UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Fall 2007

Project #1: The CS61B Configurable Voting System

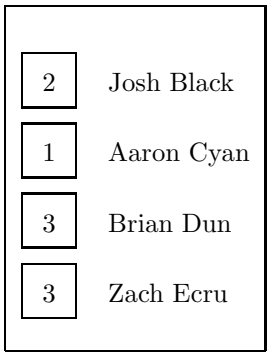
Due: 4 October 2007 at midnight (beginning of 5 October)

1 Background

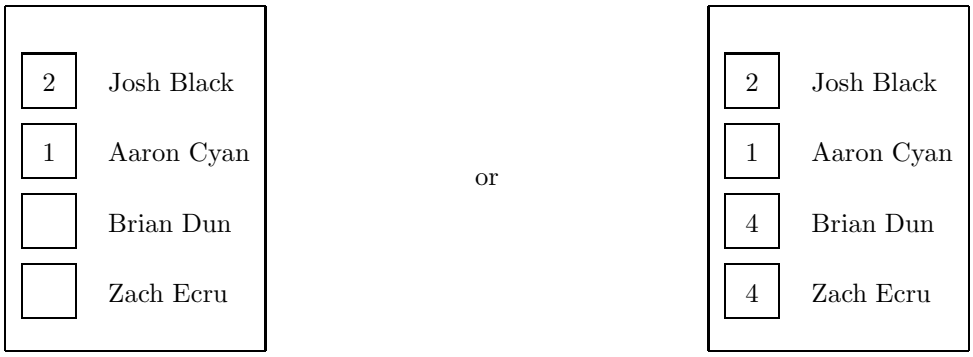
Voting is in the news these days. In fact, EECS's own Prof. David Wagner was a member of the "red team" that recently investigated the security of assorted electronic voting machines in California. In this assignment, however, we're going to address the much simpler task of counting votes in the presence of honest voters and election personnel.

Actually, counting votes is pretty easy if you're using a simple *first-past-the-post voting system,* as in the United States, where we elect a single person to each office and the winner is the one that garners a majority ($> 50\%$) of the votes cast by the members of the Electoral College. However, this system has many well-known weaknesses (especially in the U.S., where the Electors are not, in fact, the people as a whole). In fact, no voting system is perfect (see §8), and each is imperfect in its own way. Let's consider, therefore, a flexible system in which we can swap in different vote-counting methods, and, while we're at it, allow voters more expressiveness about their preferences and allow for elections where we select more than one candidate.

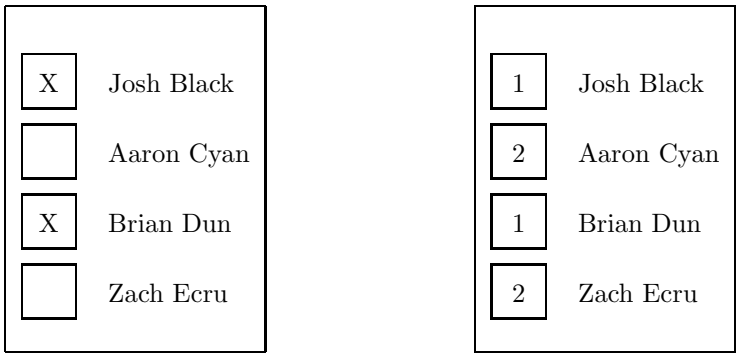
In general, voting takes as input the expressed preferences of each voter and combines them according to some rule into a collective preference among candidates. Thus, a voter might fill in a ballot like this:



to indicate Cyan as first choice, Black as second, with Dun and Ecru tied for third. Depending on the rules, we might allow the same ballot to be expressed



Some voting systems don't provide for such specific preferences. In *approval voting,* one selects all candidates one finds acceptable, as in the ballot on the left below. U.S. presidential elections are an example, with the provision that one may select at most one candidate. In any case, one could encode approval votes like these with ranked preferences, using the convention that the approved candidates tie for first, and the disapproved candidates tie for last, as shown in the ballot on the right.



