Designing Modular Vote-Tallying Software

Greg Dennis
gdennis at mit dot edu

Introduction

Interest in alternative voting methods appears to have surged in recent years in the United States. Since January 2003, Congress and 21 state legislatures have considered adopting systems other than plurality voting.[1] Most recently, on Election Day 2004, San Francisco used a system known as Instant Runoff Voting (IRV), in which voters rank candidates in order of preference, for their city and county elections. On that same day, Ferndale, Michigan, Burlington, Vermont, and 16 Massachusetts towns approved ballot measures endorsing IRV.

Although plurality is still the dominant voting system in the United States, other voting methods such as Single Transferable Vote, Borda Count, and Cumulative Voting are currently used to elect dozens of local offices, student council governments, and the leadership of corporations and organizations around the country. Outside the US, non-plurality voting systems are even more prevalent. In fact, of the 211 states and territories across the globe that have direct legislative elections, only 68 primarily use plurality voting.[2]

Jurisdictions and organizations face financial and logistical burdens when adopting a new voting system. These hurdles can include the purchase of new software built specifically for the new voting method and, in turn, the obligation to train employees to operate that software. If the available software is not quite adequate—for instance, if it does not support the specific version of the voting method that the jurisdiction requires—then the jurisdiction has two options. It could request that the manufacturer upgrade the program, which risks layering additional complexity into the design in an ad hoc, error-prone way. Alternatively, the jurisdiction could specially commission new, and possibly insufficiently tested, software to suit its needs. Neither the ad hoc upgrade nor the commission of brand new software is appealing. Each incurs significant cost and decreases reliability.

However, if the vote tallying software is designed appropriately, a change in voting method need not be a burden. In the sections that follow, I present three design features for developing a modular vote tallying system, which can accommodate a wide range of voting methods and their variations. The proposed architecture would allow new voting algorithms to be easily "plugged-in," avoiding the costs and risks of ad hoc upgrades and replacement software. Furthermore, the modularity would be conducive to rigorous unit testing and verification of the system's individual components. In addition, the strong decoupling between modules would result in code that is easier to understand, inspect, and maintain.

Ballot as Multilist

A ballot in a modular vote tallying system is best represented as a multilist, a list where not one, but a *set* of elements can be located at each index. A multilist data structure can represent a ballot for virtually every known voting method. For ranked systems, each index in the multilist can represent a rank on the ballot and the set of elements at an index is the candidates voted for at that rank. For example, a multilist ballot in which candidates A and B are at first index, candidate C is at the second index, and candidates D and E at the third index, represents the preference of a voter who prefers candidates A and B the most, candidate C less, and candidates D and E the least.

With *single candidate methods*, systems that involve the selection of at most one candidate, votes will only appear at the first index of the multilist—the first rank—and valid votes will be limited to at most one candidate at this rank. These voting methods include single-winner Plurality Voting, the Party-List Voting systems popular in Europe, and a multi-winner system known as Single Non-Transferable Vote (SNTV) that is used in Japan, Taiwan, and Puerto Rico.

A number of voting systems ask voters to assign a yes-or-no preference to a subset of the candidates on the ballot. These *binary voting methods* would make use of only a single index of the multilist but permit multiple candidates at that index. If a voter can select a number of candidates equal to the number that will be elected, the system is commonly referred to as Block Voting, or Plurality-at-Large, and is a common method of selecting at-large city councilors in the United States. Other binary systems include Limited Voting, a system in which a voter can select some fixed number of candidates smaller than the number elected, as well as Approval Voting, in which a voter can select any number of candidates.

By limiting ourselves to selecting one candidate per index, but utilizing multiple indices of the multilist, we can accommodate many of the *ranked voting methods*. Such systems include Single Transferable Vote (STV), the single-winner version of STV called Instant Runoff Voting (IRV), Borda Count, and Condorcet. Ranked systems will typically use as many ranks as there are candidates, but in some variations only a smaller number of rankings are permitted, such as the Supplemental Vote system used to elect the mayor of London, which provides only two ranks.

