The cell thread module class shown below implements the Game-of-Life *cell* including the four *transition* rules.

```
class cell: public threadModule
{
    inputStream<bool>  inStrm[nNEIGHBORS];  // Input streams
    outputStream<bool> outStrm;             // Output stream
    bool               cellState;           // cell state

    void code()                             // Thread-domain code
    {
        int  g, d, liveCount;
        bool neighborState;

        for (g = 0; g < nGENERATIONS; ++g)  // Do generations
        {
            outStrm << cellState;      // Put cellState into outStrm
            liveCount = 0;                  // Reset liveCount

            for (d = 0; d < nNEIGHBORS; ++d)// For each direction
            {
                inStrm[d] >> neighborState; // Get neighbor state
                liveCount += neighborState; // Increment if
            }                               // neighbor is alive

            cellState = (cellState == 1) ?
                       ((liveCount == 2) || (liveCount == 3)) :
                        (liveCount == 3);   // Update cellState
        }
    }

public:

    cell()                                  // Constructor
    {
        int d;

        for (d = 0; d < nNEIGHBORS; ++d)    // Set inStrm
        {                                   // directions
            inStrm[d].setDirection( (TsDirection)d );
        }

        outStrm.setVisibility( tsVISIBLE ); // Make outStrm visible
    }                                       // outside array

    void setState( bool b )                 // Set cell state
    {
        cellState = b;
    }
};
```

Some observations about the cell thread module code are as follows:

a. The eight cell data members in the array

```
inputStream<bool>  inStrm[nNEIGHBORS];
```

are the *input streams* through which a cell receives cell states, of type bool, from its eight neighbors.

b. The cell data member

```
outputStream<bool> outStrm;
```

is the single *output streams* through which a cell sends its state, of type bool, to its eight neighbors and to the display module (see below).

c. The cell member function

```
void code()
```

contains the *thread-domain code* associated with the cell. It is ordinary C code with the exception of a single TruStream *put* to outStrm:

```
outStrm << cellState;
```

and a TruStream *get* from each inStrm:

```
inStrm[d] >> neighborState;
```

d. The statement

```
outStrm << cellState;
```

puts the cell's current state into outStrm. That data value is then broadcast to the cell's eight neighbors and to the display module.

e. The inner loop

```
for (d = 0; d < nNEIGHBORS; ++d)
```

cycles through the first 8 directions of the enumeration type:

```
typedef enum { tsNORTH         = 0,
               tsNORTHEAST     = 1,
               tsEAST          = 2,
               tsSOUTHEAST     = 3,
               tsSOUTH         = 4,
               tsSOUTHWEST     = 5,
               tsWEST          = 6,
               tsNORTHWEST     = 7,
               tsALLDIRECTIONS = 8 } TsDirection;
```

f. The statement

```
inStrm[d] >> neighborState;
```

gets the state of the neighbor connected to inStrm[d] *(see below).*

g. The statement

```
cellState = (cellState == 1) ?
         ((liveCount == 2) || (liveCount == 3)) :
          (liveCount == 3);
```

implements the four Game-of-Life rules:

1. *Any live cell with fewer than two live neighbours dies, as if caused by underpopulation.*
2. *Any live cell with two or three live neighbours lives on to the next generation.*
3. *Any live cell with more than three live neighbours dies, as if by overpopulation.*
4. *Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.*

And that's it for cell*'s thread-domain code.*

We now come to the two *public* member functions of cell:

```
cell()
    void setState( bool b )
```

cell() is the *default constructor* for the cell class. Within the body of cell():

The loop

```
for (d = 0; d < nNEIGHBORS; ++d)
```

cycles through the first 8 directions of the enumeration type TsDirection*.*

The statement

```
inStrm[d].setDirection( (TsDirection)d );
```

calls the inputStream *member function*

```
void setDirection( TsDirection );
```

to set the direction attribute of inStrm[d]*. The default value for this attribute for both input and output streams is* tsALLDIRECTIONS*.*

The statement

```
outStrm.setVisibility( tsVISIBLE );
```

calls the outputStream *member function*

```
void setVisibility( TsVisibility );
```

where TsVisibility *is the enumeration type*

```
typedef enum { tsNOTVISIBLE          = 0,
               tsUNCONNECTEDVISIBLE = 1,
               tsOUTWARDVISIBLE     = 2,
               tsVISIBLE            = 3 } TsVisibility;
```

to set the visibility attribute of outStrm*. The attribute is used by the* interconnect *member function of the* streamModule *class, and by the member operator* >> *of the* module *class (see below). Definitions:*

- tsNONEVISIBLE *No instances of the output stream are visible outside the enclosing stream module.*

- *tsUNCONNECTEDVISIBLE Only dangling instances of the output stream (instances not connected to a stream) are visible outside the enclosing stream module.*
- *tsOUTWARDVISIBLE Only outward facing instances of the output stream in a module array are visible outside the enclosing stream module. (An output-stream instance is outward facing if it is on the periphery of the array facing outward.)*
- *tsALLVISIBLE All instances of the output stream are visible outside the enclosing stream module.*

The default value for the visibility attribute of both input and output streams is tsUNCONNECTEDVISIBLE, which is fine for the input streams of cells in the Game-of-Life array. That's because they are all connected, and because there is no need to make them visible outside the array. The cell output stream, however, is a different story since we need to connect the output streams of the GOL cells to the display module (see below). We therefore set the visibility attribute of the cell output stream to tsVISIBLE.

Last on our list is the cell member function

```
void setState( bool b )
```

It is used by the GameOfLifeArray stream module to initialize the states of cells in the Game-of-Life array.