After experimenting, we designed the game based on the classic Model View Controller (MVC) design pattern that was originated with Smalltalk. A design pattern describes how objects with well-defined roles can interact and solve well-known problems. Early on, the MVC pattern became a way to develop graphical user interfaces where multiple views display updatable information. Views, and the information they contain, may change as the user interacts with any number of interface elements. You can read more about the Smalltalk MVC design pattern in an article by Steve Burbeck.

Since Smalltalk, the MVC pattern has been replaced by other related approaches to building event-driven graphical user interfaces. However, MVC has enjoyed a resurgence as a way to build web applications. For more information, see Sun's
The MVC pattern seemed like an obvious design choice because we wanted to maintain the state of the game on the server. Therefore, each player's client must have a view into the state of the game resident on the server and must send messages to the server to manipulate the game. When the game state changes (for example when a square is uncovered), the application must update every player's view simultaneously. In our design, there are three primary objects that correspond to the model, view, and controller of the MVC triad shown below:

**Figure 1.** The MVC triad for the Minesweeper game

The Game View
The game view is an object that exists on the client in the Macromedia Flash Player. When a user interacts with the game view (for example by clicking on a game square), the application passes an event to the game controller on the server. On the other hand, the game view receives notifications that the game has changed from the game model and then correctly displays the game board as it should appear to the user.

The Game Controller
When the user interacts with the game view it sends a message to the game controller on the server. In turn, the controller will decide how to handle the message. It may ask the game model to carry out an action such as uncovering a square. This in turn may result in the application updating the client game view object.

The Game Model
The game model might also be called the game board. It maintains the game state and notifies all the game views of changes in the game state. For example, when a square is uncovered, the game model must broadcast a message to all the game views that the square is now visible and display the uncovered square. The game model is an object that the application instance creates on the communication server. There is one application instance for each game being played.

Communications Between Objects
In MVC, when the model changes, the application must broadcast a message to each view that the model has changed. That way, each view can accurately present model information. In our case, the model contains a game board. Changes to the visible part of the board are broadcast to the game view on each client. Flash Communication Server provides a number of ways to do this:

1. You can store the visible part of the model in a remote shared object. Whenever a property of the shared object changes, the client receives an onSync event that tells it to update the board view for the user. In turn, the view examines the shared object and updates how it displays the board.
2. You can use a shared object as a communications channel without actually storing any data in it. The shared object's send method can be used to broadcast method calls containing messages from the server to all clients. When send is used, the application calls a method of the remote shared object for each user. If enough data is passed in this way, the client will not have to query the server for information about updating the view.
3. Similarly you can use NetStream object's send method for the same purpose.
4. Finally, the server can loop through each client object in the application.clients array and use the call method of each client to call a client-side method.

In our implementation, we used a playerList shared object to provide each client with information about all the players participating. When the game model needs to inform all the attached clients that one or more squares on the board have been uncovered, the server-side script uses the player list shared object's send method. For example when the server-side application wants to tell all the clients that one user has uncovered a bomb:

```javascript
application.playerList.so.send('openBomb', square, player);
```

Where openBomb is a client-side method of the player list shared object that has been defined in the Flash movie (shown below):

```javascript
playerList_so.openBomb = function(square, player) {
    gameView.showPlayerLostScreen(square, player);
}
```

This, in turn, passes its data on to the gameView object. The gameView will display a bomb to the user in the right square. That is how the model of the game on the server informs the views of the game in each Flash movie to make a change.

