P2 - Exercise Hours

Coding Issues

April 30, 2021
Lorenzo Wipfli
Overview

- RECAP Exercise 7
- Coding Issues
  - Attributes
  - Constants
  - Constants vs. Enumerations
  - Switch Instructions
- Preview Exercise 8
RECAP: Exercise 07

> Game loop and main

```java
public void play(Game game){
    assert game != null;

    boolean validMove = false;

    while (!game.isOver()){
        out.println(game.nextPlayer() + "'s turn");
        game.move(move);
        validMove = true;
    } catch (InvalidMoveException e) {
        out.println("Try again");
    } catch (NoSuchElementException e) {
        return;
    }
}

out.println(winner.toString() + " has won the game");
out.println(loser.toString() + " has lost the game");
```

```java
public static void main(String[] args){
    boolean invalidSpecFile;
    Game game = null;
    do {
        try {
            game = new Game(new Scanner(System.in), System.out);
            invalidSpecFile = false;
        } catch (FileNotFoundException | InvalidParserInputException e) {
            invalidSpecFile = true;
        }
    } while (invalidSpecFile);

    game.play(game);
```
RECAP: Exercise 07

> Movement

```java
public void move(int row, int column, int direction) throws NotValidPositionException {
    assert board.isValidPosition(row, column);
    Square currentSquare = board.getSquare(row, column);
    assert currentSquare.isDark();
    assert currentSquare.isOccupied();
    Piece currentPiece = currentSquare.getPiece();
    assert currentPiece != null;

    if (! (currentPiece.isBlack() -- blackPlayer)) throw new NotValidPositionException("Pieces color does not match player color.");

    if (currentPiece.isValidDirection(direction)) {
        Square newSquare = executeMove(currentSquare, currentPiece, direction);
        if (! board.isAdjacentSquare(currentSquare, newSquare, direction)) cascadeAfterRemovingPieces(currentPiece, newSquare);
    } else throw new NotValidPositionException("Invalid direction for this piece");

    currentPiece.endMove();
    assert currentPiece.isBlack() == isBlackPlayer();
    assert invariant();
}
```
RECAP: Exercise 07

> Rendering

Black's turn:
Your valid moves are:
| b3-a4 | b3-c4 | d3-c4 | d3-e4 | f3-e4 | f3-g4 | h3-g4 |
Your move:
Coding Issues - Attributes

```java
public class Board {
    public Square firstSquare;
}

public class Game {
    public void client() {
        Square start = board.firstSquare;
        //...
    }
}

//... Other clients that access or use firstSquare.
```
Coding Issues - Attributes

```java
public class Board {
    public List<Squares> squares;
}
```

```java
public class Game {
    public void client() {
        Square start = board.firstSquare;
        // ...
    }
}
```

//... Other clients that access or use firstSquare.

What if we change ‘firstSquare’?
Coding Issues - Attributes

```java
public class Board {
    public List<Squares> squares;
}
```

```java
public class Game {
    public void client() {
        Square start = board.firstSquare;
        //...
    }
}
```

//... Other clients that access or use firstSquare.

Does not work anymore! We need to change the code in all clients.
Coding Issues - Attributes

```java
public class Board {
    public List<Squares> squares;
}
```

```java
public class Game {
    public void client() {
        Square start = board.squares.get(0);
        //...
    }
}
```

//... Other clients that access or use firstSquare.

Now it works, but changing client code is not nice!
Coding Issues - Attributes

```java
public class Board {
    protected Square firstSquare;

    public Square getFirstSquare(){
        return firstSquare
    }

    public void setFirstSquare(Square square){
        firstSquare = square;
    }
}

public class Game {
    public void client() {
        Square start = board.getFirstSquare();
        //...
    }
}

//... Other clients that access or use firstSquare.

Better solution: Use getters/setters
Allows changes in implementation without affecting clients.
```
Coding Issues - Attributes

```java
public class Board {
    protected List<Squares> squares;

    public Square getFirstSquare(){
        return squares.get();
    }
    public void setFirstSquare(Square square){
        squares.set(0,square);
    }
}
```

```java
public class Game {
    public void client() {
        Square start = board.getFirstSquare();
        //...
    }
}
```

//... Other clients that access or use firstSquare.

If we now change the implementation in the Board class, the code of the client remains unchanged.
public class Board {
    protected final int BOARD_SIZE;
    protected final char[] ROW_NAMES = {'A', 'B', 'C'};
    protected final int[] COL_NAMES = {1, 2, 3};
}
Coding Issues - Constants

```java
public class Board {
    protected final int BOARD_SIZE;
    protected final char[] ROW_NAMES = {'A', 'B', 'C'};
    protected final int[] COL_NAMES = {1, 2, 3};
}
```

These are not constants.
public class Board {
    protected final int BOARD_SIZE;
    protected final char[] ROW_NAMES = {'A', 'B', 'C'};
    protected final int[] COL_NAMES = {1, 2, 3};
}

These are not constants.

public class Board {
    protected final int boardSize;
    protected final char[] rowNames = {'A', 'B', 'C'};
    protected final int[] colNames = {1, 2, 3};
}