Voting Methods	# Ranks	# Candidates / Rank
Plurality, Party-List, SNTV	1	1
Block, Limited, Approval	1	>1
IRV/STV, Borda Count, Condorcet	>1	1
Cumulative, MCA, CR, ranked with equal preference	>1	>1

Figure 1. Voting Methods Usage of a Ballot Multilist

There are a number of voting systems that will require both multiple ranks of the multilist and the ability to vote for more than one candidate at each rank. Among these is a multi-winner voting method known as Cumulative Voting. In the United States, cumulative voting is probably best known as the system used to elect the Illinois state legislature from 1870 to 1980. Under this method, each voter is given some fixed number of votes to proportion among multiple candidates. For example, if provided five votes, a voter could place all five on a single candidate, or one on each of five candidates, or perhaps three on one candidate and two on another. To accommodate cumulative voting, the multilist should contain as many ranks as there are total votes. Then if a voter allocates three votes to a given candidate, the candidate will appear at rank three in the list. If a voter allocated one vote to each of four different candidates, then all candidates would appear at the first rank in the list.

There are two noteworthy properties of these cumulative voting ballots when represented as multilists. First, many of these multilists will have gaps in their rankings, e.g. a vote for candidates at rank 1, none at rank 2, and some at rank 3. Second, all valid cumulative voting ballots will exhibit the important invariant that the sum of the candidates' individual ranks should be less than or equal to the total number of ranks on the ballot. That is, it is invalid for a voter to allocate more votes than was provided.

In addition to Cumulative Voting, a multilist ballot also accommodates *rated voting methods*. Rated voting methods are systems whereby each candidate can be assigned a score from a set of possible scores. In Majority Choice Approval (MCA), for example, each candidate can be labeled as "disapproved," "accepted," or "favored." To represent this as a multilist, each label corresponds to a rank, and the candidates voted at that rank are those that were assigned that label.

In another rated method known as Cardinal Ratings (CR), or range voting, each candidate is assigned a particular numerical score. Popular uses of CR include the star ratings given to movies and restaurants. In theory, CR could ask voters to rate a candidate from an infinite set of scores, in which case we would be unable to associate each rank of a finite multilist with each possible rating. However, in practice the set of scores is always bounded and finite, so a multilist representation will suffice.

Finally, a complete multilist ballot also supports versions of ranked systems—IRV/STV, Borda Count, and Condorcet—in which multiple candidates can be assigned an equal preference. If two candidates are preferred equally, then they would both appear at the same index in the multilist. Though I am not aware of any jurisdiction or organization that uses such a voting system, they have generated serious academic discussion amongst social choice theorists.

To summarize, different voting systems impose different requirements on the ballot. Some methods require a ballot to hold only a single candidate, some require a set of candidates, some require a list of candidates, and still others require a list of sets of candidates. As demonstrated, a multilist data structure is sufficiently general to support all such systems.

Strategy Design Pattern for Tallying

It is common for a software system to invoke different algorithms for the same task under different conditions. Consider the task of sorting, for example. In some scenarios, a client module may need to invoke a sorting algorithm that is stable; at other times stability may be unnecessary. On some occasions the client may have reason to believe a list is almost sorted, which could make bubble sort an advantageous algorithm; other times, it may prefer merge sort, quick sort, or possibly insertion sort.

One way to provide a client with the option of invoking any of these algorithms involves implementing each of them directly in the client module. Then, at every point where sorting is to be performed, the client performs a series of conditional tests to determine which algorithm it should invoke. The downside of this approach is that implementing every algorithm in the client module tends to make the client monolithic, complex, and hard to maintain. Also, whenever a new algorithm is required, it becomes necessary to further disturb and complicate the code, not only by implementing the algorithm itself, but also by extending all of the conditional constructs that may be scattered throughout.

