P2: Coding Issues & Quoridor Game

Claudio Corrodi

April 7, 2017
Coding Issues: Attributes

class Board {
    public Square firstSquare;
}

class Game {
    public void client() {
        Square start = board.firstSquare;
        // ...
    }
}
What if we change “firstSquare”? Does not work anymore! We need to change code in all clients!
Coding Issues: Attributes

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

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

What if we change “firstSquare”? 

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

class Board {
    private Square firstSquare;

    public Square getFirstSquare() {
        return firstSquare;
    }
    public void setFirstSquare(Square aSquare) {
        firstSquare = aSquare;
    }
}

private void client() {
    Square start = board.getFirstSquare();
    // …
}

With getters/setters, we can change the implementation without affecting clients.
class Board {
    private List<Square> squares;

    public Square getFirstSquare() {
        return squares.get(0);
    }

    public void setFirstSquare(Square aSquare) {
        squares.set(0, aSquare);
    }
}

private void client() {
    Square start = board.getFirstSquare();
    // ...
}

With getters/setters, we can change the implementation without affecting clients.
Coding Issues: Attributes

- Make attributes protected
  - Subclasses should be able to access own state

- Use getters and setters to make them available to clients
  - Does not expose raw data structures
  - We can increase the complexity of getters and setters without worrying about clients
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

use camelCase for attributes
'static final' for 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
public class Board {
    protected final int BOARD_SIZE;
    protected final char[] ROW_NAMES = { 'A', 'B', 'C' };
    protected final int[] COL_NAMES = { 1, 2, 3};
}

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

public class Board {
    protected static final int BOARD_SIZE = 3;
    protected static final char[] ROW_NAMES = { 'A', 'B', 'C' };
    protected static final int[] COL_NAMES = { 1, 2, 3};
}
final class Direction {
    public static final int LEFT = 1;
    public static final int RIGHT = 2;
    public static final int UP = 3;
    public 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 {
    public static final int LEFT = 1;
    public static final int RIGHT = 2;
    public static final int UP = 3;
    public 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;
}
enum Direction {
    LEFT,  
    RIGHT, 
    UP,   
    DOWN
}

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

Command createCommand(Direction dir) {
    switch (dir) {
        case LEFT: return new CommandLeft();
        case RIGHT: return new CommandRight();
        case UP: // ...
        case DOWN: // ...
    }
    // ...
}
Coding Issues: Constants vs enumerations

interface CommandFactory {
    Command create();
}

enum 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();
}
enum Direction implements CommandFactory {
    LEFT {
        // Client
        Command createCommand(Direction dir) {
            return dir.create();
        }
    },
    RIGHT {
        public Command create() {
            return new CommandRight();
        }
    },
    // ...
}
```

Enums can implement interfaces.
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;
        case 'g': output = 6;
        case 'h': output = 7;
        case 'i': output = 8;
        case 'j': output = 9;
        default: output = 10;
    }
    return output;
}
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;
        case 'g': output = 6;
        case 'h': output = 7;
        case 'i': output = 8;
        case 'j': output = 9;
        default: output = 10;
    }
    return output;
}
Coding Issues: Switch instructions

```java
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;
        case 'g': output = 6 break;
        case 'h': output = 7 break;
        case 'i': output = 8 break;
        case 'j': output = 9 break;
        default: output = 10 break;
    }
    return output;
}
```

Don’t forget to break or return
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;
}
private boolean isLowercaseLetterBeforeE(char c) {
    return c - 'a' < 4;
}

This is a bit simpler though
private boolean isLowercaseLetterBeforeE(char c) {
    return c - 'a' < 4;
}

But is it a good implementation?
private boolean isLowercaseLetterBeforeE(char c) {
    assert c >= 'a' && c <= 'z';
    return c - 'a' < 4;
}
/**
 * Checks whether the given character comes 
 * before 'e' in the alphabet. 
 * @param c a character, must be a lowercase 
 * letter between 'a' and 'z' 
 */

private boolean isLowercaseLetterBeforeE(char c) {
    assert c >= 'a' && c <= 'z';
    return c - 'a' < 4;
}
Exercise 7

- Last Quoridor *programming* exercise
- Goal: Finish implementation, polish game and code
- **This exercise counts double!**
  - If you have failed an exercise already, you *must* solve this one
- **(Soft) deadline: 27 April**
  - Hard deadline: 4 May
  - You still need to ask for an extension!
Alternate level parser

7 8 2 3
#aaaaa#
#   #
# P #
###  #
#  #
#  ###
#  A###
#ppp###

P Paul Simon
A Art Garfunkel
Exercise 7: Quoridor iteration 4

- Wall placement algorithm with rigorous tests
- Refactoring, clean up, documentation, contracts, ...
- Complete and polish everything