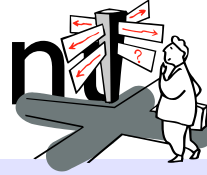


Software Engineering

(Génie logiciel et gestion de projets)

Iterative Development - Responsibility-Driven Design
Roel Wuyts

Iterative Development



Overview

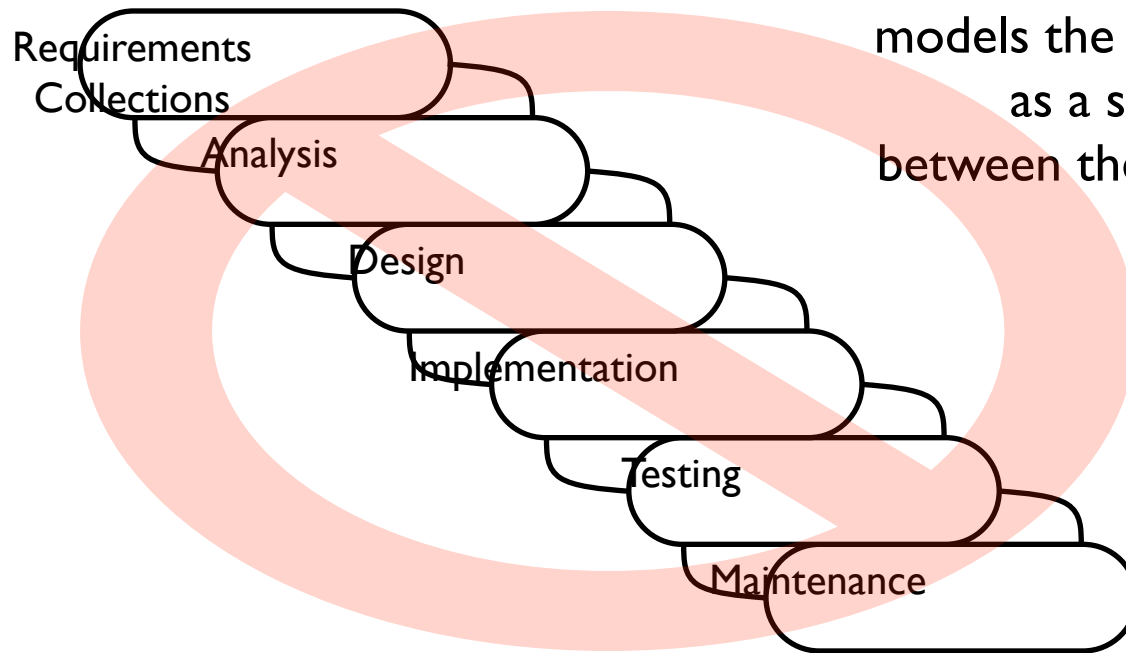
- Iterative development
- Responsibility-Driven Design
 - ☞ How to find the objects ...
 - ☞ TicTacToe example ...

Sources

- Rebecca Wirfs-Brock, Alan McKean, *Object Design — Roles, Responsibilities and Collaborations*, Addison-Wesley, 2003.
- Kent Beck, *Extreme Programming Explained — Embrace Change*, Addison-Wesley, 1999.

