P2 – Exercise Hour
Outline

• Coding Issues

• Exercise 6: Recap

• Exercise 7: Recap
public class Board {
    public Square firstSquare;
}

class Game {
    public void client() {
        Square start = board.firstSquare;
        // ...
    }
}
public class Board {
    public List<Square> squares;
}

public class Game {
    public void client() {
        Square start = board.firstSquare;
        // ...
    }
}
public class Board {
    public List<Square> squares;
}

public class Game {
    public void client() {
        Square start = board.firstSquare;
        // ...
    }
}
public class Board {
    public List<Square> squares;
}

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

Now it works, but changing client code is not nice!
public class Board {

    protected Square firstSquare;

    public Square getFirstSquare() {
        return firstSquare;
    }

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

Better solution: Use getters/setters
Allows changes in implementation without affecting clients.
public class Board {

    protected List<Square> squares;

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

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

}

public class Game {

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

}

If we now change the implementation in the Board class, the code of the client remains unchanged.
Attributes

• Make attributes protected
  ▪ Subclasses should be able to access their own state

• Use getters/setters to make them available to clients
  ▪ Does not expose raw data structures
  ▪ Increase complexity of getters/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};
}
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
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.

These are not constants.

Use camelCase for attributes

Use `static final` for constants

These are not constants.

Use camelCase for attributes

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

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

Enums can implement interfaces.
interface CommandFactory {
    Command create();
}

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

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

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;
        default: output = 10;
    }
    return output;
}
Switch Instructions

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;
}
private boolean isLowercaseLetterBeforeE(char c) {
    return c - 'a' < 4;
}
/**
 * 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!
Exercise 6: Recap

For the third stage, you should have:

• Player movement
  • Player can move one step in given four directions

• Validate Player movement
  • Check that a player can execute only valid moves

• Add a new tile
  • `C’ Completed tile: when a box is on top of a goal tile, the tile should be changed to a completed tile.

Once you have finished, tag your solution:

git tag --a v3 --m “sokoban3”
git push origin --tags
Exercise 6: Recap

For the fourth stage, you should have:

• Override `toString()` method
  • Write `toString()` method for all main objects such as Game, Player, Tiles.

• Grouping packages
  • Group src files in the `src`, test cases files in `test`, .sok files in `resource`, and exception files in the `exception` package.

• “Refactoring.md”
  • Document at least three scenario where you refactored the existing code.
Exercise 6: Recap

• Testing
  • Cover at least the 5 given cases from the exercise description:
    • Regular placement of box
    • Player movement (cannot move onto illegal_blocked tiles)
    • Player moving the box onto the goal tile
    • Completed tile
    • Player winning the game
Exercise 6: Recap

• Polishing
  • Finish off your implementation. This includes:
    • JavaDoc
    • Design by Contract
    • Responsibility Driven Design

Once you have finished, tag your final solution:

```
git tag -a v4 -m "sokoban4"
git push origin --tags
```
Exercise 7: Recap

• Document your sketches
  • Create several sketches for Mobile, Desktop, terminal etc.
  • The sketches should be different

• Prototypes
  • Choose one of the sketch 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.
  • Use physical objects to represent the objects whenever possible e.g players can be a paper craft.
Information

Next week we will have:

- The last exercise on Smalltalk
- Exam preparation session