

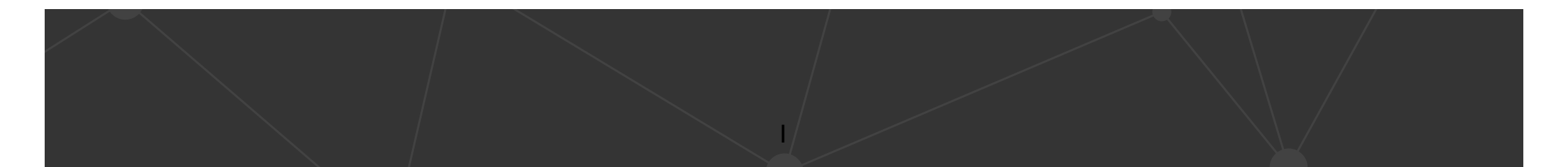


# Iterative Development

Alexandre Bergel

<http://bergel.eu>

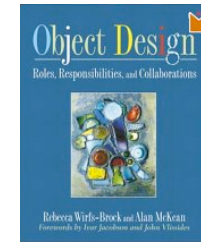
27-09-2021



# Source

---

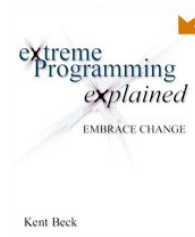
Rebecca Wirfs-Brock, Alan McKean,



Object Design — Roles, Responsibilities and Collaborations, Addison-Wesley, 2003.

Kent Beck,

Extreme Programming Explained — Embrace Change, Addison-Wesley, 1999.