The preferred technique for addressing this problem is the *Strategy* design pattern. According to the "Gang of Four" book, a definitive source on software design patterns, one uses the Strategy pattern to "[d]efine a family of algorithms, encapsulate each one, and make them interchangeable."[3] "Strategy," it continues, "lets the algorithm vary independently from clients that use it."

When implementing the Strategy pattern, a `Strategy` interface is first defined. This interface describes the behavior common to all of the algorithms the client must support. Next, this `Strategy` interface is implemented by a series of `ConcreteStrategy` classes, one to encapsulate each of the desired algorithms. Once configured with an arbitrary `ConcreteStrategy,` the client can invoke the behaviors provided in the `Strategy` interface on that concrete strategy. By writing new implementations of the `Strategy` interface, additional algorithms can be provided with ease. An object model of the Strategy design pattern is found in Figure 2.

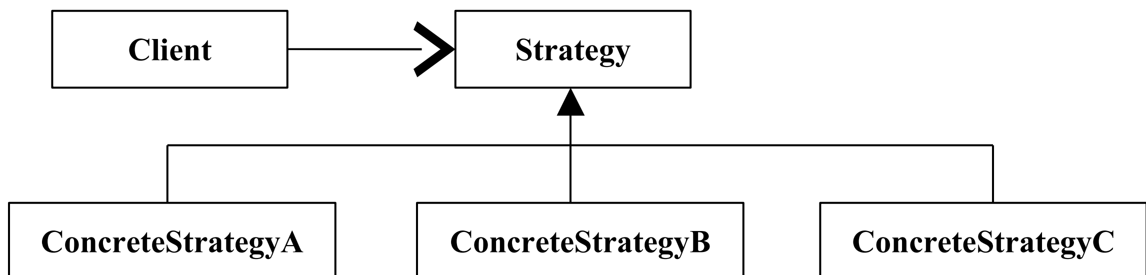

Figure 2. Object Model of the Strategy Design Pattern

Applying this pattern to accommodate the various tallying algorithms is fairly straightforward. We first define a common interface to the algorithms, which we'll call `TallyStrategy`. This common interface would presumably accept a set of ballots as an

input and yield the results of the election as an output. Next, an implementation of the `TallyStrategy` interface is implemented for each of the tallying algorithms, such as Plurality, Approval, STV, Cumulative, and so on. Now a client can be configured with an arbitrary concrete tally algorithm, to which it can provide a set of ballots and retrieve the results, without a direct dependency on the particular algorithm it invoked. An object model of the Strategy design pattern applied to tallying algorithms is found in Figure 3.

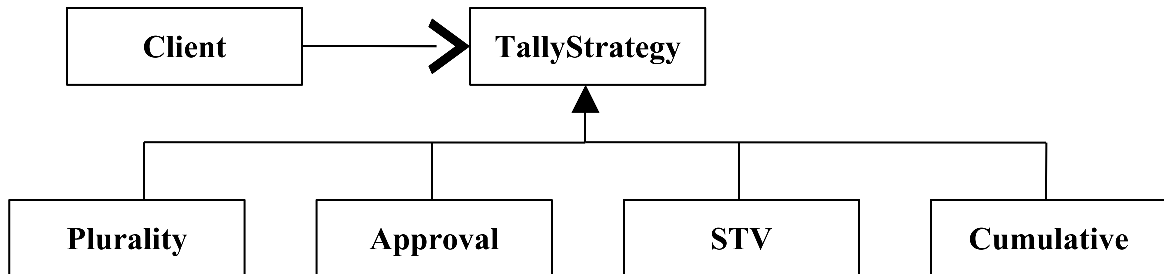

Figure 3. Strategy Design Pattern Applied to Tallying Algorithms

Many of these tallying algorithms come in several variations. Versions of STV, for example, will differ from one another in the algorithm they use for determining which candidate(s) to eliminate in each round and in their method of transferring surplus votes from winning candidates. Similarly, there are several different Condorcet methods, each of which uses a different technique for determining the winner when the core algorithm fails to find one.

