioned size() method
// would change as well.
// @review kiniry 1 Mar 2006 - Abstraction is lacking for this type
// interface. In particular, I would introduce a size method:
/**
* @return the number of ballots in this ballot box.
*/
//@ ensures 0 <= \result;
public /*@ pure @*/ long size() {
//@ assert false;
return numberOfBallots;
}
// ...and therefore change the visibility of numberOfBallots to private
// and abstract these invariants to use the pure method.
//@ public invariant 0 <= numberOfBallots;
//@ public initially numberOfBallots == 0;
//@ public constraint numberOfBallots >= \old (numberOfBallots);
public int numberOfBallots;
// @review kiniry 1 Mar 2006 - This postcondition is redundant given
// your initially assertion above.
/*@ public normal_behavior
@ ensures numberOfBallots == 0;
@*/
public /*@ pure @*/ BallotBox() {
//@ assert false;
}
}
```

B4. Candidate.java

```
package ie.koa;
/**
* The Candidate object records the number of votes received during
* each round of counting. Votes can only be added to the candidate's
* stack while the candidate has a status of <code>CONTINUING</code>.
*
* @author Dermot Cochran
* @copyright Systems Research Group and University College Dublin, Ireland.
* @reviewer Joe Kiniry
*
* @see <a href="http://www.cev.ie/htm/tenders/pdf/1_2.pdf">Department of
* Environment and Local Government, Count Requirements and Commentary on Count
* Rules, sections 3-14</a>
*
* @review kiniry 8 Feb 2006 - Your heavyweight (behavioral) specs
* should have a visibility that matches that of the method
* (typically). Thus, your public methods below should have public
* behavioral specs. This is why your protected fields must be made
* spec_public---exactly so that they can be mentioned in public
* specs.
*/
public class Candidate {
  /** Identifier for the candidate */
  //@ public invariant 0 <= candidateID;
  //@ public invariant (state != UNASSIGNED) ==> 0 < candidateID;
  /*@ public invariant (\forall Candidate a, b;
    @   a != null && b != null &&
    @   a.state != UNASSIGNED && b.state != UNASSIGNED;
    @   (a.candidateID == b.candidateID) <==> (a == b));
    @ public constraint (state != UNASSIGNED) ==>
    @   candidateID == \old (candidateID);
    @*/
  protected /*@ spec_public @*/ long candidateID;

  /** Number of votes added at each count */
  /*@ public invariant (\forall int i; 0 <= i && i < MAXCOUNT;
    @ 0 <= votesAdded[i]);
    @*/
  /*@ public initially (\forall int i; 0 <= i && i < MAXCOUNT;
    @ votesAdded[i] == 0);
    @*/
  protected /*@ spec_public non_null @*/ int[] votesAdded;

  /** Number of votes removed at each count */
  /*@ public invariant (\forall int i; 0 <= i && i < MAXCOUNT;
    @   0 <= votesRemoved[i]);
    @*/
  /*@ public initially (\forall int i; 0 <= i && i < MAXCOUNT;
    @   votesRemoved[i] == 0);
    @*/
  protected /*@ spec_public non_null @*/ int[] votesRemoved;

  /** The status of the candidate at the latest count */
  /*@ public invariant state == ELECTED || state == ELIMINATED ||
    @ state == CONTINUING || state == UNASSIGNED;
    @ public initially state == UNASSIGNED;
    @*/
  protected /*@ spec_public @*/ byte state;

  /** The number of rounds of counting so far */
  //@ public invariant 0 <= lastCountNumber;
  //@ public initially lastCountNumber == 0;
  //@ public constraint \old(lastCountNumber) <= lastCountNumber;
  //@ public invariant lastCountNumber < MAXCOUNT;
  protected /*@ spec_public @*/ int lastCountNumber;

  /** The count number at which the last set of votes were added */
  //@ public invariant 0 <= lastSetAddedCountNumber;
  //@ public initially lastSetAddedCountNumber == 0;
  /*@ public constraint
    @ \old(lastSetAddedCountNumber) < lastSetAddedCountNumber;
    @*/
  //@ public invariant lastSetAddedCountNumber <= lastCountNumber;
  protected /*@ spec_public @*/ int lastSetAddedCountNumber;