# Roadmap

---

1 - The iterative software lifecycle

2 - Responsibility-driven design

3 - TicTacToe example

Identifying objects

Scenarios

Test-first development

Printing object state

Testing scenarios

# Roadmap

---

## **1 - The iterative software lifecycle**

2 - Responsibility-driven design

3 - TicTacToe example

Identifying objects

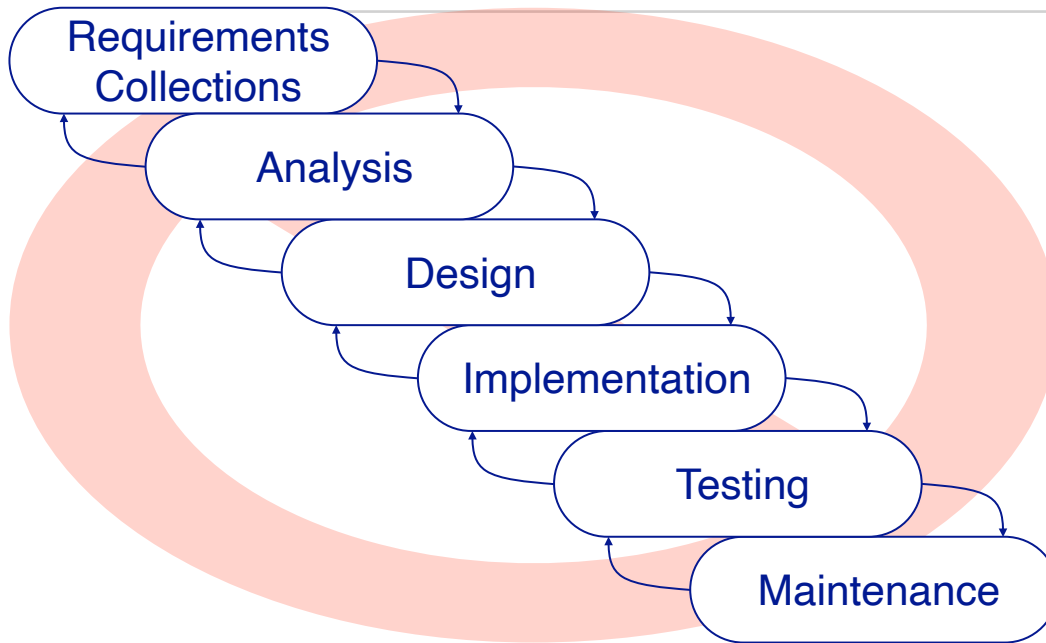
Scenarios

Test-first development

Printing object state

Testing scenarios

# The Classical Software Lifecycle



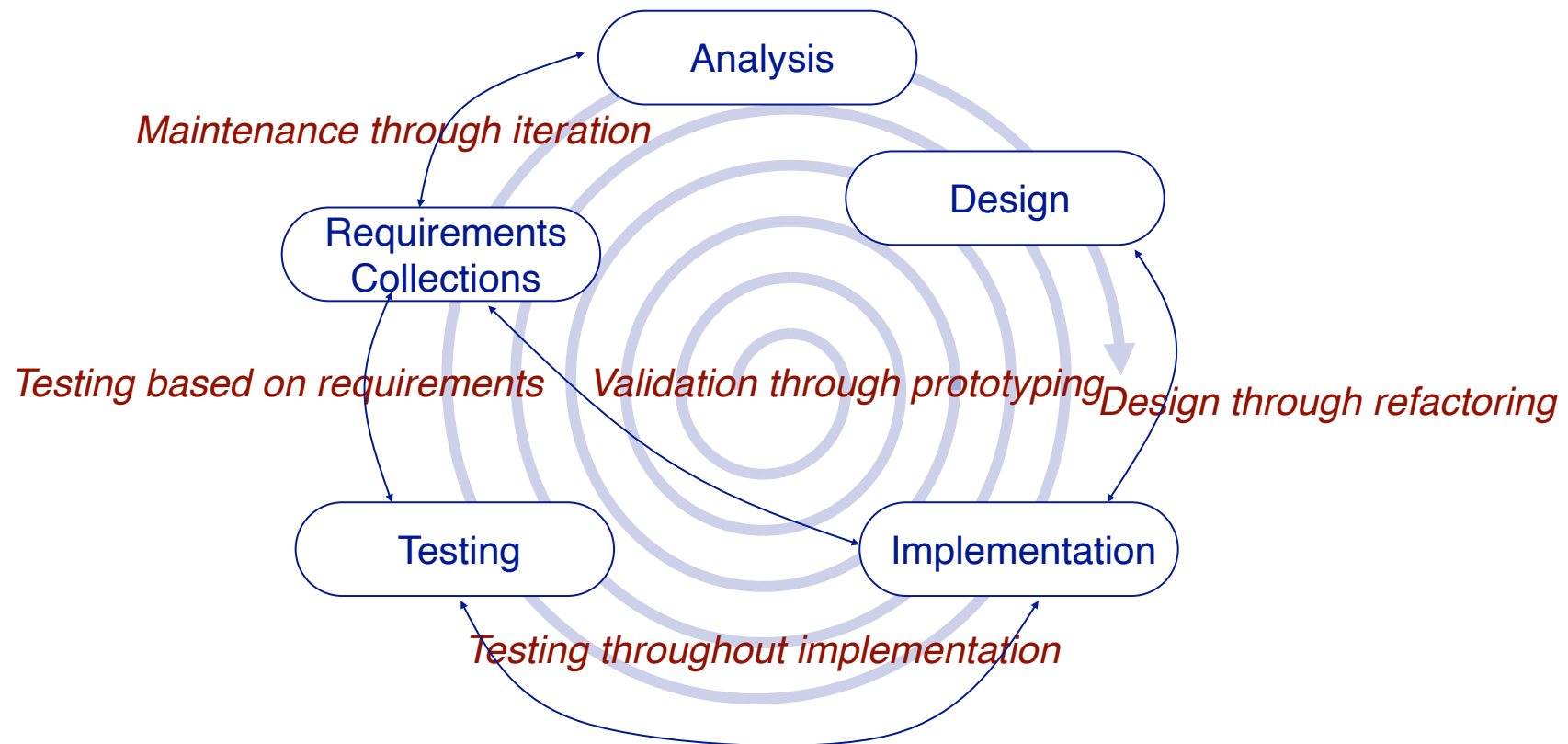
The classical software lifecycle models the software development as a step-by-step “waterfall” between the various development phases.

*The waterfall model is unrealistic for many reasons, especially:*

- requirements must be “frozen” too early in the life-cycle
- requirements are validated too late

# Iterative Development

In practice, development is always iterative, and all software phases progress in parallel.



If the waterfall model is pure fiction, why is it still used?