The Classical Software Lifecycle

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

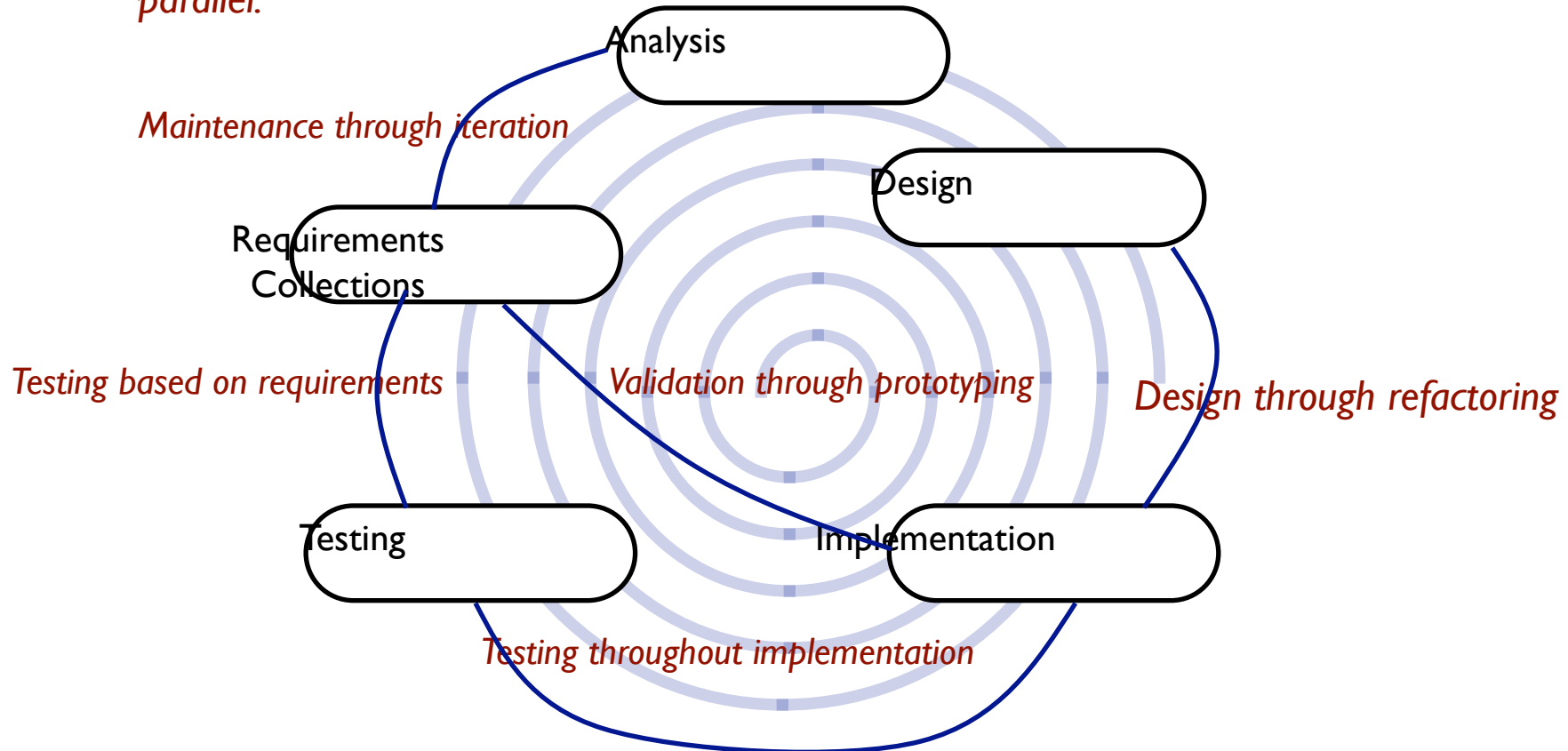


The waterfall model is unrealistic for many reasons, especially:

- 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 the standard software process?*

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:

- Which responsibilities should an object accept?
 - ✓ *“Don't do anything you can push off to someone else.”*

- How much state should an object expose?
 - ✓ *“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.

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.*

Tic Tac Toe Objects

Some objects can be identified from the requirements:

Objects	Responsibilities
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

How can you tell when you have the “right” set of objects?

Each object has a clear and natural set of responsibilities.

Missing Objects

Now we check if there are unassigned responsibilities:

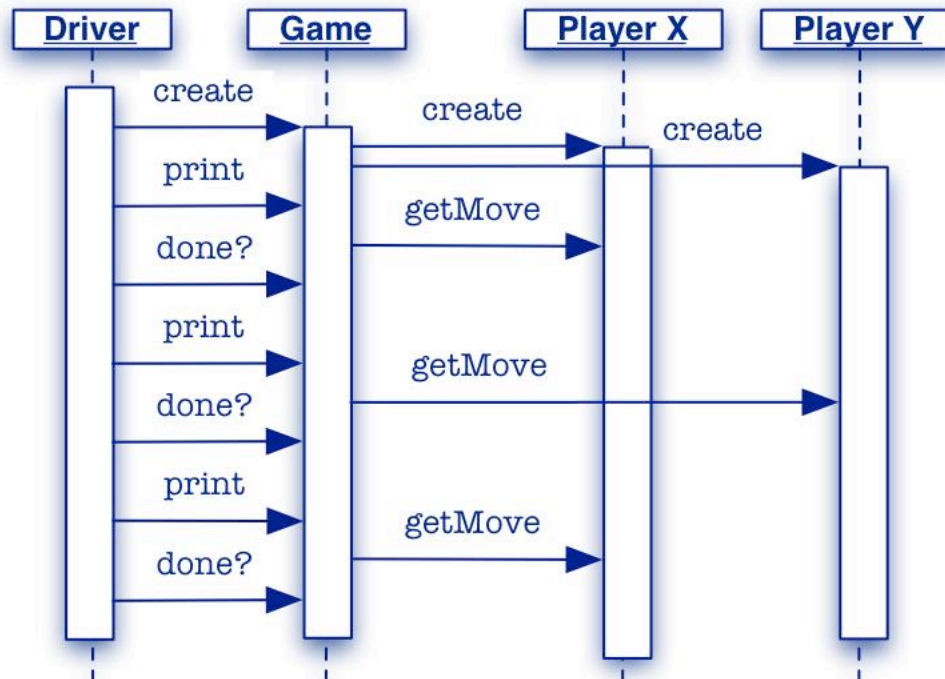
- Who starts the Game?
- Who is responsible for displaying the Game state?
- How do Players know when the Game is over?

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.*

Scenarios

A scenario describes a typical sequence of interactions:



✎ *Are there other equally valid scenarios for this problem?*

Version 1.0 (skeleton)

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.*

Version 1.1 (simple tests)

The state of the game is represented as 3×3 array of chars marked ' ', 'X', or 'O'. We index the state using chess notation, i.e., column is 'a' through 'c' and row is '1' through '3'.

```
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.

```
private void set(char col, char row, char mark) {
    assert(inRange(col, row)); // NB: precondition
    gameState_[col-'a'][row-'1'] = mark;
}
private char get(char col, char row) {
    assert(inRange(col, row));
    return gameState_[col-'a'][row-'1'];
}
private boolean inRange(char col, char row) {
    return (('a'<=col) && (col<='c')
        && ('1'<=row) && (row<='3'));
}
```

Testing the new methods

For now, we just exercise the new `set()` and `get()` methods:

```
public void test() {
    System.err.println("Started TicTacToe tests");
    assert(this.get('a','1') == ' ');
    assert(this.get('c','3') == ' ');
    this.set('c','3','X');
    assert(this.get('c','3') == 'X');
    this.set('c','3',' ');
    assert(this.get('c','3') == ' ');
    assert(!this.inRange('d','4'));
    System.err.println("Passed TicTacToe tests");
}
```

Testing the application

If each class provides its own `test()` method, we can bundle our unit tests in a single driver class:

```
class TestDriver {  
    static public void main(String args[]) {  
        TicTacToe game = new TicTacToe();  
        game.test();  
    }  
}
```

Printing the State

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

