JAKOŚĆ PROJEKTU NISKOPOZIOMOWEGO I JEGO IMPLEMENTACJI

1. Wprowadzenie

Inżynieria oprogramowania jako dziedzina nauki powstała w odpowiedzi na rosnące zapotrzebowanie usystematyzowania i skonwencjonalizowania technik wytwarzania oprogramowania, które okaże się dobre tak w krótszej perspektywie (w momencie dystrybucji produktu), jak i w tej dłuższej (w czasie jego eksploatacji).

Druga część postawionego paradygmatu nakłada wiążącą odpowiedzialność na programistów – od tego momentu w ich profesji nie chodzi jedynie o stworzenie kodu, który realizuje raz na zawsze określone wymagania – teraz winni oni dbać wręcz przede wszystkim o to, by jego pielęgnacja była wykonywalna w możliwie krótkim czasie również przez osoby, które nie brały udziału w akcie twórczym, lecz trafiły do projektu w znacznie późniejszym okresie. Dodatkowo, w obliczu potencjalnych zmianach co do oczekiwań w stosunku do produktu (wynikających z najróżniejszych przyczyn), kod winien być na tyle **otwarty na rozszerzenia**, na ile jest to możliwe, by dopasować się do zmieniającego się świata usług. Wyłania się stąd obraz właściwie napisanego kodu, o dość ściśle określonych właściwościach.

Projekt **Fischer chess** starał się, by te właściwości znalazły swoje odzwierciedlenie w przeszło **dwóch tysiącach** linii, które, interpretowane przez maszynę wirtualną Javy, stanowią odpowiedź na realne potrzeby naszych klientów.

2. Konwencja stosowanych nazw

Nawet najlepszy pod względem projektu kod może skutecznie stracić jakość, jeśli piszący go koderzy postanowią storpedować ideę poprzez używanie niedających się nijak usprawiedliwić nazw. Programiści Projektu **Fischer Chess** urzeczywistnili w tym aspekcie najlepsze i powszechnie przyjęte praktyki, opisane chociażby w uznanej książce *Czysty kod*:

a) nazwy w języku angielskim:

Język angielski jest językiem powszechnie znanym; pełni też rolę języka programowania (w sensie filologicznym), co widać już chociażby po słowach kluczowych dowolnego języka programowania. Nazwy klas, funkcji, zmiennych, parametrów czy enumeratorów tworzyliśmy jako słowa bądź skróty słów języka Albionu.

```
public static Color darkTileColour = Color.TOMATO;
public static Color lightTileColour = Color.WHITESMOKE;
public static String graphicFolder = "setOne";

public Parent createContent(boolean FischerOnes, boolean gameFromFile, Stage stage) {
    clearData();
    BorderPane root = new BorderPane();
    Pane center = new Pane();
    Pane right = new Pane();
    root.setRight(right);
    root.setCenter(center);
```

b) nazwy niedługie

```
public void addOurKing(Piece king) {
    for (Piece piece : pieces) {
        piece.setOurKing(king);
    }
}
```

Przyjmuje się, że długość nazw stosowanych w kodzie winna nie przekraczać szesnastu znaków. Prócz niewielu wyjątków Projekt wypełnił tę regułę, ograniczając się do **lapidarnego** nazewnictwa.

c) nazwy sugestywne i powszechnie używane wśród szachowej społeczności

Nawet Osoba, która po raz pierwszy przegląda kod, bez instruktażu jest w stanie wywnioskować, za co odpowiedzialny jest konkretny fragment dzięki samotłumaczącym się nazwom – jeżeli w klasie **Piece** spotka się z polem kindAfterPromotion – może domyślać się, że chodzi o figurę, którą staje się pionek po osiągnięciu pola promocji (nie – za kolor pola, na którym stoi – co mija się z prawdą).

```
private PieceKind kindAfterPromotion;
```

Jeżeli przeglądając dalej omawianą klasę natknie się na metodę findPossibleMoves, przy całym jej skomplikowaniu (wynikającym zresztą z logiki gry szachy) trafnie obstawi, że metoda odpowiedzialna będzie za określenie dla konkretnej bierki pól na szachownicy, na które może się ona ruszyć, nie zaś że np. metoda zapisuje partię do pliku (czego oczywiście nie robi).

```
public void findPossibleMoves(int enPassantXPossibility, boolean checkMode, boolean checkFlag) {
```

d) nazwy camelCase i PascalCase

Te dwie uznane metodyki łączenia części tworzących jedną nazwę widnieją i w naszym kodzie. Jeżeli specjalne względy nie przemawiały za innym użyciem, PascalCase dominuje wśród nazw obiektów, zaś camelCase widnieje w pozostałych przypadkach użycia. Projekt poświęca uwadze czytelnika realizację obu wspomnianych cech kodu na

zamieszczonej grafice (pierwszej z konwencji również na pasku otwartych plików).