# Roadmap

---

1 - The iterative software lifecycle

**2 - Responsibility-driven design**

3 - TicTacToe example

Identifying objects

Scenarios

Test-first development

Printing object state

Testing scenarios

# What is Responsibility-Driven Design?

Responsibility-Driven Design is

a method for deriving a software design in terms of *collaborating* objects  
by asking what *responsibilities* must be fulfilled to meet the requirements,  
and assigning them to the appropriate *objects* (i.e., that can carry them  
out)



# How to assign responsibility?

---

Pelrine's Laws:

“Don't do anything you can push off to someone else.”

“Don't let anyone else play with you.”

RDD leads to fundamentally different designs than those obtained by functional decomposition or data-driven design

*Class responsibilities tend to be more stable over time than functionality or representation*

# Roadmap

---

1 - The iterative software lifecycle

2 - Responsibility-driven design

## **3 - TicTacToe example**

Identifying objects

Scenarios

Test-first development

Printing object state

Testing scenarios

# Example: “Tic Tac Toe”

---

## Requirements

“A simple game in which one player marks down only crosses and another only ciphers [zeroes], each alternating in filling in marks in any of the nine compartments of a figure formed by two vertical lines crossed by two horizontal lines, the winner being the first to fill in three of his marks in any row or diagonal.”

— Random House Dictionary

We should design a program that implements the rules of Tic Tac Toe

# Setting Scope...

---

## Questions:

Should we support other games?

Should there be a graphical UI?

Should games run on a network? Through a browser?

Can games be saved and restored?

A monolithic paper design is bound to be wrong!

# Setting Scope

---

An iterative development strategy:

limit initial scope to the *minimal requirements* that are interesting

*grow the system* by adding features and test cases

let the *design emerge by refactoring* roles and responsibilities

How much functionality should you deliver in the first version of a system?

Select the minimal requirements that provide value to the client

# Roadmap

---

- 1 - The iterative software lifecycle
- 2 - Responsibility-driven design
- 3 - TicTacToe example

## **Identifying objects**

Scenarios

Test-first development

Printing object state

Testing scenarios


# Tic Tac Toe Objects...

Some objects can be identified from the requirements:

<i><b>Objects</b></i>	<i><b>Responsibilities</b></i>
Game	Maintain game rules
Player	Make moves Mediate user interaction
Compartment	Record marks
Figure (State)	Maintain game state

*Entities with clear responsibilities are more likely to end up as objects in our design*

# Tic Tac Toe Objects

Others can be eliminated:	
Non-Objects	Justification
Crosses, ciphers	Same as Marks
Marks	Value of Compartment
Vertical lines	Display of State
Horizontal lines	ditto
Winner	State of Player
Row	View of State
Diagonal	ditto
<p> <u>How can you tell when you have the “right” set of objects?</u></p> <p>✓ <i>Each object has a clear and natural set of responsibilities.</i></p>	



# Missing Objects

---

Now we check if there are unassigned responsibilities:

Who starts the Game?

Who is responsible for displaying the Game state?

How do Players know when the Game is over?

Let us introduce a Driver that supervises the Game

How can you tell if there are objects missing in your design?

*When there are responsibilities left unassigned*

# Roadmap

---

- 1 - The iterative software lifecycle
- 2 - Responsibility-driven design
- 3 - TicTacToe example

