COMP229 Object-Oriented Programming Practices

Assignment 1 - A Game of Nim

Simon Dawson
(40983129)
dawson.sa@gmail.com

17 September 2009

Defining the problem:

What are the objects to use to create this software solution? To aid in the process of encapsulation of real world objects and concepts it would be useful to look at what is currently being done in the real world that we eventually hope to model with a computer system. Should such a thing be possible?

The objective

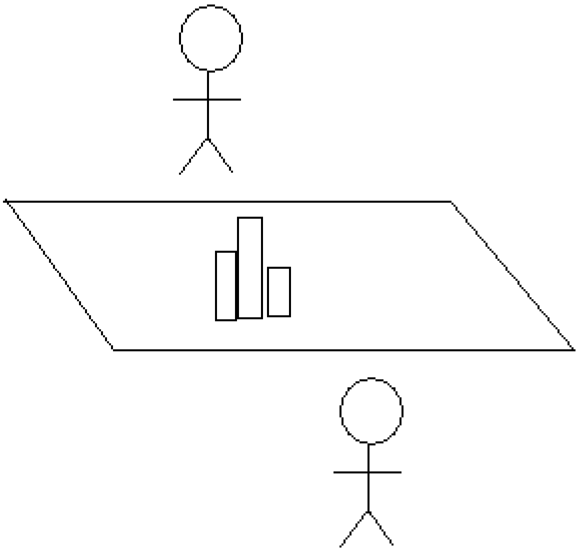
To create a computer version of the game of nim, the game where a set range of quantities of objects are removed from different piles of objects in turn by players until none are left.

In our implementation of the game one player will be black one will be white.

The following requirements are to be met by the application

1. White player always plays first.
2. There will be 3 piles of 9 objects
3. A player is only permitted to remove objects from a single pile per turn
4. When removing objects the set {1,2,3} defines what quantise of objects can be removed
5. Play alternates until there are no more matches on the table.
6. The last player to remove matches wins.

A game in action



Two **players** are sat across from each other.

The **game** a series of **turns**.

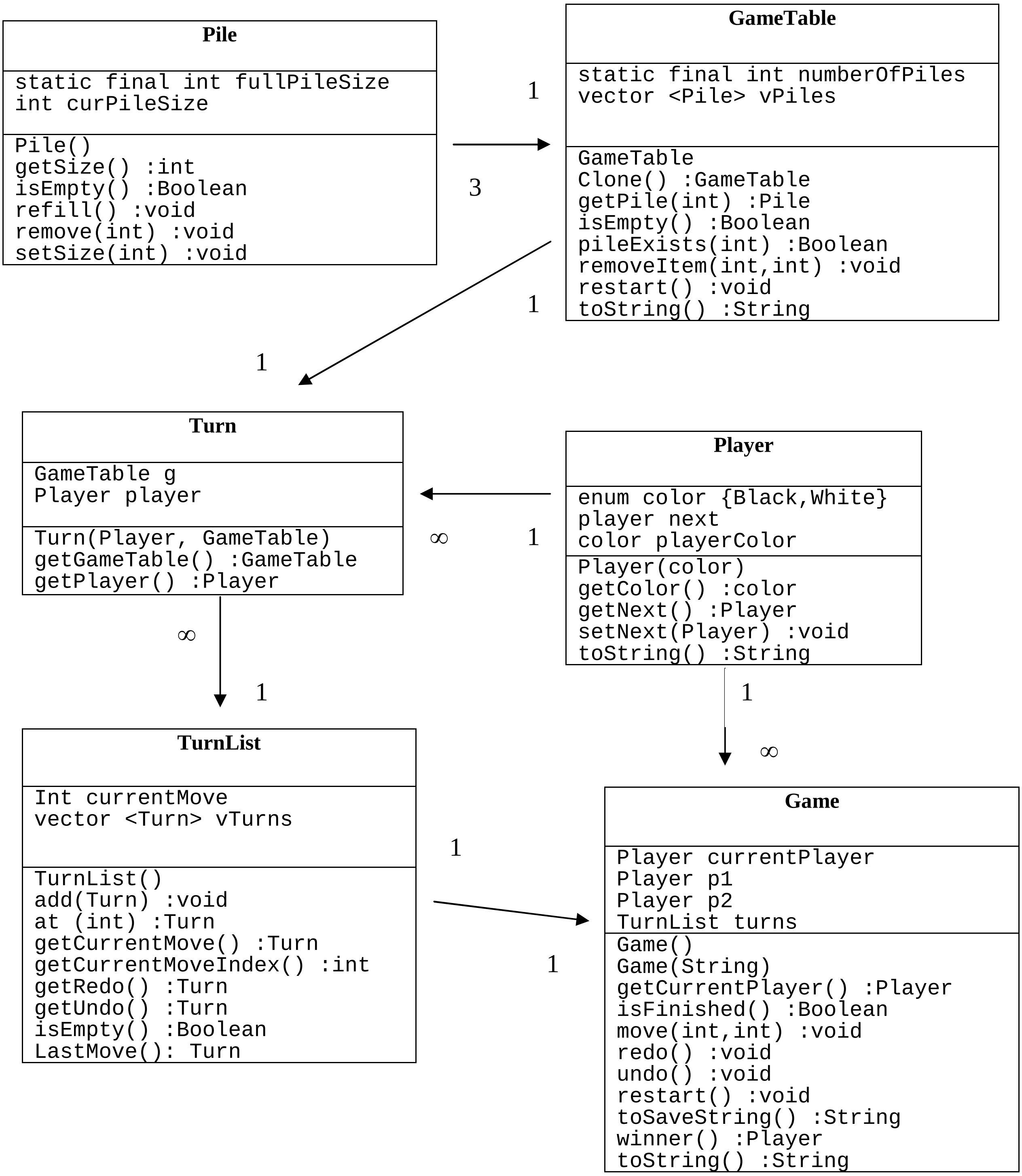
Each **turn** the **player** chooses to remove some items from one of the **piles.**