  /** Unique random number used to simulate drawing of lots between
   * candidates */
  /*@ public invariant (\forall Candidate a, b;
    @   a != null && b != null &&
    @   a.state != UNASSIGNED && b.state != UNASSIGNED;
    @   (a.randomNumber == b.randomNumber) <==> (a == b));
   @ public constraint (state != UNASSIGNED) ==>
   @   randomNumber == \old (randomNumber);
   @*/
  protected /*@ spec_public @*/ long randomNumber;

  /** State value for a candidate neither elected nor eliminated yet. */
  public static final byte CONTINUING = 0;

  /**
   * State value for a candidate deemed to have been elected either by
   * having a quota or being the highest continuing candidate for the
   * last remaining seat.
  */
  public static final byte ELECTED = 1;

  /**
   * State value for a candidate excluded from election as being one
   * of the lowest continuing candidates at the end of a round of counting.
   */
  public static final byte ELIMINATED = 2;

  /** State value for a candidate object without a valid ID */
  public static final byte UNASSIGNED = 3;

  /** State value for a candidate defeated at the last round of election
   * e.g. the second highest remaining candidate when the last seat is being
   * filled
   */
  public static final byte DEFEATED = 4;

  /** Maximum possible number of counts
   *  @design This value is not set by the legislation; it is chosen so that
   *  fixed length arrays can be used in the specification.
   */
  public static final int MAXCOUNT = 100;

  /**
   * Gets number of votes added or removed in this round count.
   *
   * @param count This count number
   * @return A positive number if the candidate received transfers or
   * a negative number if the candidate's surplus was distributed or
   * the candidate was eliminated and votes transfered to another
   */
  /*@ public normal_behavior
    @ requires state != UNASSIGNED;
    @ requires 0 <= count;
    @ requires count <= lastCountNumber;
    @ requires count < MAXCOUNT;
    @ ensures \result == votesAdded[count] - votesRemoved[count];
    @*/
  public /*@ pure @*/ int getVoteAtCount (int count)
  {
    //@ assert false;
    return 0;
  }

  /**
   * Total vote received by this candidate less transfers to other
   * candidates.
   *
   * @return Net total of votes received
  */
  /*@ public normal_behavior
    @   requires state != UNASSIGNED;
    @   ensures \result == (\sum int i; 0 <= i && i <= lastCountNumber;
    @     ((votesAdded[i]) - (votesRemoved[i])));
    @*/
  public /*@ pure @*/ int getTotalVote () {
    //@ assert false;
    return 0;
  }

/**
 * Original number of votes recieved by this candidate before
 * transfers due to elimination or distribution of surplus votes.
 *
 * @return Gross total of votes received
 */
 /*@ public normal_behavior
   @   requires state != UNASSIGNED;
   @   ensures \result == (\sum int i; 0 <= i && i <= lastCountNumber;
   @     votesAdded[i]);
   @*/
  public /*@ pure @*/ int getOriginalVote() {
    //@ assert false;
    return 0;
  }

  /**
   * Get status at the current round of counting; {@link #ELECTED},
   * {@link #ELIMINATED} or {@link #CONTINUING} or {@link #UNASSIGNED}
   *
   * @return State value for this candidate
   */
  /*@ public normal_behavior
    @ ensures \result == state;
    @*/
  public /*@ pure @*/ byte getStatus() {
    //@ assert false;
    return state;
  }

  /**
   * Get the unique ID of this candidate.
   *
   * @return The candidate ID number
   */
  /*@ public normal_behavior
    @ requires state != UNASSIGNED;
    @ ensures \result == candidateID;
    @*/
  public /*@ pure @*/ long getCandidateID() {
    //@ assert false;
    return 0;
  }
  /**
   * This is the default constructor method for a <code>Candidate</code>.
   */
  /*@ public normal_behavior
    @   ensures state == UNASSIGNED;
    @*/
  public Candidate() {
    //@ assert false;
  }

  /**
   * Add a number of votes to the candidate's ballot stack.
   *
   * @design This method cannot be called twice for the same candidate
   *   in the same round of counting.
   *
   * @param numberOfVotes Number of votes to add
   * @param count The round of counting at which the votes were added
   */
  /*@ public normal_behavior
    @   requires state == CONTINUING;
    @   requires lastCountNumber < count;
    @   requires votesAdded[count] == 0;
    @   assignable lastCountNumber, votesAdded[count], lastSetAddedCountNumber;
    @   ensures votesAdded[count] == numberOfVotes;
    @   ensures lastCountNumber == count;
    @   ensures lastSetAddedCountNumber == count;
    @*/
  public void addVote (int numberOfVotes, int count) {
    //@ assert false;
  }