But, what happens when a user clicks on an unopened square? The client must send a message to the server to update the game model. In this case it was not necessary to broadcast the message. It only has to go to the server, so we used a remote method call. We used the call method of the NetConnection object to make a remote method call. In our case, we used a NetConnection object named game_nc:

```javascript
game_nc.call("openSquare", null, (this._name).substring(3));
```

This calls a method named openSquare on the server and passes it the name of the square to open. Defining the openSquare function on the server involves using the server-side client object. There is one client object created on the server for each Macromedia Flash movie that connects to it. When a movie uses a NetConnection object to call a remote method, it calls a remote method of the server-side client object that represents it. One simple way to assign an openSquare method to each new client object as each movie connects to the server is as follows:

```javascript
application.onConnect(newClient, playerName) {
    newClient.playerName = playerName;
    newClient.openSquare = function(square) {
        gameController.requestToOpenSquare(square, this.playerName);
    }
}
```

In summary, when the user clicks on an unopened square, the application calls a remote method on the server. The remote method asks the server-side game controller object to open a square. The game controller passes the request to the game model. The model updates the game board and broadcasts any changes in the board to all the Macromedia Flash movies attached to the game, thereby updating each movie's game view to the current state of the board.

Here is a somewhat oversimplified picture showing a client-server interaction. If you aren't familiar with interaction diagrams, they are designed to show the order of interactions over time. Time increases as you move down the diagram. Each arrow represents a method call. First the user clicks on a square, invoking the onRelease handler and so on down the drawing. This particular drawing shows what happens when a player uncovers a square with a mine under it.
Figure 2. Interaction diagram showing the events triggered by a user opening a square with a bomb.

In the next section, we'll cover some of the other classes used in the game.
Of course, other classes are required in our design, which also follow the MVC pattern where possible. Player objects are created for each player connected to the game and are stored in a player list. The application broadcasts changes in the player list to all clients so that all players can see changes in player status immediately, such as players dropping out of the game. A separate player controller object handles player connections, requests to start the first game, restart a game, or disconnect from a game in progress. The player controller object can easily check the state of the game and respond to user requests accordingly. For example, once the game starts, the player controller will not allow any new players to join the game. The player view is a visible panel in the client that allows the user to connect to a game, leave a game, and see
information about other players.

Finally, our game has an audio chat system to allow players to discuss the game as it progresses. Players can decide who will hear their remarks and can mute sound from any player. We built the audio chat on top of information in the player and player list; the player controller controls it.

The following section provides a more detailed description of some of the classes that make up the game and how they work.

The GameModelClass
The game model maintains all the state information about the game and provides methods that correctly manipulate the squares in the board. It contains information about the size of the board, an array of square objects, and anything else it needs to initialize, play, and end a game. The game logic is also contained within this object. The game model contains an instance of the BoardClass where it processes most of the game logic. For example, when it receives a request to uncover a square, the game model will first check if the square is already uncovered. If it isn't uncovered, it will mark who has uncovered it (by color) and then uncover the square. If the square is blank it will recursively uncover all the neighboring squares until no adjacent blank squares remain. When the uncovering process finishes, the application broadcasts a message listing the results. There are a number of ways to implement the game board and recursive uncovering of squares.

In this game, the board is a simple one-dimensional array. As an example of how this works, you can represent a simple three-by-three board with an array with 9 slots numbered 0 through 8 that map to a board as illustrated here:

```
0 1 2
3 4 5
6 7 8
```

To begin a game, you must randomly place bombs on the board. There are a number of ways to do this; here, we created a temporary array containing the number of each square:

<table>
<thead>
<tr>
<th>Temporary array:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

As positions in the temporary array are randomly selected as bomb locations, two things happen. First, you add the bomb location to another array called the map array by placing an x in the slot and placing the number of bombs in the surrounding squares. In this example slot number 5 was randomly selected to hold a bomb and the surrounding squares given the value 1:

```
0 1 1 1 x 1 1
```

Which represents this board layout:

```
0 1
1 x
1 1
```

Second, you remove slot number five from the temporary array using the Array.splice() method:

```
tempArray.splice(randomCell, 1);
```

<table>
<thead>
<tr>
<th>Temporary array:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>---</th>
</tr>
</thead>
</table>
Reducing the size of the temporary array this way makes it easy to choose another random cell without a bomb in it:

```javascript
randomCell = Math.floor(Math.random() * tempArray.length);
```