The **game** is over when the **table** is empty.

To begin the process of abstracting code from the requirements it was necessary to consider what some of the simpler components of the game were. From the above diagram some of the most basic classes can begin to take form.

What progressed from here was an evolution of classes whose functionality continually expanded as other classes in the application demanded more or different information as well as the application itself becoming more complex.

Designing the Solution:



Pile

```
static final int fullPileSize
int curPileSize

Pile()
getSize() :int
isEmpty() :Boolean
refill() :void
remove(int) :void
setSize(int) :void
```

GameTable

```
static final int numberOfPiles
vector <Pile> vPiles

GameTable
Clone() :GameTable
getPile(int) :Pile
isEmpty() :Boolean
pileExists(int) :Boolean
removeItem(int,int) :void
restart() :void
toString() :String
```

Turn

```
GameTable g
Player player

Turn(Player, GameTable)
getGameTable() :GameTable
getPlayer() :Player
```

Player

```
enum color {Black,White}
player next
color playerColor

Player(color)
getColor() :color
getNext() :Player
setNext(Player) :void
toString() :String
```

TurnList

```
Int currentMove
vector <Turn> vTurns

TurnList()
add(Turn) :void
at (int) :Turn
getCurrentMove() :Turn
getCurrentMoveIndex() :int
getRedo() :Turn
getUndo() :Turn
isEmpty() :Boolean
LastMove(): Turn
```

Game

```
Player currentPlayer
Player p1
Player p2
TurnList turns

Game()
Game(String)
getCurrentPlayer() :Player
isFinished() :Boolean
move(int,int) :void
redo() :void
undo() :void
restart() :void
toSaveString() :String
winner() :Player
toString() :String
```

Basic Class Diagram

Object Summaries

Pile

Field Summary

```
public static final int fullPileSize = 9;
int curPileSize;
```

Constructor summary

```
public Pile ()
```

Method Summary

```
int getSize()
      returns the size of pile
void setSize(int _i)
      assigns the magnitude of curPileSize to _i
void remove (int _howMany)
      Decreases the size of the pile by howMany if there are
      enough left
void refill()
      Resets the value of this pile size to full
boolean isEmpty()
      returns true if curPileSize>=0 otherwise false
```

GameTable

Field Summary

```
public static final int numberOfPiles=3;
private Vector <Pile> vPiles = new Vector<Pile>();
```

Constructor Summary

```
public GameTable ()
// Creates the piles. The number of piles to creates is defined
as an attribute
```

Method Summary

```
public Pile getPile(int _n) throws NimError
// Return the n'th pile in the vector
public void removeItem(int _fromPile, int _howMany)
// Removes _howMany, from the _fromPile'th piles size attribute
public void restart()
// Restarts the GameTable object restoring each piles
public void showPiles ()
// Debugging method prints the size of the piles in a string
XXX with the n'th character is the size the n'th pile
public String toString()
// Prints the piles out.
// ex.
/*

                *
              * *
              * *
              * *
            * * *
            * * *
            * * *


*/
public boolean isEmpty()
// Returns true iff the size of all piles are empty
public boolean pileExists(int _i)
// Returns true iff the Pile is between 0 and the defination of
the numberOfPiles-1
```

```
public GameTable clone()
// returns a hard copy of this current object
```