  /**
   * Removes a number of votes from a candidate's ballot stack.
   *
   * @design This method cannot be called twice for the same candidate
   * in the same round of counting.
   *
   * @param numberOfVotes Number of votes to remove from this candidate
   * @param count The round of counting at which the votes were removed
   */
  /*@ public normal_behavior
    @ requires state == ELIMINATED || state == ELECTED;
    @ requires lastCountNumber < count;
    @ requires votesRemoved[count] == 0;
    @ assignable lastCountNumber, votesRemoved[count];
    @ ensures votesRemoved[count] == numberOfVotes;
    @ ensures lastCountNumber == count;
    @*/
  public void removeVote (int numberOfVotes, int count) {
    //@ assert false;
  }

  /** Sets the candidate ID
    * @param candidateIDToAssign Identification number for this candidate
    * @design The candidate ID must be assigned by the client object because
    * the client object must know who the candidate is */
  /*@ public normal_behavior
    @ requires state == UNASSIGNED;
    @ requires 0 < candidateIDToAssign;
    @ requires (\forall Candidate other; other != null;
    @   other.candidateID != candidateIDToAssign);
    @ assignable state, candidateID;
    @ ensures candidateID == candidateIDToAssign;
    @ ensures state == CONTINUING;
    @*/
  public void setCandidateID (long candidateIDToAssign) {
    //@ assert false;
    }

  /** Declares the candidate to be elected */
  /*@ public normal_behavior
    @ requires state == CONTINUING;
    @ assignable state;
    @ ensures state == ELECTED;
    @*/
  public void declareElected() {
    //@ assert false;
  }

  /** Declares the candidate to be eliminated */
  /*@ public normal_behavior
    @ requires state == CONTINUING;
    @ assignable state;
    @ ensures state == ELIMINATED;
    @*/
  public void declareEliminated() {
    //@ assert false;
  }

  /**
  * Gets number of votes in last set of votes added
  *
  * @design In the first round of counting this is the same as
  * the number of first preferences, otherwise it is the most
  * set of votes received. The last set of votes received
  * are the only votes considered when a surplus is being distributed.
  *
  * @return The number of votes in the last set added
  */
  /*@ public normal_behavior
    @ ensures \result == votesAdded[lastSetAddedCountNumber];
    @*/
  public /*@ pure @*/ long getNumberOfVotesInLastSet() {
    //@ assert false;
    return 0;
  }

  /**
   * Gets the count number at which the last set of votes was added.
   *
   * @design This is needed to check which ballots are in the last set
   * added
   *
   * @see requirement 19, section 7, item 2, page 19
   *
   * @return The last count number at which votes were added
   */
  /*@ public normal_behavior
    @ ensures \result == lastSetAddedCountNumber;
    @*/
  public /*@ pure @*/ int getLastSetAddedCountNumber() {
    //@ assert false;
    return 0;
  }

 /**
  * Compares with another candidate's secret random number.
  *
  * @design It is intended to be able to compare random numbers without
  * revealing the exact value of the random number, so that the random
  * number cannot be manipulated in any way.
  *
  * @param other Other candidate to compare with this candidate
  *
  * @return <code>true</code> if other candidate has lower random number
  */
 /*@ public normal_behavior
   @ requires state != UNASSIGNED;
   @ ensures (\result == true) <==>
   @   (this.randomNumber > other.randomNumber);
   @*/
  public /*@ pure @*/ boolean isAfter(Candidate other) {
    //@ assert false;
    return false;
  }
}
```

B5. Decision.java

```
package ie.koa;
/** Decisions taken during the counting of votes.
 *
 * @author Dermot Cochran
 * @copyright Systems Research Group and University College Dublin, Ireland.
 *
 * @design It is necessary to be able to record any decision which might
 * influence the order in which votes are counted of transfered.
 */
public class Decision {
  /**
   * State value for decision to declare this candidate elected by reaching
   * the quota
   */
  public final static byte ELECTBYQUOTA = 0;

  /**
   * State value for decision to eliminate this candidate as the lowest
   * continuing candidate
   */
  public final static byte EXCLUDE = 1;

  /**
   * Default state value to indicate that no decision has been taken
   */
  public final static byte NODECISION = 2;

  /**
   * State value for decision to round up a fractional transfer of votes to
   * this candidate
   */
  public final static byte ROUNDUPFRACTION = 4;

  /**
   * State value for decision to distribute the surplus of this candidate
   */
  public final static byte DISTRIBUTESURPLUS = 5;