After you repeat this process and place all the bombs, users could begin the game. However, there is a problem with converting minesweeper to a multi-player game. In the single-user version, the first square uncovered cannot contain a mine. One way to implement this is to secretly move the mine to another square if the user clicks on a mine. In a multi-user version it didn't seem fair to let everyone try to click on a square with the possible result that after one square was uncovered others would contain a mine and end the game for those users. Instead we chose to uncover at least one square when the game begins. That way each player has a clear starting position to assess. Describing the entire game is beyond the scope of this tutorial. However, some methods of interest in the BoardClass are:

- populateBomb - uses a temporary array to randomly choose bomb locations
- getNeighbors - given a square location returns a list of neighboring squares
- preOpen - opens a square and its neighbors so that all players have somewhere to start
- openSpaces - recursively opens spaces

The source for the GameModelClass is in the GameModelClass.asc file while the source for BoardClass is in the BoardClass.asc file. Also see SquareClass.asc.

The GameModelClass coordinates user interactions and updates the views; it bases the updates on the current state of the game within the board object. The most complex part of this process is what the GameModelClass does when it receives an openSquare message from a movie. (See the openSquare method of the GameModelClass.) This process is complicated because many users may click on an uncovered square within a few milliseconds of each other. However, the first openSquare message that arrives at the server may not represent the first click in real-time. To understand this scenario, imagine two players playing the game. Player A connects to the server from outside Ryerson’s campus and has an average network latency of 120 milliseconds, or just longer than one tenth of a second. Player B is on campus and experiences an average latency of only 10 milliseconds or one hundredth of a second. The following sequence of events illustrates how an unfair situation could happen:

<table>
<thead>
<tr>
<th>Event Sequence</th>
<th>Time of Event (milliseconds)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
<td>Player A clicks on an unopened square one second into the game.</td>
</tr>
<tr>
<td>2</td>
<td>1100</td>
<td>Player B clicks on the same square one tenth of a second later.</td>
</tr>
<tr>
<td>3</td>
<td>1110</td>
<td>Player B’s openSquare request arrives at the server 10 milliseconds later, which uncovers the square.</td>
</tr>
<tr>
<td>4</td>
<td>1120</td>
<td>Player A’s openSquare request arrives at the server 120 milliseconds after it was sent but the square was already uncovered by Player B.</td>
</tr>
</tbody>
</table>

In this unfair situation, player B gets credit for opening a square even though player A clicked on it first in real-time.

Off-campus users may experience much higher network latencies, which introduces an unfair disadvantage due to the system. To reduce unfairness, the application measures each client’s network latency and estimates the real-time for each user’s click on an unopened square. If user A’s request to open a square arrives after the square has already been opened, user A is assigned credit for opening the square. In the current version of the game, we only measure network latency by making a remote method call after getting a current date/time. The server simply returns from the method call and compares a new date/time to the old one. The application determines an approximate latency by dividing the round-trip delay in half.

We describe a better way of doing this later in a related latency problem. In the case where we need to correct who will get credit for uncovering a square, we send a “correctGame” message to all the movies:

```javascript
application.playerList_so.send("correctGame",
    this.gameBoard.mapArray[square],
    square, removeFromPlayer);
```
All GameModelClass methods that communicate with views use the send method of the playerList_so shared object. For example the restart method contains this statement:

```javascript
application.playerList_so.send("reStartGame");
```

And here is another example from the openSquare method:

```javascript
application.playerList_so.send("openSpaces", str);
```

In this case, the openSpaces method of the player list shared object will call the openSpaces method of the game view. The string passed will contain a comma separated lists of square locations and the number to display in each of them.