Given a set of preferences, there are various ways of combining the votes—that is, deciding who wins. In our presidential system, where everyone gets to vote for at most one candidate, the candidate (if any) with a majority of votes wins. If voters may express finer preferences, things become more complicated. For example, in *instant-runoff elections,* a candidate who is ranked highest (nearest to first choice) by a majority of the ballots cast is elected, and if there is no such candidate, then the lowest-scoring candidate is eliminated from all ballots (in effect), and the process is repeated. The process fails only if at some point all candidates are tied. Under *Condorcet methods* of voting, candidates are compared pairwise. If one candidate if preferred by a majority of voters to every other candidate, then that candidate wins. If there is no such candidate, the method fails to find a winner.

To provide a flexible voting framework, then, you have to accommodate both a variety of input formats and conventions and a variety of decision procedures. In this assignment, we're going to create such a framework—one that is both *configurable* and *extensible* (i.e., one can add more input formats and voting rules without rewriting existing code).

2 Input

The input to your program will consist of a *configuration file,* which will specify the format and source of ballots, the rules for voting, the names of the candidates, and the number of candidates to be elected. Depending on the ballot source selected, there will be additional input containing the actual ballots submitted. You'll start your program by typing

`java Elect` *CONFIG SOURCE...*

where *CONFIG* is the name of a configuration file and the remaining arguments (*SOURCE...*) identify the source of ballots (file names, for example). The interpretation of the *SOURCE...* arguments will depend on the input method selected in the configuration file.

The configuration file is in the form of Lisp S-expressions. These in turn consist of a sequence of *tokens:* '`(`', '`)`', strings enclosed in double quotes ('`"`') and containing no quotation marks, and words (sequences of characters not including parentheses, double quotes, semicolons, or whitespace). Whitespace (spaces, tabs, and newlines) serves only to separate word tokens (outside of strings). Also outside of strings, everything between a semicolon ('`;`') and an end-of-line is a comment, which is treated as whitespace. All input is case-sensitive.

The following expressions are meaningful in the configuration file:

`(input-method` *symbol*`)`	The symbol (a word token) names a Java class the defines how ballot input is read.
`(ballot-format` *symbol*`)`	The symbol names a Java class that describes how to parse a ballot.
`(voting-system` *symbol*`)`	The symbol names a Java class that describes how to determine the winner(s) of an election
`(candidate "`*name*`")`	Adds a candidate with the given name. Leading and trailing blanks around a candidate's name are ignored. Each candidate's name (stripped of leading and trailing blanks) must be unique and non-empty. There can be more than one of these expressions, and their order is significant.
`(slots` N`)`	The number of candidates to be elected is N (a positive integer).
`(rankings` N`)`	The range of preference levels is 1 (highest) to N (lowest).

So for example, the very first ballot in this handout might have come from an election configured like this:

```
(slots 1)                  ; One winner to be selected
(rankings 4)               ; Candidates get integer rankings 1-4.

(candidate "Josh Black") (candidate "Aaron Cyan")
(candidate "Brian Dun") (candidate "Zach Ecru")

(voting-system SimpleMajority)
(input-method FileInput)   ; Input is from a file
(ballot-format OrderedRankFormat)
                           ; Each ballot contains 4 numbers in same
                           ; order as the candidates
```

Order, as you can see, is irrelevant, except for the order in which candidates are listed. Likewise, the input format is more liberal than I've shown. You could have written, e.g.,

```
(
slots
    1  )
```

We'll talk about how the classes named "`FileInput`" and so forth get used in a bit. For this configuration, the input method might expect that each ballot is one line in a file and with the expected input format, the first sample ballot from earlier might look like this:

```
2 1 3 3
```

3 Output

We'll keep the output short and sweet. Your program should print the names of the winning candidates (the number of candidates is given by the "slots" parameter in the configuration file) in the same order that they were listed in the configuration file, one per line. If the voting method fails to find a winner or the right number of winners, the output will consist of the single line "`Result undetermined.`" with no further output.