Identifying objects

## **Scenarios**

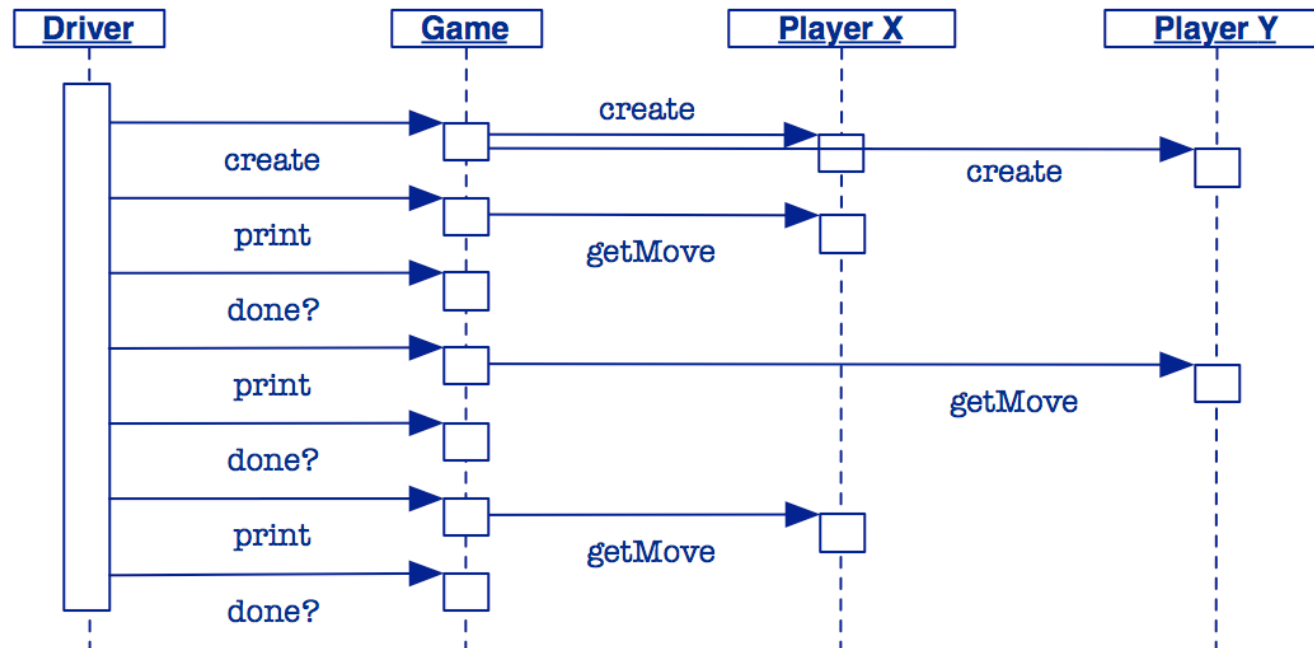
Test-first development

Printing object state

Testing scenarios

# Scenarios

A scenario describes a typical sequence of interactions:



Are there other equally valid scenarios for this problem?

# Version 0 — skeleton

*Our first version does very little!*

```
class GameDriver {  
    public static void main(String args[]) {  
        TicTacToe game = new TicTacToe();  
        do { System.out.print(game); }  
        while(game.notOver());  
    }  
    public class TicTacToe {  
        public boolean notOver() { return false; }  
        public String toString() { return "TicTacToe\n"; }  
    }  
}
```

How do you iteratively “grow” a program?  
Always have a running version of your program.

# Roadmap

---

1 - The iterative software lifecycle

2 - Responsibility-driven design

3 - TicTacToe example

Identifying objects

Scenarios

**Test-first development**

Printing object state

Testing scenarios

# Version 1 — game state

---

We will use chess notation to access the game state

Columns 'a' through 'c'

Rows '1' through '3'

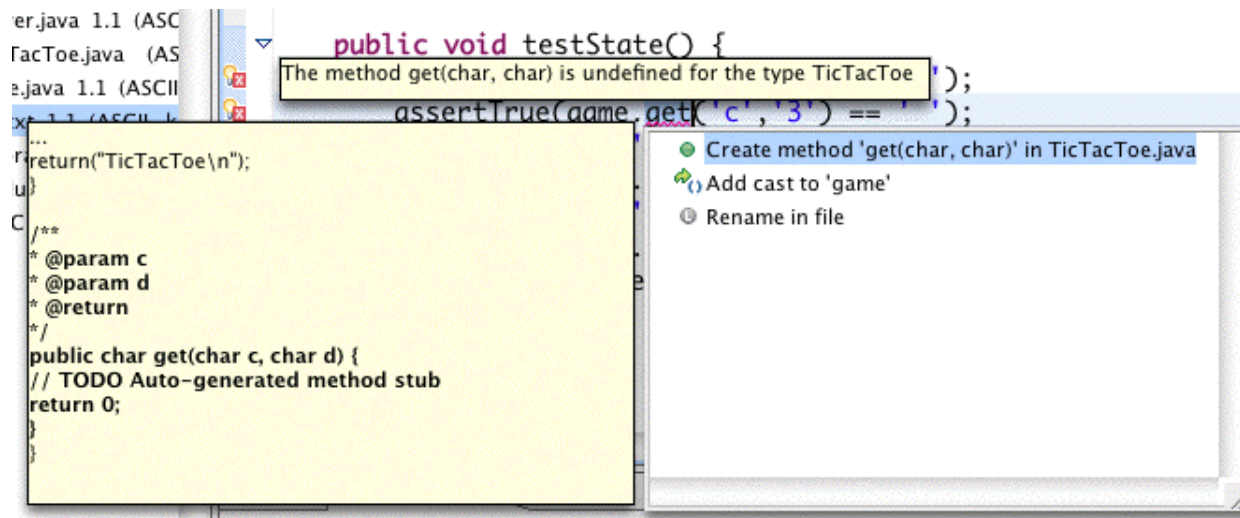
How do we decide on the right interface?