**GameControllerClass**

An application may contain many MVC triads. Each controller does not maintain state or store data. It simply manages messages as they arrive. When there are multiple MVC triads at work, the controllers cooperate by passing messages to each other in order to ensure that they update the correct model and views from user events. In this game, using this process turned out to be unnecessary. The game controller simply passes all its messages directly to the game model. In more complex applications cooperating controllers may have some value. Here is a typical game controller method:

```javascript
GameControllerClass.prototype.requestToPreOpenBoard = function() {
    gameModel.preOpenBoard();
}
```

The source for the GameControllerClass is in the GameControllerClass.asc file.

**GameViewClass**

The GameViewClass is a movie clip subclass that shows the user what is happening in the game. It is responsible for drawing the board and dealing with messages sent from the game model. In turn the GameViewClass contains another movie clip subclass: the ClientSquareClass. A ClientSquareClass instance draws each board square and contains the usual graphics in its timeline for each state the square can appear: covered, opened, number, bomb, and disappear.

When the openSquares method of this class is called, it receives a string of squares to open and the values to place in each square. It splits the string into an array and alters the contents of each square accordingly.

Each square is only responsible for reporting when the user clicks on it. Here is the onRelease handler of the ClientSquareClass:

```javascript
square.onRelease = function(){
    //start the clicking sound.
    click_sound.start(0, 1);

    //send the info to the server.
    game_nc.call("openSquare", null, (this._parent._name).substring(3));
}
```

The source for the GameViewClass is in the GameViewClass.as file. Also, see the ClientSquareClass.asc file.

**The Player List, Controller, and View Classes**

Aside from the client object, which represents each movie's network connection to the game, we use a separate player object to keep track of each player's name, score, network latency, display color, and status on the server. (See the PlayerClass.asc file.) The player objects are stored in the PlayerList shared object. When players arrive or leave, the application adds or deletes the entry in the PlayerList through the PlayerListClass. Also, when a player's score changes, the class updates the player objects in the player list.

When the PlayerListClass adds or deletes a player to the PlayerList shared object, each movie automatically receives an onSync event. This is a feature of remote shared objects and provides another way that information can be broadcast to all the movies connected to an application. For each movie to receive and properly handle each onSync event, you must define an onSync event handler method for the shared object. The listing below shows the handler for the PlayerList. It receives a list of items that have changed whenever players add or remove themselves. Lastly, the class examines each item's code property to see what has happened to the item and uses the item's name field to identify the name of the shared object property that was, in this case, added or deleted.
playerList_so.onSync = function(list) {
    for (var i = 0; i<list.length; i++) {
        // on change Add the new player.
        if (list[i].code == "change") {
            if (playerList_so.data[list[i].name].name == playerName) {
                player = playerList_so.data[list[i].name];
            }
            var Tplayer = playerList_so.data[list[i].name];
            playerView.addPlayer(list[i].name, Tplayer.color);
        }
        // on delete then remove the player
        if (list[i].code == "delete") {
            trace("Deleting: " + list[i].name);
            playerView.info.info_txt.text += list[i].name + " Has left the game\n";
            playerView.removePlayer(list[i].name);
        }
    } // end for
}

Whenever anything visible must change, the process calls a method of the playerView object and in turn the playerView object hides or shows the movie clip that represents each player.

As was the case with the GameControllerClass, the PlayerListControllerClass just passes messages it receives to the player list which acts as the model in the player list MVC triad. See the PlayerListClass.asc, Player.asc, PlayerListController.asc and PlayerViewClass.as files.

The Lobby and Games
If you have tried the game, it placed you in a lobby before you could create or join a game. The lobby is a separate and very simple communication application. It lives in the flashcom/applications/mmsLobby folder. When the app creates a game in a lobby, nothing more than a name is created. The game is created and lives in the flashcom/application/OOMS folder when users attempt to connect to it. Each game is a separate instance of an OOMS game. For example players who play a game they name "challenge" actually connect to a game instance through an RTMP request: rtmp://hostname/OOMS/challenge.