3. Podział na klasy

a) wprowadzenie

Z projektu wynikał pierwotny podział na klasy, odpowiedzialne za udostępnianie pozostałym określonych funkcjonalności. Kod **Fischer chess** zgodnie z wymogami programowania obiektowego, dzieli się na odpowiednie moduły (szczegółowy opis obowiązków poszczególnych plików .java można znaleźć w pliku **Jakość projektu architektury**)

b) długość klas

Projekt **Fischer chess** starał się ograniczyć długość poszczególnych klas, by zachować czytelność. Rzeczywiście, lwia ich część mieści się w zakresie do około dwustu linii. Na

```
public static final int TILE_SIZE = 80;
public static final int SQUARE_NUMBER = 8;
public static final int PIECE_SIZE = 80;
public static final int PROMOTION_BUTTON_SIZE = TILE_SIZE / 2;
```

zarzut o nadmiernej długości klasy Main.java Projekt odpowiada, że: po pierwsze, klasa ta jest punktem odniesienia dla większości klas pozostałych, posiadając także metadane wyznaczające ramy całej aplikacji (chociażby zestaw pól **public static final** opisujących wymiary kluczowych obiektów mnogiej liczby składowych). Poza tym, klasa Main posiada klasę wewnętrzną Clock, zatem w rzeczywistości pod względem teorii architektury plik Main.java zawiera dwie, nie zaś jedną klasę.

```
448
449     public class Clock {
450
451         private HBox timer;
452
453         private short tinyTimer;
454
455         private Label tensOfMinutes;
456         private Label onesOfMinutes;
457
458         private Label tensOfSeconds;
459         private Label onesOfSeconds;
460
461         private Label semicolon;
```

4. Hermetyzacja

```
private PieceColour colour;
private PieceKind kind;
private double mouseX, mouseY;
private double oldX, oldY;
private TurnIndicator onMove;
private Square coordinates;
private ArrayList<Move> possibleMoves;
private Tile[][] board;
private boolean isTargeted;
private Army enemyArmy;
private Piece ourKing;
private Rectangle back;
private PieceKind kindAfterPromotion;
private int promotionMoveNumber;
```

Projekt **Fischer Chess** chroni wielką ilość zasobów prywatnych poszczególnych klas. Pola (jak i niektóre metody), oznaczone specyfikatorem dostępu **private**, mogą być czytane, modyfikowane (lub, w przypadku metod prywatnych: wywoływane), dzięki całym zastępom tak zwanych geterów i seterów – to jest metod publicznych, które zezwalają obiektom wobec danej klasy zewnętrznym

(oraz jej własnym metodom) na odczyt i modyfikację jej zasobów własnych. Dzięki takiemu pośrednictwu, klasa utrzymuje kontrolę nad kolejnością i sposobem akcesji jej pól. Bodaj najlepszą ilustracją dla niniejszego punktu będzie pula rozwiązań z klasy **Piece.java**. Naturalnie, przykładów na implementację opisywanego paradygmatu programowania obiektowego można szukać w wielu innych ustępach kodu.

```
public double getOldX() { return oldX; }

public Square getCoordinates() { return coordinates; }

public double getOldY() { return oldY; }

public PieceKind getKind() { return kind; }

public void setIsTargeted() { isTargeted = true; }

public void unsetIsTargeted() { isTargeted = false; }

public boolean getIsTargeted() { return isTargeted; }

public void setEnemyArmy(Army enemyArmy) { this.enemyArmy = enemyArmy; }

public void setOurKing(Piece ourKing) { this.ourKing = ourKing; }

public ArrayList<Move> getPossibleMoves() { return possibleMoves; }

public void setKindAfterPromotion(PieceKind kind) {
    kindAfterPromotion = kind;
}

public void setKind(PieceKind kind) {
    this.kind = kind;
    String src = getImageSource(colour, kind,  mode: 0);
```

5. Magiczne liczby

Magiczne liczby to literały liczbowe używane wprost w kodzie, które w danym kontekście są co najmniej niejasne dla każdej Osoby, która nie tworzyła kodu. Nowoczesna myśl programistyczna stara się rugować magiczne liczby z kodu. Uzasadnienie takiej praktyki, prócz zwykłej niejasności, jest fakt, że w przypadku nawet marginalnej zmiany koncepcji wymuszającej zmianę wartości numerycznej literału jest zastosowanie tej modyfikacji w **każdym miejscu** kodu, w którym ów literał występował.

```
public class Square {
    private int x;
    private int y;
    public final int charConstant = 97;

    Square(int x, int y) {
        this.x = x;
        this.y = y;
    }

    int getX() { return x; }

    char getXasChar(){
        int n = x + charConstant;
        return (char)n;
    }

    int getY() { return y; }

    void set(int x, int y) {
        this.x = x;
        this.y = y;
    }
```

Alternatywne podejście, używające stałej (np. statycznej bądź finalnej) zamiast grupy równoznacznych literałów – niweluje oba niekorzystne nazwiska – literał staje się zrozumiały z uwagi na nazwę (jeśli spełnia ona np. warunki wyszczególnione w 2.), zaś alternatywna wartość może zostać zaaplikowana poprzez przepisanie jednej linii kodu – i nie ma tu miejsca na pominięcie któregoś z wystąpień.

