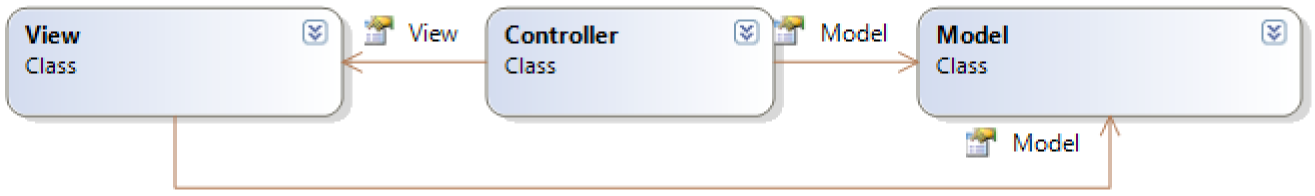
System Architecture

Overall Pattern

The Model-View-Controller (MVC) pattern forms the core structure of our system architecture. We have chosen to do this because we think it forms a solid backbone with regards to the state and extendability of the system.

MVC places the state and logic of the system in an independent *Model*. The *Model* can focus on getting the task done and maintaining a correct state without worrying about how it is going to be represented to the user. Eventually the *View* and *Controller* is built to fit the *Model*. The *View* mimics the state of the *Model* and presents it to users. It also allows users to send queries and commands. The *Controller* propagates these commands and queries to the *Model* in a suitable way.



The message passing between the 3 actors is crucial and it is implemented via the **Observer Pattern**:

- The *View* observes the *Model* (Notifications about change of state)
- The *Controller* observes the *View* (Notifications about user input)

This principle is very well supported by the C# syntax via events, delegates, lambdas and the like. It is also the pattern used throughout Windows Forms allowing forms to play the role of *View* directly.

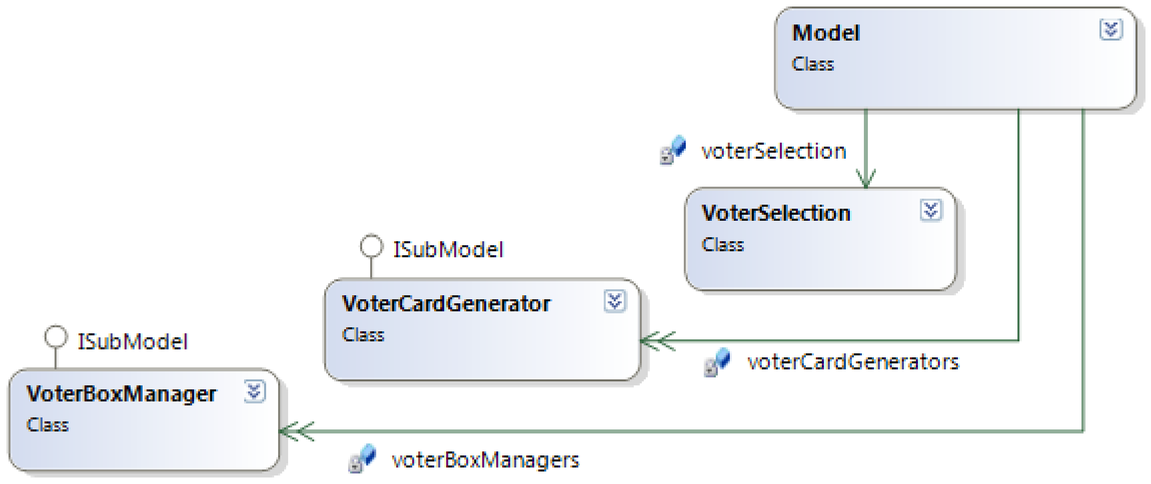
Sub-Models

Our ‘Central’ system has been divided into 3 very different sub-systems regarding manipulation of voter data:

- Voter Selector
- Voter Card Generator
- Voter Box Manager

This makes it possible to run simultaneous tasks. The overall system is built to support easy addition of further sub-systems. Each sub-system has it’s own implementation of the MVC pattern.

‘Central’ has 3 overall *Model*, *View* and *Controller* classes responsible for instantiating these sub-systems. One could imagine an additional set of View and Controller being added and they should be able reproduce the currently active sub-systems appropriately based on the state of the *Model*.



Closing a sub-model

When a sub-model is ordered to shutdown this stepwise process follows:

1. *Controller* tells *Model* that a user has ordered the sub-model to shut down.
2. Reference to the sub-model is dropped in *Model.*
3. *View(s)* are notified and their corresponding sub-views are closed.
4. There are no longer any references to the sub-model, sub-view(s) and sub-controller(s) so the entire sub-system should be garbage collected.

This illustrates the loose coupling despite imaginable scenarios containing a vast network of sub-models, sub-views and sub-controllers. There can be any number of Views subscribing to the *Model*, it won't make a difference in terms of the state and operation of the *Model*.

Testing the overall architecture

We have made a number of manual unit tests that verify instances of the overall MVC implementation. They attempt to instantiate different object scenarios and verifies that the right notifications are given and that the resulting states in the different modules are as expected.