To accommodate the many versions of each voting algorithm, the Strategy pattern should be re-applied to each point of variation. In STV, for example, the candidate elimination algorithm and the vote transfer algorithm should each be abstracted away into separate Strategy interfaces, each of which may have several implementations. Likewise, each of the techniques for completing the Condorcet method should be encapsulated into concrete strategy classes, all of which implement a common interface.

To some, utilizing the Strategy design pattern for this task may seem obvious. However, inspection of at least one commercial vote tallying program finds this to not be the case. The program in question is ChoicePlus Pro, an open-source program for tabulating ranked ballots manufactured by Voting Solutions.[4] According to a 2001 press release, ChoicePlus Pro is the "only computer program that has ever been used to count ranked-ballot style government elections in the United States."[5] ChoicePlus Pro can tally votes according to several variations of Single Transferable Vote, including Instant Runoff Voting.

Instead of encapsulating each of the variations in STV into Strategy objects, the software implements all of them side-by-side, with frequent conditionals to choose which one to use. As a result, their code is far more complex than necessary. I do not intend to single-out this one program or this one company; ChoicePlus Pro just happens to be the only piece of commercial vote counting software whose source code is public. It does suggest, however, is that the use of the Strategy pattern for this purpose is, for some commercial developers, a non-obvious design choice.

Results in Rounds

Reporting the results in an algorithm-independent way is perhaps the most difficult challenge of developing modular vote tallying software. Simply showing the number of votes each candidate received does not suffice. Under STV/IRV, for example, one would expect to see each round along with which candidates were elected and defeated in that round. With Condorcet, one would expect to see the results of each pair-wise election. Most methods assign an integral number of votes to each candidate, but with versions of STV where votes are transferred fractionally, the number of votes is not necessarily an integer. Other systems don't actually distribute "votes" amongst candidates, but *points* according to some predetermined scale.

An architecture that supports all these systems can be designed as follows. Consider the results of an election as a sequence of *rounds*. Every round contains a rational number score that each candidate received in that round. Each round also labels every candidate as a "winner," a "loser," or "undecided." The winners, losers, and "undecideds" found by the complete tallying algorithm are those labeled as such in the final round. The final round can still have undecideds, because nearly every method has a scenario in which winners cannot be determined. For example, most systems do not select a winner when two candidates receive the exact same number of votes. To enhance the user's understanding of the results, each round is supplied a title and descriptive text, as well.

For some systems like Plurality, Approval, and Borda Count, there will likely be only one such round. STV and IRV, on the other hand, will often require multiple rounds before enough candidates reach the necessary threshold to be elected. In Condorcet, each round can be a pair-wise election, followed by one or more rounds to interpret the ultimate winner from the pair-wise contests. The title and description of the rounds would explain their purpose and significance to the user.

Conclusion

In summary, I've proposed three design features to facilitate the development of modular vote tallying software:

- multilist data structure to represent a ballot
- strategy design pattern to encapsulate vote tallying methods
- sequential rounds of tallies to represent the results

The object model of such a modular vote tallying would resemble the diagram presented in Figure 4. If implemented correctly, such a design will provide a robust, flexible, and intuitive architecture for counting the votes via every existing, and most every conceivable, vote tallying method.