Use camelCase for attributes
Coding Issues - Constants

```java
public class Board {
    protected final int BOARD_SIZE;
    protected final char[] ROW_NAMES = {'A', 'B', 'C'};
    protected final int[] COL_NAMES = {1, 2, 3};
}

These are not constants.

public class Board {
    protected final int boardSize;
    protected final char[] rowNames = {'A', 'B', 'C'};
    protected final int[] colNames = {1, 2, 3};
}

Use camelCase for attributes

public class Board {
    protected static final int boardSize;
    protected static final char[] rowNames = {'A', 'B', 'C'};
    protected static final int[] colNames = {1, 2, 3};
}

Use 'static final' for constants
```
final class Direction {
    protected static final int LEFT = 1;
    protected static final int RIGHT = 2;
    protected static final int UP = 3;
    protected static final int DOWN = 4;
}

public static Command createCommand(int type) {
    if (type == LEFT) {
        return new CommandLeft();
    } else if (type == RIGHT) {
        return new CommandRight();
    } else {
        // ...
    }
    return null;
}
final class Direction {
    protected static final int LEFT = 1;
    protected static final int RIGHT = 2;
    protected static final int UP = 3;
    protected static final int DOWN = 4;
}

public static Command createCommand(int type) {
    if (type == LEFT) {
        return new commandLeft();
    } else if (type == RIGHT) {
        return new commandRight();
    } else {
        // …
        return null;
    }
}
 Coding Issues – Constants vs. Enumerations

```java
enum Direction {
    LEFT,
    RIGHT,
    UP,
    DOWN
}

Command createCommand(Direction dir) {
    switch (dir) {
        case LEFT: return new CommandLeft();
        case RIGHT: return new CommandRight();
        case UP: return new CommandUp();
        case DOWN: return new CommandDown();
    }
    // ...
}
```
enum Direction {
    LEFT,
    RIGHT,
    UP,
    DOWN
}

Command createCommand(Direction dir) {
    switch (dir) {
        case LEFT: return new CommandLeft();
        case RIGHT: return new CommandRight();
        case UP: return new CommandUp();
        case DOWN: return new CommandDown();
    }
    // ...
}
interface CommandFactory {
    Command create();
}

dependence Direction implements CommandFactory {
    LEFT {
        public Command create() {
            return new CommandLeft();
        }
    },
    RIGHT {
        public Command create() {
            return new CommandRight();
        }
    },
    // ...
}
Coding Issues – Constants vs. Enumerations

```java
interface CommandFactory {
    Command create();
}

denum Direction implements CommandFactory {
    LEFT {
        Command create() {
            return new CommandLeft();
        }
    },
    RIGHT {
        Command create() {
            return new CommandRight();
        }
    },
    // …
}

// Client
Command createCommand(Direction dir) {
    return dir.create();
}
```

 Enums can implement interfaces.
Coding Issues – Switch Instructions

```java
private int convertToInt(char c) {
    int output;
    switch (c) {
        case 'a': output = 0;
        case 'b': output = 1;
        case 'c': output = 2;
        case 'd': output = 3;
        case 'e': output = 4;
        case 'f': output = 5;
        default: output = 10;
    }
    return output;
}
```

What does convertToInt('e') return?
private int convertToInt(char c) {
    int output;
    switch (c) {
        case 'a': output = 0;
        case 'b': output = 1;
        case 'c': output = 2;
        case 'd': output = 3;
        case 'e': output = 4;
        case 'f': output = 5;
        default: output = 10;
    }
    return output;
}
private int convertToInt(char c) {
    int output;
    switch (c) {
        case 'a': output = 0; break;
        case 'b': output = 1; break;
        case 'c': output = 2; break;
        case 'd': output = 3; break;
        case 'e': output = 4; break;
        case 'f': output = 5; break;
        default: output = 10; break;
    }
    return output;
}
private boolean isLowercaseLetterBeforeE(char c) {
    boolean result;
    switch (c) {
        case 'a':
        case 'b':
        case 'c':
        case 'd':
            result = true;
            break;
        default:
            result = false;
            break;
    }
    return result;
}
Coding Issues – Switch Instructions

```java
private boolean isLowercaseLetterBeforeE(char c) {
    return c - 'a' < 4;
}
```

This is a bit simpler...
/**
 * Checks whether the given character comes before ‘e’ in the alphabet
 * @param c a character, must be lowercase letter between ‘a’ & ‘z’
 */
private boolean isLowercaseLetterBeforeE(char c) {
    assert c >= ‘a’ && c <= ‘z’;
    return c – ‘a’ < 4;
}

...but don’t forget your contracts!
> Optional means that the groups that passed all exercises until now can skip this one!

> This is a chance to recover for groups that unfortunately failed an exercise.
Pen and paper exercise!
> Create a GUI for your Checker game.
> Document your sketches
  > Create several sketches for Mobile, Desktop, terminal etc. (at least 6)
  > The sketches should be different, at least 3 radically different.
Prototypes

- Choose one of the sketches and show different states of the game via the prototype.
- For example, Welcome state of the game, Game params, Player’s turn, winning screen and after game screen. (All needed are listed in exercise description.)
- Use physical objects to represent the objects whenever possible e.g pieces can be a paper craft.
> See exercise_08.md for more information.
Thank you!