There are three general kinds of error your program can encounter: errors during initialization (before ballot-tallying starts), errors in input of ballots, and spoiled ballots (ballots that don't parse or that disobey the rules of a particular voting system). The first two kinds of error are fatal: on encountering one, your program should print one or more lines of error message on the "standard error" stream (which in Java corresponds to the stream `System.err`). The first line of the error message should start with the string "`ERROR:`". Our autograder will look only for that string, and will ignore anything after it in the standard error stream. The last kind of error should result in a warning message, also on the standard error stream. The warning message can be any one-line message that begins with the string "`WARNING:`". The autograder will check only that there are the right number of such warnings (ignoring the content of the message).

4 Extensibility: Implementing Plugins

[We're throwing you a little advanced material here just to give you some idea to what you can do in Java and how hard (compared to Scheme) you have to work to do it. Don't worry if it is not immediately clear. Over the course of the semester, the mysterious parts will get filled in.]

The names given in the `input-method`, `ballot-format`, and `voting-system` parameters are to be converted by your program into the names of Java classes that supply the implied operations of reading input, interpreting the input, checking the validity of the input, and determining winners. However, you are to write your program without knowing what these classes might be (except for a few samples that we ask you to provide). Only when you *execute* the program does it actually find out the identities of the classes that contain parts of its implementation. In contrast, the usual case with standard procedural languages such as C++ is for all code to get written and linked together *before* execution. Your system will allow an enterprising municipality to write a new voting system and plug it in without having to alter your program.

In Scheme, this sort of thing is easy to arrange, using the functions for reading files and the `eval` function. For typical implementations of C and C++, it is considerably harder. Java is somewhere in between. For now, I'm afraid that a lot of the code below will appear to be magical incantation, which you probably won't be able to appreciate fully for awhile yet. It shouldn't really matter.

The basic idea is that Java provides mechanisms for *reflection:* for making programs able to manipulate at execution time entities that correspond to pieces of programs (in particular, pieces of themselves, hence the term). Here, for example, is a Java program that, when started with

```
java Execute MyClass
```

will look for a compiled Java class named `MyClass`, and print whatever its `toString` method returns (`toString` is defined for all Java objects):

```
public class Execute {
   public static void main (String[] args) {
      if (args.length < 1)
          return;
      try {
          Class<?> clazz = Class.forName (args[0]);
          Object obj = clazz.newInstance ();
          System.out.println (obj.toString ());
      } catch (Exception e) {
          System.err.printf ("Something went wrong with %s.%n", args[0]);
      }
   }
}
```

So if you run Execute in a directory where you also happen to have compiled the following Java class:

```
public class MyClass {
    public String toString () {
        return "Hello, world!";
    }
```

then the `Execute` program will print "`Hello, world!`". On the other hand, the random command

```
java Execute xyzzy
```

will most likely just get you the message

```
Something went wrong with xyzzy.
```

You should be able at this point to figure out how to use a similar procedure to take a string, N, read from the configuration file and use it to create an object whose type is named `voting.plugins.`N. Here, `voting.plugins` is a *package* in which you can put classes for input, voting, checking, etc.

This leaves the problem of doing something useful with the resulting object (reading input, counting votes, etc.). The template files we supply for this project define a set of Java interfaces, which define the methods that each kind of object must supply. Your program (that is, the thing that loads these plugins), only needs to know that the objects it creates implement the appropriate interfaces. To tell Java that an Object, `obj`, has a class that implements an interface *PluginInterface*, just write:

```
PluginInterface plugin = (PluginInterface) obj;
```

This either works, or, if `obj` happens *not* to implement `PluginInterface`, causes a runtime exception (which, as a careful programmer, you make sure to catch and handle appropriately).

For example, suppose that `PluginInterface` is defined

```
public interface PluginInterface {
   String myName ();
}
```

We could modify `Execute`, above, to replace the first `println` with

```
        PluginInterface plugin = (PluginInterface) obj;
        System.out.printf ("My name is %s%n", plugin.myName ());
```

Now if `MyPlugin` is defined

```
public class MyPlugin implements PluginInterface {
   public String myName () {
      return "John Q. Plugin";
   }
}
```

then `java Execute MyPlugin` will print "My name is John Q. Plugin". whereas `java Execute MyClass` will cause an error.