  /**
   * State value for the decision to deem this candidate elected as one
   * of the highest continuing candidates for the last remaining seats
   */
  public final static byte DEEMELECTED = 6;

  /** Type of decision taken */
  //@ public initially decisionTaken == NODECISION;
  /*@ public invariant (decisionTaken == ELECTBYQUOTA) ||
    @   (decisionTaken == EXCLUDE) || (decisionTaken == NODECISION) ||
    @   (decisionTaken == ROUNDUPFRACTION) ||
    @   (decisionTaken == DISTRIBUTESURPLUS) || (decisionTaken == DEEMELECTED);
    @   public constraint (\old (decisionTaken) != NODECISION) ==>
    @   decisionTaken == \old (decisionTaken);
    @*/
  public byte decisionTaken;

  /** Candidate to which the decision applied */
  //@ public invariant (decisionTaken != NODECISION) ==> 0 < candidateID;
  //@ public invariant candidateID != Ballot.NONTRANSFERABLE;
  /*@ public constraint (decisionTaken != NODECISION) ==>
    @   candidateID == \old (candidateID);
    @*/
  //@ public initially candidateID == 0;
  public long candidateID;

  /** Round of counting at which decision was taken */
  //@ public invariant 0 <= atCountNumber;
  //@ public initially atCountNumber == 0;
  /*@ public constraint (decisionTaken != NODECISION) ==>
    @ atCountNumber == \old (atCountNumber);
    @*/
  public long atCountNumber;

  /** Indicates if lots were drawn to make this decision */
  /*@ public invariant (decisionTaken == ELECTBYQUOTA) ==>
    @ chosenByLot == false;
    @ public constraint (decisionTaken != NODECISION) ==>
    @ chosenByLot == \old (chosenByLot);
    @*/
  public boolean chosenByLot;

  /** Default constructor */
  /*@ public normal_behavior
    @ ensures decisionTaken == NODECISION;
    @ ensures atCountNumber == 0;
    @ ensures candidateID == 0;
    @ ensures chosenByLot == false;
    @*/
  public /*@ pure @*/ Decision () {
    //@ assert false;
  }
}
```

B6. ElectionDetails.java

```
package ie.koa;
/** Data transfer structure for candidate ID details and number of seats.
 */
public class ElectionDetails {
  /** Number of candidates for election in this constituency */
  //@ public invariant 0 < numberOfCandidates;
  public long numberOfCandidates;

  /** Number of seats to be filled in this election */
  //@ public invariant 0 < numberOfSeatsInThisElection;
  //@ public invariant numberOfSeatsInThisElection <= totalNumberOfSeats;
  public long numberOfSeatsInThisElection;

  /** Number of seats in this constituency */
  //@ public invariant 0 < totalNumberOfSeats;
  public long totalNumberOfSeats;
  /** List of ID values for all candidates in this election */
  /*@ public invariant (\forall int i;
    @   0 <= i && i < numberOfCandidates;
    @   0 < candidateIDs[i] &&
    @   candidateIDs[i] != Ballot.NONTRANSFERABLE);
    @*/
  // @constraint No duplicate candidate IDs are allowed.
  /*@ public invariant (\forall int i, j;
    @ 0 <= i && i < numberOfCandidates &&
    @ 0 <= j && j < numberOfCandidates &&
    @ i != j;
    @ candidateIDs[i] != candidateIDs[j]);
    @*/
  public /*@ non_null @*/ long[] candidateIDs;

  public /*@ pure @*/ ElectionDetails() {
    //@ assert false;
  }
}
```

B7. ElectionResults.java

```
package ie.koa;
/**
 * Election results
 */
public class ElectionResults {
  //@ public invariant 0 <= numberElected;
  public long numberElected;

  /*@ public invariant (\forall int i;
    @   0 <= i && i < numberElected;
    @   0 < electedCandidateIDs[i] &&
    @   electedCandidateIDs[i] != Ballot.NONTRANSFERABLE);
    @ public invariant (\forall int i, j;
    @   0 <= i && i < numberElected &&
    @   0 <= j && j < numberElected &&
    @   i != j;
    @   electedCandidateIDs[i] != electedCandidateIDs[j]);
    @*/
  public /*@ non_null @*/ long[] electedCandidateIDs;

  //@ public invariant 0 <= totalNumberOfCounts;
  public long totalNumberOfCounts;

  public /*@ pure @*/ ElectionResults() {
    //@ assert false;
  }
}
```