6. Iterative Development

Oscar Nierstrasz
Iterative Development

Sources


Roadmap

> The iterative software lifecycle
> Responsibility-driven design
> TicTacToe example
  — Identifying objects
  — Scenarios
  — Test-first development
  — Printing object state
  — Testing scenarios
  — Representing responsibilities as contracts
Roadmap

> The iterative software lifecycle
> Responsibility-driven design
> TicTacToe example
  — Identifying objects
  — Scenarios
  — Test-first development
  — Printing object state
  — Testing scenarios
  — Representing responsibilities as contracts
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 the standard software process?
Roadmap

- The iterative software lifecycle
- **Responsibility-driven design**
- TicTacToe example
  - Identifying objects
  - Scenarios
  - Test-first development
  - Printing object state
  - Testing scenarios
  - Representing responsibilities as contracts
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

> The iterative software lifecycle
> Responsibility-driven design
> TicTacToe example
  — Identifying objects
  — Scenarios
  — Test-first development
  — Printing object state
  — Testing scenarios
  — Representing responsibilities as contracts
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!

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

> The iterative software lifecycle
> Responsibility-driven design
> TicTacToe example
  — Identifying objects
  — Scenarios
  — Test-first development
  — Printing object state
  — Testing scenarios
  — Representing responsibilities as contracts
Tic Tac Toe Objects

Some objects can be identified from the requirements:

<table>
<thead>
<tr>
<th>Objects</th>
<th>Responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game</td>
<td>Maintain game rules</td>
</tr>
<tr>
<td>Player</td>
<td>Make moves</td>
</tr>
<tr>
<td></td>
<td>Mediate user interaction</td>
</tr>
<tr>
<td>Compartment</td>
<td>Record marks</td>
</tr>
<tr>
<td>Figure (State)</td>
<td>Maintain game state</td>
</tr>
</tbody>
</table>

*Entities with clear responsibilities are more likely to end up as objects in our design.*
**Tic Tac Toe Objects ...**

*Others can be eliminated:*

<table>
<thead>
<tr>
<th>Non-Objects</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crosses, ciphers</td>
<td>Same as Marks</td>
</tr>
<tr>
<td>Marks</td>
<td>Value of Compartment</td>
</tr>
<tr>
<td>Vertical lines</td>
<td>Display of State</td>
</tr>
<tr>
<td>Horizontal lines</td>
<td>ditto</td>
</tr>
<tr>
<td>Winner</td>
<td>State of Player</td>
</tr>
<tr>
<td>Row</td>
<td>View of State</td>
</tr>
<tr>
<td>Diagonal</td>
<td>ditto</td>
</tr>
</tbody>
</table>
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

> The iterative software lifecycle
> Responsibility-driven design
> TicTacToe example
  — Identifying objects
  — **Scenarios**
    — Test-first development
    — Printing object state
    — Testing scenarios
    — Representing responsibilities as contracts
A scenario describes a typical sequence of interactions:

Are there other equally valid scenarios for this problem?
Our first version does very little!

class GameDriver {
    static public 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

> The iterative software lifecycle
> Responsibility-driven design
> TicTacToe example
  — Identifying objects
  — Scenarios
  — **Test-first development**
  — Printing object state
  — Testing scenarios
  — Representing responsibilities as contracts
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!
public class TicTacToeTest {
    private TicTacToe game;

    @Before public void setUp() {
        game = new TicTacToe();
    }

    @Test public void testState() {
        assertEquals(game.get('a','1'), ' ');
        assertEquals(game.get('c','3'), ' ');
        game.set('c','3','X');
        assertEquals(game.get('c','3'), 'X');
        game.set('c','3',' ');
        assertEquals(game.get('c','3'), ' ');
        assertFalse(game.inRange('d','4'));
    }
}
Generating methods

Test-first programming can drive the development of the class interface …
Roadmap

