Refactoring
(continued)

Four Design Strategies

Responsive Design

Kent Beck

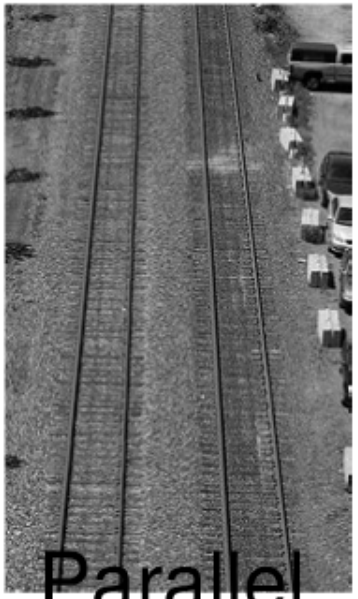
Three Rivers Institute

For more information contact kent@threeriversinstitute.org

http://www.infoq.com/presentations/responsive-design (32:00-1:03:45)

Strategies

Leap



Parallel

Stepping Stone

Simplification

Summary: Four Design Strategies

- Principle: Move the design in **safe steps**.
- **Leap**
  - Just do it.
  - Pay the price for iteration vs. Change it all at once and sometimes it blows up
- **Parallel**
  - Operate two designs in parallel for a while. When the two are equivalent, switch to the new design.
- **Stepping Stone**
  - "I can't get from here to here safely, but if I had a ???, then getting from there to here would be a safe step (or at least progress)."
  - Danger: Temptation to over-engineer the stepping stone.
- **Simplification**
  - "What's the least I could do that would be progress?"
  - Example: Sudoku solver for 1 by 1 grid.
  - "I can always simplify a problem so much that it's a safe step."

Refactoring 1: Composed Method

```
public String toString() {
    StringBuilder sb = new StringBuilder();
    for (int row = 0; row < board.length; row++) {
        for (int col = 0; col < board[row].length; col++) {
            if (fallingBlock != null && fallingBlock.isAt(row, col)) {
                sb.append(fallingBlock.style());
            } else {
                sb.append(board[row][col]);
            }
        }
        sb.append('\n');
    }
    return sb.toString();
}
```

Refactoring *Board.java* after passing *ANewBoard.testIsEmpty()* and *WhenABlockIsDropped.testItStartsFromTheTopMiddle()*

Refactoring 1: Composed Method

```
public String toString() {
    StringBuilder sb = new StringBuilder();
    for (int row = 0; row < board.length; row++) {
        for (int col = 0; col < board[row].length; col++) {
            char c;
            if (fallingBlock != null && fallingBlock.isAt(row, col)) {
                c = fallingBlock.style();
            } else {
                c = board[row][col];
            }
            sb.append(c);
        }
        sb.append('\n');
    }
    return sb.toString();
}
```

Refactoring 1: Composed Method

```
public String toString() {
    StringBuilder sb = new StringBuilder();
    for (int row = 0; row < board.length; row++) {
        for (int col = 0; col < board[row].length; col++) {
            sb.append(cellAt(row, col));
        }
        sb.append('\n');
    }
    return sb.toString();
}

private char cellAt(int row, int col) {
    if (fallingBlock != null && fallingBlock.isAt(row, col)) {
        return fallingBlock.style();
    } else {
        return board[row][col];
    }
}
```

Refactoring 2: Duplicate Code

```
public class Board {
...
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (int row = 0; row < board.length; row++) {
            for (int col = 0; col < board[row].length; col++) {
                sb.append(cellAt(row, col));
            }
            sb.append('\n');
        }
        return sb.toString();
    }

    private char cellAt(int row, int col) {
        if (fallingBlock != null && fallingBlock.isAt(row, col)) {
            return fallingBlock.style();
        } else {
            return board[row][col];
        }
    }
}
```


Refactoring after passing all of *RotatingTetrominoesTest*

Refactoring 2: Duplicate Code

```
public class Piece implements Rotatable {
...
    public String toString() {
        return arraysToLines(blocks);
    }

    private static String arraysToLines(char[][] x) {
        StringBuilder sb = new StringBuilder();
        for (char[] line : x) {
            for (char c : line) {
                sb.append(c);
            }
            sb.append('\n');
        }
        return sb.toString();
    }
}
```

Refactoring 2: Duplicate Code

```
public interface Table {

    int rows();

    int columns();

    char cellAt(int row, int col);
}
```

Refactoring 2: Duplicate Code

```
public class Board implements Table {
...
    public String toString() {
        Table t = this;
        StringBuilder sb = new StringBuilder();
        for (int row = 0; row < t.rows(); row++) {
            for (int col = 0; col < t.columns(); col++) {
                sb.append(t.cellAt(row, col));
            }
            sb.append('\n');
        }
        return sb.toString();
    }

    public char cellAt(int row, int col) {
        if (fallingBlock != null && fallingBlock.isAt(row, col)) {
            return fallingBlock.style();
        } else {
            return board[row][col];
        }
    }
}
```

Refactoring 2: Duplicate Code

```
public class Board implements Table {
...
    public String toString() {
        return visualize(this);
    }

    private static String visualize(Table t) {
        StringBuilder sb = new StringBuilder();
        for (int row = 0; row < t.rows(); row++) {
            for (int col = 0; col < t.columns(); col++) {
                sb.append(t.cellAt(row, col));
            }
            sb.append('\n');
        }
        return sb.toString();
    }

    public char cellAt(int row, int col) {
        if (fallingBlock != null && fallingBlock.isAt(row, col)) {
            return fallingBlock.style();
        } else {
            return board[row][col];
        }
```

Refactoring 2: Duplicate Code

```
public class Board implements Table {
...
    public String toString() {
        return TableAsciiView.visualize(this);
    }

    public char cellAt(int row, int col) {
        if (fallingBlock != null && fallingBlock.isAt(row, col)) {
            return fallingBlock.style();
        } else {
            return board[row][col];
        }
    }
}
```

Refactoring 2: Duplicate Code

```
public class TableAsciiView {

    public static String visualize(Table t) {
        StringBuilder sb = new StringBuilder();
        for (int row = 0; row < t.rows(); row++) {
            for (int col = 0; col < t.columns(); col++) {
                sb.append(t.cellAt(row, col));
            }
            sb.append('\n');
        }
        return sb.toString();
    }
}
```

Refactoring 2: Duplicate Code

```
public class Piece implements Table, Rotatable {
...
    public int rows() {
        return blocks.length;
    }

    public int columns() {
        return blocks[0].length;
    }

    public char cellAt(int row, int col) {
        return blocks[row][col];
    }

    public String toString() {
        return TableAsciiView.visualize(this);
    }
}
```

...and later this helped in making *Tetriminoe*, *Piece* and *Block* polymorphic.