*First write some tests!*

# Test-first development

```
public class TicTacToeTest {  
    private TicTacToe game;  
  
    @Before public void setUp() {  
        game = new TicTacToe();  
    }  
  
    @Test public void testState() {  
        assertTrue(game.get('a','1') == ' ');  
        assertTrue(game.get('c','3') == ' ');  
        game.set('c','3','X');  
        assertTrue(game.get('c','3') == 'X');  
        game.set('c','3',' ');  
        assertTrue(game.get('c','3') == ' ');  
        assertFalse(game.inRange('d','4'));  
    }  
}
```

# Generating methods



Test-first programming can drive the development of the class interface ...



# Roadmap

---

1 - The iterative software lifecycle

2 - Responsibility-driven design

3 - TicTacToe example

Identifying objects

Scenarios

Test-first development

**Printing object state**

Testing scenarios

# Representing game state

```
public class TicTacToe {  
    private char[][] gameState;  
    public TicTacToe() {  
        gameState = new char[3][3];  
        for (char col='a'; col <='c'; col++)  
            for (char row='1'; row<='3'; row++)  
                this.set(col,row, ' ');  
    }  
    ...  
}
```

# Printing the state

By re-implementing `TicTacToe.toString()`,  
we can view the state of the game:

```
3      |  |  |  
  ---+---+---  
2      |  |  |  
  ---+---+---  
1      |  |  |  
      a  b  c  
Player X moves:
```

How do you make an object printable?  
Override `Object.toString()`

# TicTacToe.toString()

Use a `StringBuilder` (not a `String`)  
to build up the representation:

```
public String toString() {  
    StringBuilder rep = new StringBuilder();  
    for (char row='3'; row >= '1'; row--) {  
        rep.append(row);  
        rep.append("  ");  
        for (char col='a'; col <='c'; col++) { ... }  
        ...  
    }  
    rep.append("  a  b  c\n");  
    return(rep.toString());  
}
```