```
3      |      |
  ---+---+---
2      |      |
  ---+---+---
1      |      |
   a    b    c
```

➤ How do you make an object printable?

✓ *Override `Object.toString()`*

TicTacToe.toString()

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

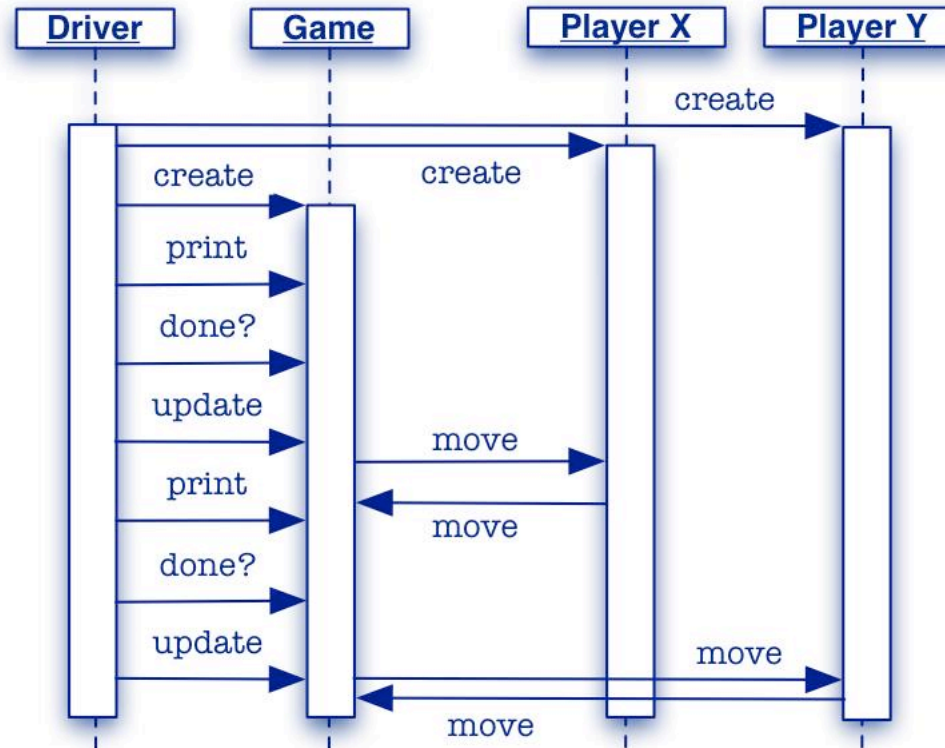
```
public String toString() {
    StringBuffer rep = new StringBuffer();
    for (char row='3'; row>='1'; row--) {
        rep.append(row);
        rep.append(" ");
        for (char col='a'; col <='c'; col++) { ... }
        ...
    }
    rep.append(" a b c\n");
    return(rep.toString());
}
```

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.*



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

Version 1.2 (functional)

We must introduce state variables to implement the contracts

```
public class TicTacToe {
    private char[][] gameState_;
    private Player winner_ = new Player(); // = nobody
    private Player[] player_;
    private int turn_ = X; // initial turn
    private int squaresLeft_ = 9;
    static final int X = 0; // constants
    static final int O = 1;
    ...
}
```

Supporting test Players

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

```
public TicTacToe(Player playerX, Player playerO)
    throws AssertionError
{
    // ...
    player_ = new Player[2];
    player_[X] = playerX;
    player_[O] = playerO;
}
```

Invariants

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

```
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:

```
public void move(char col, char row, char mark)
                throws AssertionError
{
    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.

```
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:

```
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()
    throws AssertionError
{
    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[]) {
        try {
            Player X = new Player('X');
            Player O = new Player('O');
            TicTacToe game = new TicTacToe(X, O);
            playGame(game);
        } catch (AssertionException err) {
            ...
        }
    }
}
```

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 mark, BufferedReader in) {  
    mark_ = mark;  
    in_ = in;  
}
```

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(' ');  
}
```

Defining test cases

The TestDriver builds games using test Players that represent various test cases:

```
public class TestDriver {
    private static String testX1 = "a1\nb2\nc3\n";
    private static String testO1 = "b1\nc1\n";
    // + other test cases ...

    public static void main(String args[]) {
        testGame(testX1, testO1, "X", 4);
        // ...
    }
    ...
}
```

Checking test cases

The TestDriver checks if the results are the expected ones.

```
public static void testGame(String Xmoves,
    String Omoves, String winner, int squaresLeft)
{
    try {
        Player X = new Player('X', Xmoves);
        Player O = new Player('O', Omoves);
        TicTacToe game = new TicTacToe(X, O);
        GameDriver.playGame(game);
        assert(game.winner().name().equals(winner));
        assert(game.squaresLeft() == squaresLeft);
    } catch (AssertionException err) { ... }
}
```

Running the test cases

Started testGame test

```
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!

Passed testGame test

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?*

Wrap-up

- You now know:

References