Player

Field Summary

```
public enum colour{Black,White};
private colour playerColour;
private Player next;
```

Constructor Summary

```
public Player (colour _colour)
```

Method Summary

```
public Player getNext()
// returns which player is next
public void setNext(Player _next)
// sets the player whos turn it is after this one
public colour getColor ()
// Returns the color of this player
public String toString()
// returns the string describing the player of the player
```

Turn

Field Summary

```
private Player player;
private GameTable g;
```

Constructor Summary

```
public Turn (Player _player,GameTable _g)
```

Method Summary

```
public Player getPlayer()
// Returns the player whos turn is stored
public GameTable getGameTable()
// Returns the GameTable which resulted in this turn
public String toString()
// Returns a textual description of the turn
```

TurnList

Field Summary

```
private Vector <Turn> v = new Vector<Turn>();
private int currentMove;
```

Constructor Summary

```
public TurnList ()
```

Method Summary

```
public int getCurrentMoveIndex ()
// returns the index of the move in the list that is currently
being looked at
public void add (Turn _t)
// addds a Turn to the turn list. if the currentMove is not the
end of the list it adds it at the end and deletes all turns
after it in the list
public Turn at (int i)
//return the i'th move back
public Turn getCurrentMove ()
// returns the current turn
public Turn lastMove()
// returns the last move made
public boolean isEmpty ()
// returns true iff there have been no moves in the game
public String toString()
// returns a list of off the turns that have been made in the
game
```

```
      public Turn getUndo() throws NimError
      // returns the turn one back from the current move iff it
      exists
      public Turn getRedo() throws NimError
      // returns the turn on forwards from the current move iff it
      exists
```

Game

Field Summary

```
      private Player p1;
      private Player p2;
      private Player currentPlayer;
      private GameTable g;
      private TurnList turns;
```

Constructor Summary

```
      public Game ()
      // Default constructor
      //  Pre:
      //          None
      //  Post:
      //    - Creates a Game object where Player White
      //    - The currentPlayer is assigned to white
      //          - Loads a GameTable object
      //          - Stores the first turn in the list of turns
      public Game (String _saveString)
      // Constructor (String _saveString)
      //    Pre:
      //          - String is in correct format
      //  Post:
      //          - Creates players
      //    - The currentPlayer is assigned to white iff
            _saveString[0]is a 0 or black
      //          - Loads a GameTable object
      //          - Assigns the size of the n'th Pile on the Table to
                  the size represented
      //                  by the number from _saveString[n]
      //          - Stores the first turn in the list of turns
```

Method Summary

```
      public String toSaveString () throws NimError
      //    Quickly summerises the gamestate in a small transportable
string
      //    Pre:
      //          - None
      //  Post:
      //          - return value[0] is 0 iff currentPlayer is
assigned to white else 1
      //          - return value[n] is assigned to the character
equivalent for the n'th pile size
      public Player getCurrentPlayer ()
      //    Returns the who's turn it currently is
      public void move (int _moveFromPile, int _howManyFromPile)
      throws NimError
      //  The _moveFromPile'th pile attribute size is reduced by
_howManyFromPile
      //  Checks to see if the pile exists and whether the number of
things you
      //    choose to remove is a legal move
      public boolean isFinished()
      // Returns true iff gameTable is empty
      public void switchPlayer()
```

```
// Assigns currentPlayer to the next Player
public void restart()
// Restarts the game
// Gets the GameTable to restart
// Clears the turns list and adds the first move
public Player winner()
// Returns null iff the game is not over
// else returns the last person to make a move
public void undo () throws NimError
// Sets the state of the game to that of the previous turn in
the list
public void redo() throws NimError
// Sets the state of the game to that of the next turn in the
list
public String toString()
// Gives a string representation of the GameTable
```

Testing the solution:

Tests were undertaken for 3 key aspects of the games functionality. Implrementations of JUnit tests were implemented for the following classes Pile, GameTable and the Game.

