OXO Debrief

COMSM0086

Dr Simon Lock & Dr Sion Hannuna

Aim of this session

Our aim is to provide insight into the OXO exercise
By working through JUST ONE possible solution
(Note: other approaches might be equality valid)

It is problematic to produce an "ideal" solution
(Probably couldn't agree on the best approach ;o)

Aim to provide something simple & understandable
(Which will permit the discussion of various issues)

A sensible and achievable implementation of OXO
(rather than trying to show off how "clever" we are)

Driving Principles

Divide and Conquer: delegate clearly defined tasks
Granularity: short methods to aid understandability
Simplicity: avoid unnecessarily complex control flow
Modesty: avoid sophisticated language constructs
Understandability: give things clear sensible names

Overall aim:

Produces a solution everyone can understand !
Short and simple: My OXOController is 190 lines

Recap of Features

Build OXO / Tic-Tac-Toe / noughts & crosses with:

- Command Parsing & Validation
- Cell Claiming
- Dynamic Board Size
- Dynamic Win Threshold
- Automated Win and Draw Detection
- Extendable Number of Players
- Game Reset

Identified Classes

Main Class: OXOGame

MVC: OXOModel, OXOView, OXOController

Exceptions: OXOMoveException

Player class: OXOPlayer

Test class: OXOTests

Didn't really feel the need for any additional classes

32		48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Command Parsing and Validation

```
command = command.toLowerCase();
if(command.length() != 2) throw new InvalidIdentifierLengthException(command.length());
int rowNumber = convertToRowNumber(command.charAt(0));
int colNumber = convertToColNumber(command.charAt(1));
int currentPlayerNumber = gameModel.getCurrentPlayerNumber();
claimBoardCell(gameModel.getPlayerByNumber(currentPlayerNumber), rowNumber, colNumber);

private static int convertToRowNumber(char rowChar) throws InvalidIdentifierCharacterException {
    if((rowChar<'a') || (rowChar>'z')) throw new InvalidIdentifierCharacterException(RowOrColumn.R
    else return rowChar-'a';
}

private static int convertToColNumber(char colChar) throws InvalidIdentifierCharacterException, Ou
{
    if (colChar=='0') throw new OutsideCellRangeException(RowOrColumn.COLUMN, 0);
    else if((colChar<'1') || (colChar>'9')) throw new InvalidIdentifierCharacterException(RowOrCol
    else return colChar-'1';
}
```

Claiming Cells

Again, let's try to keep everything clean and simple

A compact multi-part IF statement to check validity

Appropriate exception thrown if command is invalid

The final branch claims the cell for the player

```
public void claimBoardCell(OXOPlayer player, int rowNum, int colNum) throws OXOMoveException
{
    if(rowNum>=gameModel.getNumberOfRows()) throw new OutsideCellRangeException(RowOrColumn.ROW
    else if(colNum>=gameModel.getNumberOfColumns()) throw new OutsideCellRangeException(RowOrCo
    else if(gameModel.getCellOwner(rowNum, colNum) != null) throw new CellAlreadyTakenException
    else gameModel.setCellOwner(rowNum, colNum, player);
}
```

Dynamic Board Size

Replace 2D array: ArrayList of ArrayLists of Players

Create top level list ("cells") to hold all the rows

For each row in board, create a new list of players

Initialise all elements in each row to null (empty)

```
private ArrayList<ArrayList<OXOPlayer>> cells;
public OXOModel(int numberOfRows, int numberOfColumns, int winThresh) {
    winThreshold = winThresh;
    cells = new ArrayList<ArrayList<OXOPlayer>>();
    for(int i=0; i<numberOfRows ;i++) {
        ArrayList<OXOPlayer> newRow = new ArrayList<OXOPlayer>(numberOfColumns);
        for(int j=0; j<numberOfColumns ;j++) newRow.add(null);
        cells.add(newRow);
    }
}
```

Unlimited Number of Players

Make use of an extendable ArrayList to store players

When advancing to next player, check for wrap-around

```
private ArrayList<OXOPlayer> players;

public int getNumberOfPlayers() {
    return players.size();
}

public void addPlayer(OXOPlayer player) {
    players.add(player);
}

if(currentPlayerNumber < gameModel.getNumberOfPlayers()-1) currentPlayerNumber++;
else currentPlayerNumber = 0;
gameModel.setCurrentPlayerNumber(currentPlayerNumber);
```

Dynamic Win Threshold

Win threshold maintained *inside* OXOModel object

No COPIES of this value kept anywhere in the code

All references to threshold call the accessor method

When central value changes, all references will see

```
private int winThreshold;

public void setWinThreshold(int winThresh) {
    winThreshold = winThresh;
}

public int getWinThreshold() {
    return winThreshold;
}
```

Game Reset

Reinitialise ArrayList of ArrayLists of OXOPlayers

Reset player number, winner, game drawn

Don't bother resetting size of board or win threshold

```
public void reset() {
    cells = new ArrayList<ArrayList<OXOPlayer>>();
    for(int i=0; i<numberOfRows ;i++) {
        ArrayList<OXOPlayer> newRow = new ArrayList<OXOPlayer>(numberOfColumns);
        for(int j=0; j<numberOfColumns ;j++) newRow.add(null);
        cells.add(newRow);
    }
    currentPlayerNumber = 0;
    winner = null;
    gameDrawn = false;
}
```

Win Detection

Basic principle: Scan through the board, cell by cell
Keeping a count of consecutively claimed cells

Chose not to be clever (1 method for all directions !)
Used separate (simpler) methods for each direction
Would probably be flagged for not using DRY code !