The main timeline of the SWF file contains a lobby segment and a game segment and all the code required to connect to the lobby and the game.

In the last section, we'll discuss our conclusions and some of the possible enhancements to the game.
The game and the code presented in this article were a proof-of-concept experiment. It has provided us with a useful test bed for experimenting and learning. The game itself was never meant to be the ultimate multi-player version of minesweeper or to provide a tutorial on best coding practices. The following are some notes on what we learned from building the game.

**Messaging Resources**
The Macromedia Flash Communication Server MX provides a rich variety of ways to send and receive messages. These include:
Remote method invocation from client to server or from server to client is available using the call method of either the NetConnection or Client objects.

You can use shared objects to broadcast messages through the send method. You can also do so with the NetStream object (although we did not use it in this tutorial).

Shared object updates are reflected in changes in every copy of a remote shared object and the app generates onSync events to notify interested clients or the server of the changes.

Deciding how to pass messages is an important part of designing a communication application. Here are a few suggestions:

- Consider using remote shared objects to store application state data especially when you want users to be able to connect to an application at any time. Also use persistent shared objects to maintain state even when no users are connected.
- Use the send method of a shared object when you need to broadcast a message to all clients (and the server) and you don't need to receive result.
- Use the NetStream object's send method when you only want the stream publisher to send messages and the sender does not need to receive its own message. When a stream is recorded, send method calls are stored in the stream and replayed when a client plays the stream.
- Use remote method calls (using the call method on a NetConnection or Client object) when you want to send a message from one client to the server or from the server to a client. The call method has the advantage that a result object can be returned and it can target a specific client or list of clients.

Security Enhancements

A common problem with online games is that out of the millions of people who play them, a small number will attempt to cheat. A common example is when game players attempt to hack the high score messaging system to give themselves an earned status. Cheating is potentially a serious problem for the academic value of any automated marking or tracking system in educational software.

If you design educational software, you can discourage cheating by making it extremely difficult. The scores that the players see in our game are stored in a remote shared object. An enterprising player could attempt to hack a copy of the game in order to write a different set of scores into the shared object. There is a simple way to prevent this problem. While not used in this game, you can use the writeAccess property of the server-side Client object to stop clients from directly updating shared objects or calling the send method on them. This is an important feature to use when designing a game that discourages hacking attempts.

In the game described here, the app maintains state on the server and updates the game views through a shared object. Making it impossible for clients to directly update the shared object guarantees that they reflect the state as maintained by the server. You can deny write access in the application.onConnect method. For example, to restrict write access to a non-existent directory area named null:

```javascript
application.onConnect = function(newClient) {
    newClient.writeAccess = "/null";
}
```

Or, to completely disable write access, you can use reserved URI characters to create an illegal address:

```javascript
application.onConnect = function(newClient) {
    newClient.writeAccess = "/../../";
}
```

In our design, the movie makes a remote method call to request an action on the server, for example, when the user clicks on a square. Here is how we did this in an earlier version of the game:

```javascript
game_nc.call("openSquare", null, (this._name).substring(3), player);
```

In this statement the movie passes the name of the square to open and name of the player making the request as parameters. But what if someone wanted to cheat by sending a message to the server that an opponent had clicked on a square with a bomb? As the game progresses it is very easy to find squares that have bombs in them. This is why we rewrote the statement:

```javascript
game_nc.call("openSquare", null, (this._name).substring(3));
```
In this case, we don’t send the player’s name at all. Instead, the server knows which client requests the action and uncovers the square in his or her name:

```javascript
application.onConnect(newClient, playerName) {
    newClient.playerName = playerName;
    newClient.openSquare = function(square) {
        gameController.requestToOpenSquare(square, this.playerName);
    }
}
```

In this snippet of code the player’s name is stored in the client object and reused when he/she sends a request through the openSquare method.