Pile:

✓ Pile created with the correct size

```
public void testPileCreationSize () {
      Pile p1= new Pile();
      Assert.assertTrue(p1.getSize()==9);}
```

✓ Pile created was not empty. When 9 were removed it was empty

```
public void testPileCreationEmpty () {
      Pile p1= new Pile();
      Assert.assertFalse(p1.isEmpty());
      p1.remove(9);
      Assert.assertTrue(p1.isEmpty());}
```

✓ Removing 1 from the pill leaves size 8, removing 4 from the pile after leaves 4

```
public void testRemove () throws NimError {
      Pile p1= new Pile();
      p1.remove(1);
      Assert.assertTrue(p1.getSize()==8);
      p1.remove(4);
      Assert.assertTrue(p1.getSize()==4);}
```

GameTable:

✓ GameTable created correct number of Piles

```
public void testGameTableCreationSize () {
      GameTable gt = new GameTable();
      for (int i =0; i < GameTable.numberOfPiles;i++)
      Assert.assertTrue(gt.pileExists(i));}
```

✓ While removing items from piles isEmpty returns false. As soon as all 27 items are removed isEmpty returns true

```
public void testEmpty () throws NimError {
      GameTable gt = new GameTable();
      for (int x=0;x<3;x++){
            for (int i=0;i<3;i++){
                  Assert.assertFalse(gt.isEmpty());
```

```
                        gt.removeItem(x, 3);}//end for i}//end for x
            Assert.assertTrue(gt.isEmpty());}
```

✓ All items removed correctly till none were left, then all items restored correctly with the restart()

```
      public void testRestart () throws NimError {
            GameTable gt = new GameTable();
            //empty table
            for (int x=0;x<3;x++){
                  for (int i=0;i<3;i++){
                        Assert.assertFalse(gt.isEmpty());
                        gt.removeItem(x, 3);}//end for i}//end for x
            Assert.assertTrue(gt.isEmpty());
            gt.restart();
            for (int i =0; i < GameTable.numberOfPiles;i++) {
      Assert.assertTrue(gt.getPile(i).getSize()==Pile.fullPileSize);}
```

Game:

✓ Game was created it was not finished, there was no winner yet. White is first player to move

```
      public void testGameCreation () {
            Game g = new Game();
            Assert.assertFalse(g.isFinished());
            Assert.assertTrue(g.winner()==null);
      Assert.assertTrue(g.getCurrentPlayer().toString()=="White");}
```

✓ A move of 4 created exception
 Move of -1 created exception
 Move from pile 5 created exception
 Move from pile -1 created exception

```
      public void testMove() throws NimError {
            Game g = new Game();
try { g.move(1, 4); } catch (NimError e) {Assert.assertTrue(true);}
try { g.move(1, -1); } catch (NimError e) {Assert.assertTrue(true);}
try { g.move(-1, 1); } catch (NimError e) {Assert.assertTrue(true);}
try { g.move(5, 1); } catch (NimError e) {Assert.assertTrue(true);}}
```

✓ Game started 27 items remove game was declared finished and winner was Black correctly

```
      public void testFinish() throws NimError {
            Game g = new Game();
            for (int x=0;x<3;x++){
                  for (int i=0;i<3;i++){
                        Assert.assertFalse(g.isFinished());
                        g.move(x, 3);}//end for I }//end for x
            Assert.assertTrue(g.isFinished());
            Assert.assertTrue(g.winner().toString()=="Black");}
```

✓ Game created an undo or redo caused an exception correctly when no moves have been made. A move was made correctly. That move was undone correctly and lastly was redone.

```
      public void testUndoRedo () throws NimError {
            Game g = new Game();
            String state1=g.toSaveString(); //0999
      try {g.undo();} catch (NimError e) { Assert.assertTrue(true);}
      try {g.redo();} catch (NimError e) { Assert.assertTrue(true);}
```

```
            g.move(1,1);
            String state2=g.toSaveString(); //1989
            g.undo();
            String state3=g.toSaveString(); //0999
            g.redo();
            String state4=g.toSaveString(); //1989
            Assert.assertFalse(state1.equals(state2));
            Assert.assertTrue(state1.equals(state3));
            Assert.assertFalse(state1.equals(state4));
            Assert.assertFalse(state2.equals(state3));
            Assert.assertTrue(state2.equals(state4));
            Assert.assertFalse(state3.equals(state4));}
```

Conclusion

Throughout the design the need to ensure object simplicity was keep paramount. Classes were written to behave as close to their real world counterparts. The classes relating the logic and function of the game mechanics were kept from the user via visage classes that dealt with the interactions between objects and users. What the product of all of these became was an object-oriented application that met the requirements.