Jeżeli któryś literał w kodzie **Fischer chess** spełniał definicję magicznej liczby, został zastąpiony odpowiednią stałą (wobec specjalnych własności języka Java, często była to zmienna okraszona modyfikatorami **static final**).

Pośród wielu innych przykładów poświadczających powyższe słowa, prezentujemy listing klasy **Square.java**. Zmiana współrzędnych na odpowiadające im znaki drukowalne wymusza każdorazowe dodanie do nich liczby 97, której wystąpienie bezpośrednie mogłoby wywołać niepotrzebne zamieszanie. Odpowiedź Projektu mieści się w użyciu zmiennej **charConstant** o odpowiedniej wartości.

(Naturalnie, opis punktu 5. nie ma zamiaru sugerować Czytelnikowi, że w kodzie nie pojawiają się nigdzie literały numeryczne. Projekt broni wszakże stanowiska, że ich użycie w tych przypadkach jest usprawiedliwione.)

6. Zasada DRY

DRY Principle, czyli zasada *Don't repeat yourself*, stanowi, że programiści winni zastępować wielokrotne wystąpienie tożsamych fragmentów kodu użyciem pętli i wywołań funkcji. **Fischer chess** postępował w myśl tej zasady (z drobnymi wyjątkami, w których niewielkie powtórzenia wynikały z odpowiednich przyczyn).

Flagowym przykładem zasady DRY jest zawartość StageManager.java, odpowiadającej za zmianę pojawiających się okien w odpowiedzi na działania Użytkownika. Zamiast wielokrotnie pisać ten sam kod w miejscach, gdzie jedno okno może być skutkiem inputu pojawiającego się w różnych stadiach działania aplikacji, klasa StageManager korzysta z zestawu metod tworzących właściwe okna. Wystarczy wywołać metodę, by zaspokoić zapotrzebowanie na pojawienie się okna – powtarzają się wywołania funkcji, nie zaś – i dobrze! – ich zawartość.

```
public Scene createEndingScene(Result result, GameSupervisor gameSupervisor) {...}

public Scene createGame(boolean Fischer, boolean gameFromFile){...}

public void MainMenu(){...}

public void GameStartMenu(){...}

public void Statistic(){...}

public void readFromFileScene(){...}

public void OpenMainMenu(){
    primaryStage.setScene(MainMenu);
    primaryStage.show();
}

public void OpenEndingScene(Result result, GameSupervisor gameSupervisor){
    primaryStage.setScene(createEndingScene(result, gameSupervisor));
}
```

7. Enumeratory

Działanie programu łączy się z używaniem **flag** w różnych formach – zatem wartości, które w jakiś sposób determinują znaczenie bądź sposób obsługi występujących sytuacji.

Złą praktyką jest jednak stosowanie sobie tylko wiadomych wartości liczbowych, w zależności od których podejmowane są decyzje – cóż bowiem znaczy dla Osoby postronnej wybór sterowania na podstawie rozróżnienia pomiędzy wartością parametru koloru wynoszącą 0, 1 i 2? O niebo lepiej użyć enumeratorów, przykładowo: Colour.Black, Colour.Yellow i Colour.Grey, odpowiednio.

```
if (result.type == MoveType.KILL) {
    Army whichArmy = (onMove.getPieceColour() == PieceColour.WHITE) ? darkAlivePieces : whiteAlivePieces;
    secondMoved = new Note(board [newX] [newY].getPiece(), newX, newY, graveyard.getX(), graveyard.getY());
    board [newX] [newY].getPiece().controlCastlePossibilities();
    whichArmy.kill(board [newX] [newY].getPiece());
    Army whichArmy = (onMove.getPieceColour() == PieceColour.WHITE) ? darkAlivePieces : white
} el
    Army whichArmy = (onMove.getPieceColour() == PieceColour.WHITE) ? darkAlivePieces : whiteAlivePieces;
    int yOfVictim = (onMove.getPieceColour() == PieceColour.WHITE) ? newY + 1 : newY - 1;
    secondMoved = new Note(board [newX] [yOfVictim].getPiece(), newX, newY, graveyard.getX(), graveyard.getY());
    board [newX] [yOfVictim].getPiece().controlCastlePossibilities();
    whichArmy.kill(board [newX] [yOfVictim].getPiece());
} else if (result.type == MoveType.SIMPLE_PROMOTION) {
```