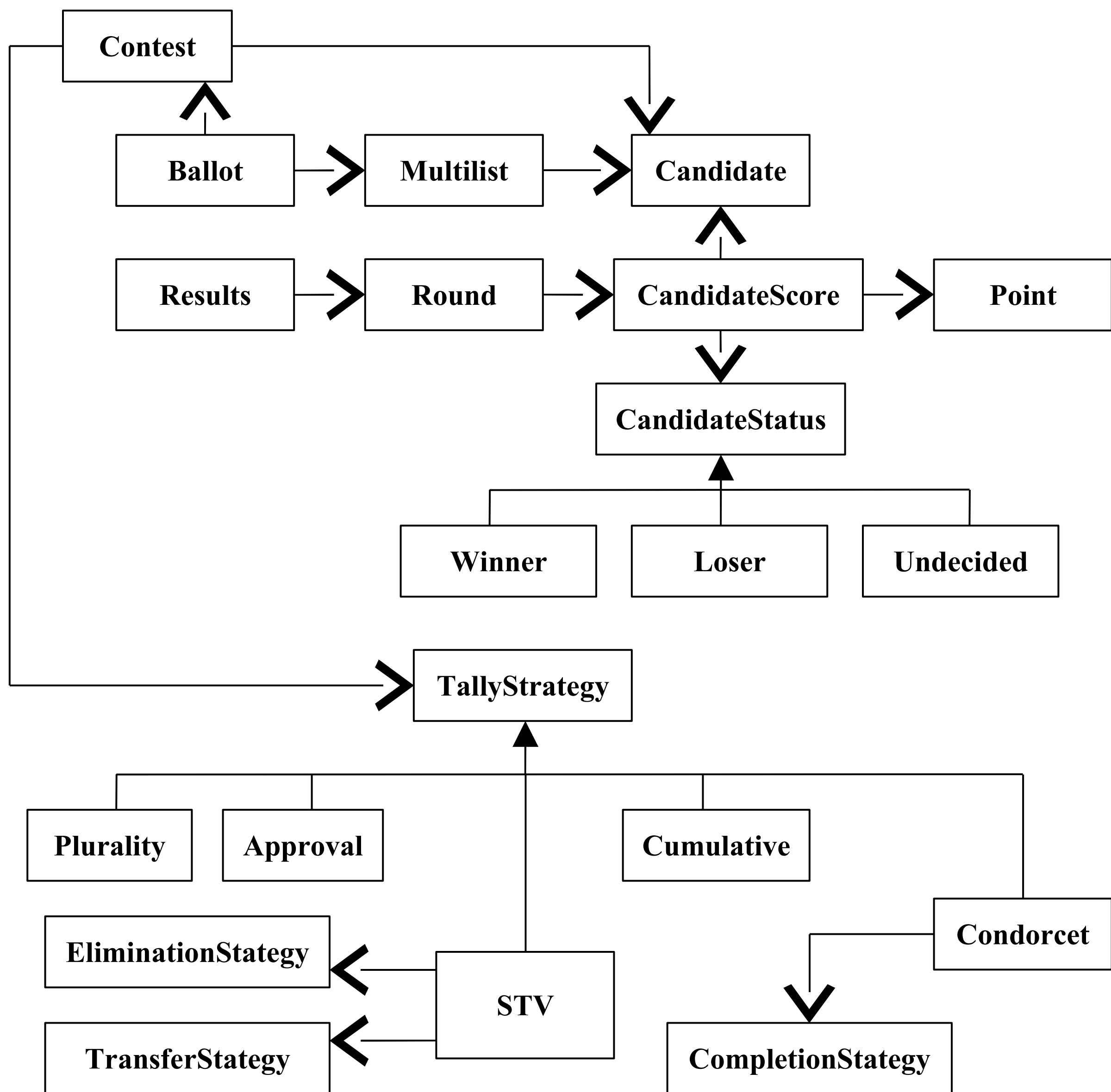

Figure 4. Object Model for Modular Vote Tallying Software

[1] "Pending Legislation and Ballot Measures." Center for Voting and Democracy. http://www.fairvote.org/action/index.html
[2] "Full Representation Around the World." Center for Voting and Democracy. http://www.fairvote.org/pr/global/index.html
[3] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading Mass., Addison Wesley: 1995.
[4] Voting Solutions. http://www.votingsolutions.com
[5] "City of Cambridge upgrades vote counting software with ChoicePlus Pro." *Business Wire*. 15 October 2001.