**Dealing with Latency**

Using remote method calls to determine latency has some drawbacks. If other remote calls are queued on the server the round trip time will be inflated by the time it takes to handle those other calls. Also, continuing with the theme of not trusting the client whenever possible, relying on the client to participate in determining network latency is also dangerous. At first we thought this was a necessary drawback, as there seemed to be no way to avoid putting some code on the client in order to measure the round trip time of a remote method call. However, it turns out there is an undocumented feature used in the FCCommunicationLight component that helps us in this situation. The two undocumented server-side methods are:

- Client.ping - which pings the movie represented by the client object
- Client.getStats - which reports a number of statistics including the round trip time of the last Client.ping

For example to ping a client it is only necessary to call a Client instance’s ping method on the server:

```javascript
myClient.ping();
```

Later, when you want to know the round trip time, call getStats and examine the object it returns:

```javascript
var stats = myClient.getStats();
for (var p in stats){
    trace(p + " : " + stats[p]);
}
```

Here is the trace output:

```
bytes_in: 63
bytes_out: 3376
msg_in: 3
msg_out: 3
msg_dropped: 0
ping_rtt: 134
```

In this case it took 134 milliseconds for a test message to get to the client and return to the server. Dividing this in half, we get a rough estimate of the time it takes for a message to get from the client to the server of 67 milliseconds. Unfortunately, network latency is constantly changing. To monitor it, as the FCConnectionLight component does, you would have to use setInterval to regularly ping the client. Pinging the client regularly will add extra network traffic and incur a performance penalty. If you ping your servers regularly, consider doing this at half second or longer intervals. Network latency can change quickly and dramatically. Ping provides only an estimate of latency.

Earlier the problem of determining which user clicked on a square first was mentioned. The idea was to make the game fair by compensating for network lag. However, there is another problem we did not try to address. Clients receive game updates at different speeds. Consequently, some players will have an unfair advantage in that they will see updates before other players. Managing this problem is quite difficult. One approach is to simply prevent people with slow connections from playing. Another approach sends updates to each client individually based on their network latency so that they all arrive at roughly the same time. Playing a game with this feature will slow the game down so that it is equally slow for everyone. To do this you would have to use Client.call instead of a shared object send method.

**Design Enhancements**

In the design outlined, the PlayerList remote shared object is used for two purposes. One is to hold player information and
the other is to send game board updates to each. This is not a good design choice. It is much cleaner to use a separate shared object for every MVC triad that needs one. Instead of using the player list to send board updates, we should have used a separate MinesweeperGame shared object. We also could have used a MinesweeperGame shared object to hold information about the current visible state of the game. This would have allowed users to leave and return later to the game in progress.

Next, we realized too much movie clip code exists on the main timeline of the FLA and in the library. In other work, we have found it a much better practice to place all our code in external .as files with the minimal exception of short state transition code on the main timeline. We do this regularly now even when we are "just experimenting."

Lastly, the game model directly updates the player information in the player list. This sort of model-to-model cross-talk seems harmless in a simple test application. However, you should avoid using it in larger applications. As one alternative, the player list or player controller would be a view of the game model and would receive updates through the normal model/viewer update mechanism. In the work we are doing now, we use the Macromedia communication components and are investigating different mechanisms for components to communicate with each other. If we were to redo this game, we would redesign it as a component that cooperates with modified versions of other Macromedia communication components, such as the people list. Creating one-off communication applications like this game are a great way to experiment with the server and learn how it works. However, to build useful applications with a reasonable amount of code reuse, a component framework is a necessity.

Designing and building real-time communications applications can be challenging. We were pleasantly surprised by the rich variety of tools that Macromedia Flash Communication Server MX makes available, and the ease with which we used them. If you are planning on developing communication applications, we highly recommend you take the time to write test scripts and experiment with the base communication objects before seriously modifying the Macromedia communication components or creating your own. We also have one last warning: once you understand more, you may want to make everything into a communication application!