Gra w szachy w naturalny sposób wskazuje, w którym momencie enumeratory mogłyby być przydatne. Projekt używa ich w wielu przypadkach, każdorazowo w sposób pomocny dla piszących kod i jasny dla tych, którzy go odczytują (poniżej klasy enumeratorów wraz z jednym z przypadków użycia).

```
public enum TimeControl {
    NONE( s: "No control"), THREE( s: "3 + 0"), FIVE( s: "5 + 0"), FIFTEEN( s: "15 + 0"), THREE_PLUS_TWO( s: "3 + 2");
    String name;
    TimeControl(String s) {
        name = s;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

```
public enum Result {
    WHITE, BLACK, DRAW;
}
```

```
public enum MoveType {
    SIMPLE, KILL, CASTLE_QUEENSIDE, CASTLE_KINGSIDE, EN_PASSANT, PAWN_DOUBLE_MOVE, SIMPLE_PROMOTION, KILL_PROMOTION;
}
```

```
public enum PieceKind {
    KING( name: "king"), QUEEN( name: "queen"), ROOK( name: "rook"), BISHOP( name: "bishop"), KNIGHT( name: "knight"), PAWN( name: "pawn");


    String name;

    PieceKind(String name) { this.name = name; }

    public String getName() { return name; }
}
```

8. Zasada KISS

Projekt **Fischer chess** nie komplikuje kodu, gdy jest to niepotrzebne. Przykładowo, w klasie **ExCoordinates.java** odpowiedzialnej za wyznaczanie pozycji początkowej figur, w jej części dotyczącej szachów standardowych, każdy amator szachów wie, co oznacza listing (jeśli dodatkowo zna język angielski!):

```
} else {
    t [0] = PieceKind.ROOK;
    t [1] = PieceKind.KNIGHT;
    t [2] = PieceKind.BISHOP;
    t [3] = PieceKind.QUEEN;
    t [4] = PieceKind.KING;
    t [5] = PieceKind.BISHOP;
    t [6] = PieceKind.KNIGHT;
    t [7] = PieceKind.ROOK;
}
```

9. Sekwencje Try-Catch

Używanie funkcji rzucających wyjątki (w tym funkcji z pakietu standardowego, które na mocy dokumentacji takie wyjątki rzucają) wiąże się z koniecznością zapewnienia stabilności wykonania aplikacji nawet w skrajnych – awaryjnych – przypadkach.

```
public void readMoves(String filePath){
    try(FileReader reader = new FileReader(filePath)){
        int sup = supervisor.counter;
        Scanner sc = new Scanner(reader);
        for(int i = 0; sc.hasNext(); i++){
            String line = sc.next();
            if(i%3 != 0) {
                resolveMove(line);
                if(sup == supervisor.counter){
                    System.out.println("ERROR");
                }
            }
            sup = supervisor.counter;
        }
        supervisor.counter--;
    }catch(IOException err){
        System.err.format("IOException: %s%n", err);
    }
}
```

W wielu językach programowania, w tym w języku **Java**, do obsługi wyjątków służy instrukcja try-catch. Dzięki niej, nawet pojawiające się działanie funkcji niezgodne z oczekiwaniem nie kończy się katastrofą widoczną dla użytkownika – obsługa sytuacji kryzysowej pozostaje w gestii osoby implementującej kod. Projekt pragnie zaprezentować instrukcję try-catch w klasie **GameFile.java**, w momencie odpowiedzialnym za otwieranie pliku.

10. SOLID

Dyrektywy wchodzące w skład SOLID niekiedy wzajemnie się wykluczają. Projekt pragnie zaprezentować te ich części, które ziścił w czasie implementacji.

a) S is for **Single Responsibility Principle**

Pojęcie pojedynczej odpowiedzialności klasy bywa płynne. Zgodnie z informacją na temat klas, zawartą w *Jakości projektu architektury*, klasy Projektu mają jasno określone role, które realizują (więcej informacji we wskazanym dokumencie wchodzącym w skład dokumentacji).

b) O is for **Open/Closed Principle**

Istotna część zachowań klas opisana jest w stosownych metodach, nie zaś w niejasnych fragmentach kodu. Modyfikacja zachowania może być więc wykonana zdalnie, bez zmiany samej klasy. Nie dotyczy to oczywiście wszystkich klas ani wszystkich ich części, jednak w wybranych, a ważnych przypadkach niniejsza zasada jest przez Projekt respektowana.

(L, I, D – nie dotyczy)