5 What to Implement

You are to implement the framework that makes all this work as described, along with a few particular plugins, described below. We have provided some skeleton files for you to start with. They are available via Subversion. To set up your initial directory, use the following commands (assuming `~/svn` is your Subversion working directory):

```
> cd ~/svn
> svn mkdir proj1
> svn copy svn+ssh://cs61b-te@nova.cs/public/proj1/trunk proj1
> svn commit -m "Initial files for Project 1"
```

Use `svn copy` rather than `svn checkout` because you want to be able to change and commit all these files, and you are not allowed to change the public files (for obvious reasons). The commands above will set you up with the usual directory structure. If (or perhaps I should say when) we update the public files with corrections, etc., you can use the procedures described in the "Merging" section of the Subversion chapter of Reader #3 to include these changes in your files. That is, we'll keep all the released versions of the Project #1 skeleton files around in the repository directory `.../public/proj1/tags` as subdirectories `v0`, `v1`, etc.

The trunk file contains a README... file that indicates which parts of the skeleton files should *not* be changed (because they are relied upon by "third-party" plugin implementors, so they are part of the specification of your program, in effect. Otherwise, however, you are free to change or discard anything we've given you, as long as you meet the specifications in this project description.

In addition to the bare framework, you are to implement the following plugin modules, which go in files `voting/plugins/`*NAME*`.java`:

voting.plugins.StringInput This is an input module, which the user can name in the `input-method` configuration parameter, and which implements the interface `voting.InputMethodPlugin`. Instead of reading from a file, this module uses the command-line arguments directly. So, for example, with `(input-method StringInput)` in the configuration file *C*, running

> `java voting.Elect` *C* `"2 1 3 3" "4 3 2 1" "3 2 1 4"`

"reads" the three ballots '`2 1 3 3`', etc. Not very useful for real elections, of course, but perhaps one might use it for debugging.

voting.plugins.NamedRankedFormat This is a format module, which a user can name in the `ballot-format` configuration parameter, and which implements the interface `voting.BallotFormatPlugin`. It describes another format for ballots in which the candidates' names appear after their rankings, like this:

> `3 Zach Ecru 2 Josh Black 1 Aaron Cyan 3 Brian Dun`

In this format, candidates may be listed on a ballot in any order. At the time it is initialized, this module checks that no candidate's name contains any numerals. It indicates an error whenever a name on a ballot does not match any candidate, when a rank number is not between 1 and the number of possible ranks, or when there is some sort of syntax error (like a missing rank number).

voting.plugins.SingleTransferableVote This is a voting-system plugin, which the user can can name in the `voting-system` configuration parameter, and which implements the interface `voting.VotingSystemPlugin`. This voting method (used in the Republic of Ireland, for example) allows for multiple winners (the number given by the "slots" configuration parameter), and is intended to avoid "wasted votes". First, our version checks each ballot to make sure that no two candidates have the same ranking, discarding erroneous ballots. The module signals an error on initialization if the number of candidates is less than the number of slots. Suppose there are S slots and N ballots. We define the threshold number of votes to elect a candidate to be $T = \frac{N}{(S+1)} + 1$. That is, any candidate with at least this number of votes is elected (yes, fractional numbers of votes are possible in this system). Initially, each ballot counts for one vote. We repeat the following procedure until all candidates are eliminated or all slots filled:

1. Find the current number of votes for each candidate by adding up the number of votes on each remaining ballot that lists the candidate first among the remaining candidates.
2. If there is a candidate who has $V \geq T$ votes, then declare that candidate elected. We're done if all slots are now filled.
3. The quantity $V - T$ is the *surplus* of votes the candidate received. Compute the fraction $f = (V - T)/V$. Multiply the number of votes for each ballot that listed the candidate first by f. Eliminate that candidate from further consideration on all ballots.
4. Otherwise, if no candidate is over above threshold, eliminate the candidate that places last (got the fewest votes). Eliminate that candidate from all ballots and discard all ballots that now have no candidates left. If more than one candidate ties for last, choose the one who was listed first in the configuration file.