- The iterative software lifecycle
- Responsibility-driven design
- TicTacToe example
  - Identifying objects
  - Scenarios
  - Test-first development
- **Printing object state**
  - Testing scenarios
  - Representing responsibilities as contracts
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,' ');
    }
    ...
}
Checking pre-conditions

set() and get() translate from chess notation to array indices.

```java
public void set(char col, char row, char mark) {
    assert(inRange(col, row)); // NB: precondition
    gameState[col-'a'][row-'1'] = mark;
}

public char get(char col, char row) {
    assert(inRange(col, row));
    return gameState[col-'a'][row-'1'];
}

public boolean inRange(char col, char row) {
    return (('a'<=col) && (col<='c')
     && ('1'<=row) && (row<='3'));
}
```
Printing the State

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

```
  3 |   |  
  ---+---+---
  2 |   |  
  ---+---+---
  1 |   |  
    a | b | c
```
TicTacToe.toString()

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

```java
public String toString() {
    StringBuffer 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
     ");
    return(rep.toString());
}
```
Roadmap

- The iterative software lifecycle
- Responsibility-driven design
- TicTacToe example
  - Identifying objects
  - Scenarios
  - Test-first development
  - Printing object state
  - Testing scenarios
  - Representing responsibilities as contracts
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

```java
@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

Player O moves: O at c1

Player X moves: X at c3

Player X moves: X at a1

...
The Player

We use **different constructors** to make real or test Players:

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

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

*A real player reads from the standard input stream:*

*This constructor just calls another one ...*
But a Player can be constructed that reads its moves from any input buffer:

```java
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:

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

The default constructor returns a dummy Player representing “nobody”

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

> The iterative software lifecycle
> Responsibility-driven design
> TicTacToe example
  — Identifying objects
  — Scenarios
  — Test-first development
  — Printing object state
  — Testing scenarios
  — Representing responsibilities as contracts
**Tic Tac Toe Contracts**

*Explicit invariants:*
- turn (current player) is either X or O
- X and O swap turns (turn never equals previous turn)
- game state is 3×3 array marked X, O or blank
- winner is X or O iff winner has three in a row

*Implicit invariants:*
- initially winner is nobody; initially it is the turn of X
- game is over when all squares are occupied, or there is a winner
- a player cannot mark a square that is already marked

*Contracts:*
- the current player may make a move, if the invariants are respected
Encoding the contract

We must introduce state variables to implement the contracts

```java
public class TicTacToe {
    static final int X = 0; // constants
    static final int O = 1;
    private char[][] gameState;
    private Player winner = new Player(); // = nobody
    private Player[] player;
    private int turn = X; // initial turn
    private int squaresLeft = 9;
    ...
}
```
Supporting test Players

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

```java
public TicTacToe(Player playerX, Player playerO)
{
    // ...
    player = new Player[2];
    player[X] = playerX;
    player[O] = playerO;
}
```
Invariants

These conditions may seem obvious, which is exactly why they should be checked ...

```java
private boolean invariant() {
    return (turn == X || turn == O)
        && (this.notOver()
            || this.winner() == player[X]
            || this.winner() == player[O]
            || this.winner().isNobody())
        && (squaresLeft < 9 // else, initially:
            || turn == X && this.winner().isNobody());
}
```

Assertions and tests often tell us what methods should be implemented, and whether they should be public or private.
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:

```java
public void move(char col, char row, char mark) {
    assert(notOver());
    assert(inRange(col, row));
    assert(get(col, row) == ' ');
    System.out.println(mark + " at " + col + row);
    this.set(col, row, mark);
    this.squaresLeft--;
    this.swapTurn();
    this.checkWinner();
    assert(invariant());
}
```
Small Methods

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

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

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

Accessor methods protect clients from changes in implementation:

```java
public Player winner() {
    return winner;
}

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

✍️ **When should instance variables be public?**

✔️ **Almost never! Declare public accessor methods instead.**
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;
        }
    }
    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:

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

✨ 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

You are free:
• to copy, distribute, display, and perform the work
• to make derivative works
• to make commercial use of the work

Under the following conditions:

**Attribution.** You must attribute the work in the manner specified by the author or licensor.

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

• For any reuse or distribution, you must make clear to others the license terms of this work.
• Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.