Horizontal and Vertical lines are relatively easy
Diagonal lines are a lot tricker to check

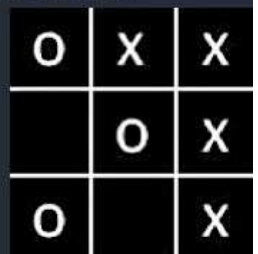
Let's take a look at how I did Vertical...

Example: Vertical Win Detection

Scan down through each column, checking each cell

Compare owner of cell against owner of previous cell

```
public OXOPlayer searchForVerticalWin()
{
    for(int colNumber=0; colNumber<gameModel.getNumberOfColumns() ;colNumber++) {
        int consecutiveCount = 0;
        OXOPlayer ownerOfPreviousCell = null;
        for(int rowNumber=0; rowNumber<gameModel.getNumberOfRows() ;rowNumber++) {
            OXOPlayer ownerOfCurrentCell = gameModel.getCellOwner(rowNumber,colNumber);
            if(ownerOfCurrentCell == null) consecutiveCount = 0;
            else if(ownersDiffer(ownerOfPreviousCell,ownerOfCurrentCell)) consecutiveCount = 1;
            else consecutiveCount++;
            if(consecutiveCount >= gameModel.getWinThreshold()) return ownerOfCurrentCell;
            ownerOfPreviousCell = ownerOfCurrentCell;
        }
    }
    return null;
}
```



Diagonal Win Detection

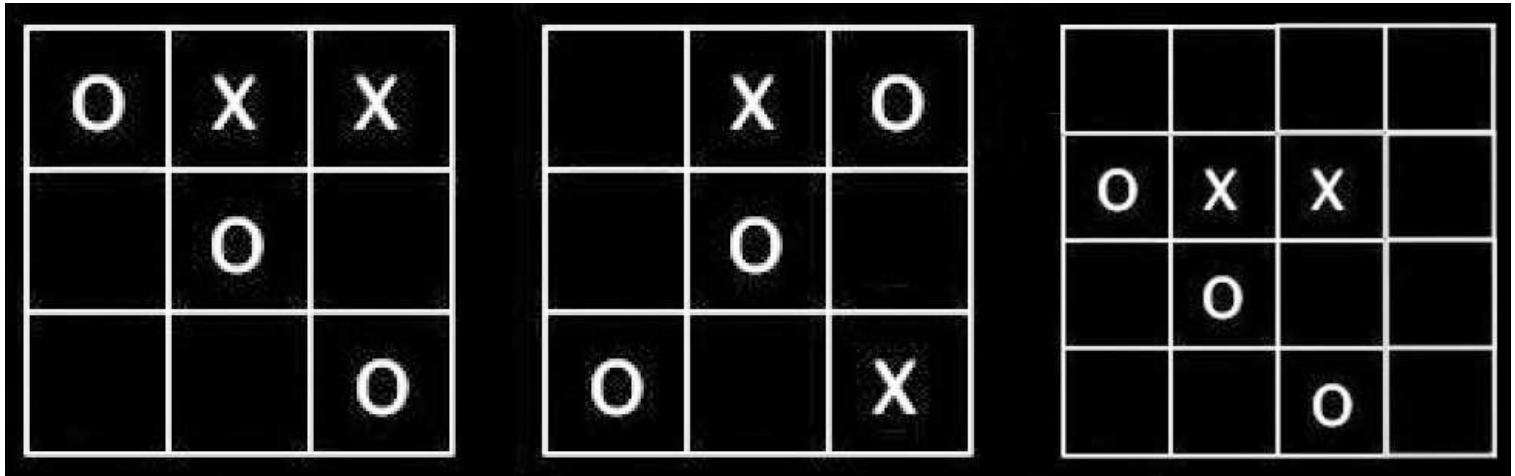
Diagonal Win Detection much harder to implement

Need to increment both x and y at the same time

Must also deal with two different directions of slope

Must work for ALL board sizes (not just 3x3)

Might not start in corners or go through centre cell



A Novel Approach

Rather than trying to solve this difficult problem...
What if we TRANSFORM it into a simpler problem ?
(a problem that we *already* have a solution for ;o)

SmartWinDetection

This kind of intelligent pre-processing is powerful
Think laterally - don't just brute-force a solution
Try to make your life easier (chopping onions !)

Row Shifting Code: Versatile DRY code

```
public void indentRows(Direction direction, Action action)
{
    int sizeOfIndent = 0;
    int rowNumber;
    if(direction == Direction.DOWN) rowNumber = 0;
    else rowNumber = this.getNumberOfRows()-1;
    while((rowNumber>=0) && (rowNumber<this.getNumberOfRows())) {
        for(int i=0; i<sizeOfIndent ;i++) {
            if(action == Action.ADD) this.getRow(rowNumber).add(0,null);
            else if(action == Action.REMOVE) this.getRow(rowNumber).remove(0);
        }
        sizeOfIndent++;
        if(direction == Direction.DOWN) rowNumber++;
        if(direction == Direction.UP) rowNumber--;
    }
}
```

Automated Draw Detection

Draw detection is relatively simple:

The game is drawn when ALL cells have been filled...

Provided that no player has won the game !

Common Problems and Issues...

Common Problems and Issues

- Broken encapsulation (direct access of internal state)
- State leakage (multiple inconsistent copies of data)
- Game continues (turns can be taken) even after win
- Unconstrained resize of board (add/remove row/col)
- Not all erroneous inputs detected and prevented
- Win not always detected (with expanded board size)
- Hardwired checking of playing letter (this isn't C !)
- Limited coverage of features by your own test cases

Questions ?