*Concatenating strings using + is very costly*

# Roadmap

---

1 - The iterative software lifecycle

2 - Responsibility-driven design

3 - TicTacToe example

Identifying objects

Scenarios

Test-first development

Printing object state

**Testing scenarios**

# Version 2 — adding game logic

---

We will

Add test scenarios

Add Player class

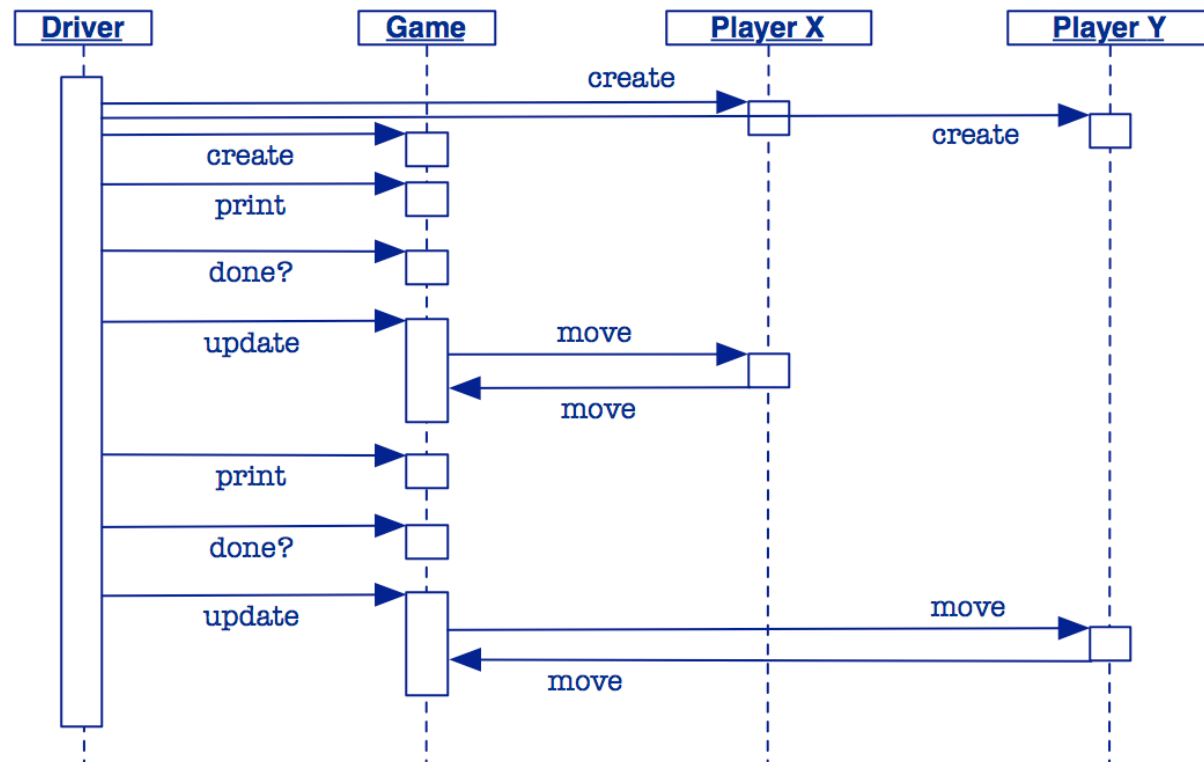
Add methods to make moves, test for winning

# Refining the interactions

We will want both real and test Players,  
*so the Driver should create them*

Updating the  
Game and  
printing it should  
be separate  
operations.

The Game  
should ask the  
Player to make a  
move, and then  
the Player will  
attempt to do so.



# Testing scenarios

Our test scenarios will play and test *scripted* games

```
@Test public void testXWinDiagonal() {
    checkGame("a1\nb2\nc3\n", "b1\nc1\n", "X", 4);
}
// more tests ...

public void checkGame(String Xmoves, String Omoves,
    String winner, int squaresLeft) {
    Player X = new Player('X', Xmoves);    // a scripted player
    Player O = new Player('O', Omoves);
    TicTacToe game = new TicTacToe(X, O);
    GameDriver.playGame(game);
    assertTrue(game.winner().name().equals(winner));
    assertTrue(game.squaresLeft() == squaresLeft);
}
```



# Running the test cases

```

3      |      |
  ---+---+---
2      |      |
  ---+---+---
1      |      |
   a    b    c
Player X moves: X at a1
3      |      |
  ---+---+---
2      |      |
  ---+---+---
1  X    |      |
   a    b    c
...
  
```

```

Player O moves: O at c1
3      |      |
  ---+---+---
2      |  X    |
  ---+---+---
1  X    |  O    |  O
   a    b    c
Player X moves: X at c3
3      |      |  X
  ---+---+---
2      |  X    |
  ---+---+---
1  X    |  O    |  O
   a    b    c
game over!
  
```

# The Player

We use different constructors to make real or test Players:

```
public class Player {  
    private final char mark;  
    private final BufferedReader in;
```

A real player reads from the standard input stream:

```
public Player(char mark) {  
    this(mark, new BufferedReader(  
        new InputStreamReader(System.in)  
    ));  
}
```

This constructor just calls another one ...

# Player constructors...

---

But a Player can be constructed that reads its moves from any input buffer:

```
protected Player(char initMark, BufferedReader initIn) {  
    mark = initMark;  
    in = initIn;  
}
```

This constructor is not intended to be called directly.