This procedure can fail when all candidates are eliminated before all slots are filled. This can happen when some ballots don't list all candidates.

6 What to Turn In

You will need to turn in all your Java files, a user manual, an `INTERNALS` file, and an assortment of JUnit test classes with names ending in `Test` (e.g., `ElectTest.java`), including one particular class named `FullTest`, which should run *all* your individual JUnit tests as a *suite* (see §3.4 of *Pragmatic Unit Testing*). Our script will run `FullTest` in the JUnit framework, expecting it to pass. Part of your grade depends on the thoroughness of your testing. JUnit, by the way, is perfectly capable of testing main programs, since in Java you can call the `main` method of any class as an ordinary method, supplying the command-line arguments directly as an array. We've helped by doing the annoying work you need to capture the standard output to a String. Finally, turn in any additional files you need (for example, input files for the JUnit test of the main program).

The user manual has two parts. The first describes how an election official uses the program. It tells about the configuration file, the plugins that are available, the format of ballots, and how to run an election. The second describes the system for the plugin writer: how to write a new module and what it must do. We've specified most of this in this project description, but we want your paraphrase. In part, this exercise is intended to get you to really understand our specification.

The purpose of your `INTERNALS` file is to give a description of the structure of your project to future maintainers—people who take over the code from you and go on to fix (the few remaining) bugs or add enhancements. This description will give them a good idea of everything is supposed to work together, so that it is easier for them to read and understand your code. Since we've constrained the internal structure of this project pretty much, this time you will essentially be describing the structure we've given in your own words, plus any extra mechanisms you might decide to add. You don't need to echo all the comments: those details that are easily understood just by reading the comments on methods needn't be duplicated in `INTERNALS`, which is more of an overview.

We will expect your finished programs to be workmanlike—consistently indented, well-commented, readably spaced. Don't leave debugging print statements lying around. In fact, don't use them; learn to use the debugger (either `gjdb` or that of `Eclipse` or your favorite Java-system vendor).

7 Advice

First, get started immediately, of course. Don't just jump in and code, though. There should not be a huge amount of coding to do for this project, but there is a good deal of code *reading* you'll have to do to understand just what we're looking for.

Read the skeleton files and *understand* as much as possible. Don't allow things to remain mysterious to you, or they'll surely bite you at some point. We've put lots of new stuff in the skeleton files precisely to get you to ask questions and (especially) to browse the Java library documentation.

I suggest you write the user manual early on. First, it gets that task over with. Second, by restating the problem, you might get to understand it better. The same comment goes for the `INTERNALS` manual.

The JUnit book we assigned describes a test-early-and-often attitude. You can write tests of the overall system *before* you write a line of code.

Above all, it is always fair to ask for help and advice. We don't *ever* want to hear about how you've been beating your head against the wall over some problem for hours. If you can't make progress, don't waste your time guessing or bleeding: ask. If nobody's available to ask, do something else (or get some sleep).

8 Appendix: Arrow's Impossibility Theorem

Isn't there a perfect voting system? Alas, no. If we think of a voting systems as a way to take a bunch of ranked ballots (where each voter orders the candidates from first to last) and deterministically combine them into an overall societal ranking of all the candidates, then you'd probably agree that ideally, the following conditions should hold:

1. All ballots are counted equally; no one voter dictates the outcome.
2. If all ballots rank candidate A above B, then so does the resulting overall ranking.
3. If one removes any candidate from all ballots (keeping the other candidates in the same relative order on each) then the effect on the overall ranking is also simply to remove that candidate without affecting the relative order of the others.

These all *sound* reasonable, but Kenneth Arrow proved that for more than two candidates, all systems that rank voters as we've been showing here must violate at least one of these criteria for some set of ballots.