# Player constructors

A test Player gets its input from a String buffer:

```
public Player(char mark, String moves) {  
    this(mark, new BufferedReader(  
        new StringReader(moves)  
    ));  
}
```

The default constructor returns a dummy Player representing “nobody”

```
public Player() { this(' '); }
```

# Supporting test Players

The Game no longer instantiates the Players, but accepts them as constructor arguments:

```
public TicTacToe(Player playerX, Player player0)
{    // ...
    player = new Player[2];
    player[X] = playerX;
    player[0] = player0;
}
```

# Delegating Responsibilities...

When Driver updates the Game, the Game just asks the Player to make a move:

```
public void update() throws IOException {  
    player[turn].move(this);  
}
```

Note that the Driver may not do this directly!

# Delegating Responsibilities...

The Player, in turn, calls the Game's move() method:

```
public void move(char col, char row, char mark) {  
    System.out.println(mark + " at " + col + row);  
    this.set(col, row, mark);  
    this.squaresLeft--;  
    this.swapTurn();  
    this.checkWinner();  
}
```

# Small Methods

---

Introduce methods that make the *intent* of your code clear.

```
public boolean notOver() {  
    return this.winner().isNobody()  
        && this.squaresLeft() > 0;  
}  
private void swapTurn() {  
    turn = (turn == X) ? 0 : X;  
}
```

Well-named variables and methods typically eliminate the need for explanatory comments!



# Accessor Methods

Accessor methods protect clients from changes in implementation:

```
public Player winner() {  
    return winner;  
}  
public int squaresLeft() {  
    return this.squaresLeft;  
}
```

When should instance variables be public?

*Almost never! Declare public accessor methods instead.*

# getters and setters in Java

---

Accessors in Java are known as “getters” and “setters”.

Accessors for a variable `x` should normally be called `getX()` and `setX()`

Frameworks such as EJB depend on this convention!

# Code Smells: TicTacToe.checkWinner()

Duplicated code stinks!  
How can we clean it up?

```
private void checkWinner() {  
    char player;  
    for (char row='3'; row>='1'; row--) {  
        player = this.get('a',row);  
        if (player == this.get('b',row)  
            && player == this.get('c',row)) {  
            this.setWinner(player);  
            return;  
        }  
    }  
}
```

```
for (char col='a'; col <='c'; col++) {  
    player = this.get(col,'1');  
    if (player == this.get(col,'2')  
        && player == this.get(col,'3')) {  
        this.setWinner(player);  
        return;  
    }  
}  
player = this.get('b','2');  
if (player == this.get('a','1')  
    && player == this.get('c','3')) {  
    this.setWinner(player);  
    return;  
}  
if (player == this.get('a','3')  
    && player == this.get('c','1')) {  
    this.setWinner(player);  
    return;  
}  
}
```

# GameDriver

In order to run test games, we separated Player instantiation from Game playing:

```
public class GameDriver {  
    public static void main(String args[]) {  
        Player X = new Player('X');  
        Player O = new Player('O');  
        TicTacToe game = new TicTacToe(X, O);  
        playGame(game);  
    }  
}
```

How can we make test scenarios play silently?

# What you should know!

---

What is Iterative Development, and how does it differ from the Waterfall model?

How can identifying responsibilities help you to design objects?

Where did the Driver come from, if it wasn't in our requirements?

Why is Winner not a likely class in our TicTacToe design?

Why should we evaluate assertions if they are all supposed to be true anyway?

What is the point of having methods that are only one or two lines long?

# Can you answer these questions?

---

Why should you expect requirements to change?

In our design, why is it the Game and not the Driver that prompts a Player to move?

When and where should we evaluate the TicTacToe invariant?

What other tests should we put in our TestDriver?

How does the Java compiler know which version of an overloaded method or constructor should be called?

# License



## Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

- Share: copy and redistribute the material in any medium or format
- Adapt: remix, transform, and build upon the material for any purpose, even commercially

The licensor cannot revoke these freedoms as long as you follow the license terms



**Attribution:** you must give appropriate credit



**ShareAlike:** if you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original

Complete license: <https://creativecommons.org/licenses/by-sa/4.0/>

Original version of this lecture is from Prof. Oscar Nierstrasz



**dcc**

CIENCIAS DE LA COMPUTACIÓN  
UNIVERSIDAD DE CHILE

[www.dcc.uchile.cl](http://www.dcc.uchile.cl)

f @ in / DCCUCHILE