JAKOŚĆ PROJEKTU ARCHITEKTURY

1. Wprowadzenie

Implementacja kodu powinna być realizacją planu – projektu, w myśl którego należy rozwijać aplikację, wzbogacając ją o kolejne funkcjonalności. W ten sposób programiści wiedzą, jaką drogą mają podążać (zatem tą, którą wytycza projekt), by osiągnąć istniejący w ich myśli zbiorowej produkt ostateczny – gotowy na podbój rynku dystrybucji.

Dobrze, gdy projekt posiada pewne cechy, niemal domyślnie zapewniające zwiększoną jakość całego oprogramowania. Dzieje się tak, gdyż już kilka pokoleń programistów, na kanwie wspólnych i spersonalizowanych doświadczeń, bolesnych porażek i zaskakujących sukcesów wypracowało uniwersalne rozwiązania, sprawdzające się w mnogich sytuacjach. Projekt **Fischer chess** pragnie zaprezentować wybór z tych praktyk, które stały się udziałem planu, zaraz po wstępnym opisie zaprojektowanych klas.

2. Opis zadań najważniejszych klas

StageManager – odpowiedzialna za to, co faktycznie widzi użytkownik. Metody MainMenu(), GameStartMenu() oraz Statistics(), a także grupa metod pomocniczych, wywoływane w odpowiedniej kolejności tworzą scenografię żądaną przez użytkownika. Analogia teatralna, jak i nazwa klasy nie jest przypadkowa – otóż w frameworku JavaFX podstawą wszystkich widocznych interfejsów jest obiekt typu Stage (a więc scena – fizyczna platforma), na który nakrywane są odpowiednio skomponowane obiekty typu Scene (czyli moment przedstawienia).

Main – właściwe serce całej aplikacji – tutaj zbierają się wątki ze wszystkich pozostałych klas, dodawane są rzeczywiste figury na planszę, kontrolowany i wykonywany jest ruch, a także pomiar czasu (dokonywany dzięki obiektom klasy Clock, czyli klasy wewnętrznej z pliku Main.java); w końcu – sprawdzany jest koniec partii na podstawie sytuacji na szachownicy, na zegarze lub na przyciskach.

Army – w rozgrywce wykorzystywane są każdorazowo dwie instancje tej klasy, po jednej dla każdej ze stron. Klasa Army, zgodnie z nazwą, stanowi kontener grupujący wszystkie figury białych lub czarnych (dodawane do armii metodą hire()). Dzięki zebraniu w jednym miejscu wszystkich bierek danego koloru, w sposób wygodny i zautomatyzowany można wygenerować na początku każdej tury wszystkie dozwolone ruchy wszystkich figur (makeThemReady()), a co za tym idzie – sprawdzić, czy ruszający ma jeszcze jakiekolwiek ruchy (doTheyHaveAnyMoves()), czyli wykryć konieczność zakończenia rozgrywki. Metoda lookForChecks() używana jest do symulowania ruchów, tyle że przez przeciwną armię, których poprawność aplikacja sprawdza – ruch

ma szansę być poprawnym, jeśli po nim żadna z figur przeciwnika nie szachowałaby króla strony będącej na ruchu.

ExCoordinates – obiekt tej klasy jest używany raz na każdą rozgrywkę – ustala on pozycję startową bierek obu stron. Naturalnie, chodzi jedynie o ustawienie figur (pionki zawsze startują z tych samych pozycji) i to tylko w szachach Roberta Jamesa Fischera, gdyż w wariancie klasycznym ustawienie figur pozostaje niezmienne. Algorytm wykorzystany przy wyborze pozycji początkowej szachów Fischera został opisany w punkcie czwartym. Nazwa klasy (Ex-) nawiązuje do faktu, że jedna ze współrzędnych (dla ustalenia uwagi niech będzie to y) nie wymaga nigdy ustaleń – białe stoją na jednej ze skrajnych wartości współrzędnej *pionowej*, czarne – na drugiej skrajnej.

GameFile – klasa realizująca funkcjonalność wczytania partii z formatu PGN do aplikacji – metoda readMoves() czyta ruch z pliku, by następnie metoda resolveMove(), przy pomocy funkcji pomocniczych, przekształciła go w pożądany opis, zrozumiały dla aplikacji i przekształcalny w faktyczny ruch odgrywany na szachownicy – do czego służą (w zależności od skomplikowania związanych z typem ruchu działań), dedykowane metody movePiece(), movePawn(), moveKing(), jak i castle().

GameSupervisor – nadzorca rozgrywki oraz jej stadium, dokumentujący wszystkie dokonane dotychczas ruchy, przesyłane do niego metodą add(). Adnotacje od razu przekształcane są również do formatu pgn i grupowane w polu pgnNotation, by po zakończeniu partii zostać zapisane razem do stosownego pliku. GameSupervisor udostępnia również narzędzia do przeglądania partii do przodu i do tyłu (odpowiednio: repeatMove() i retraceMove()), wywoływane przez naciśnięcie stosownych guzików.

Square - prosta klasa, posiadająca dwa pola i funkcje zezwalające na dostęp do owych pól i ich modyfikację. Pola reprezentują współrzędne, stanowiące podstawę lokalizacji i ruchu figur w każdej grze planszowej.

Move - klasa rozszerzająca klasę Square, wzbogacona o enumerator typu MoveType; wykorzystywana do zapisu i przesyłania dokonywanych ruchów, a więc: współrzędnych docelowych figury i rodzaju ruchu, który się odbywa (do rodzaju ruchu w tym rozumieniu należy przykładowo roszada krótka, roszada długa, bicie w przelocie, ruch zwykły itd.). Sposób wykorzystania tej klasy jest zgodny z intuicją.

Note – klasa służąca do dokumentacji całego ruchu konkretnej figury. Z uwagi na specyfikę zapisu i konieczność umieszczenia w pamięci poprzedniej (prawdziwej przed ruchem) pozycji danej figury, klasa Note posiada dwa zestawy współrzędnych rodzaju Square; kompletność zestawu dopełnia referencja do figury, która wykonała ruch.

Piece – instancje tej klasy należy rozumieć jako poszczególne bierki. Projekt nie zdecydował się na zastosowanie schematu dziedziczenia, z uwagi na potencjalne niepożądane komplikacje, wynikające chociażby z promocji pionków (zmieniających

wówczas tożsamość z bycia pionkiem na bycie figurą). Zamiast tego do identyfikacji rodzaju bierki wykorzystuje się enumerator PieceKind. Klasa zawiera wszelkie informacje o figurach, ich obecną lokalizację, kolor, historię promocji, a przede wszystkim mechanizm wyznaczania poprawnych ruchów, wykorzystywany na większą skalę przez klasę Army. Ruch poprawny to ruch z efektem końcowym w obrębie szachownicy, nienaruszający zasad ruchu rodzaju figury, wykonany w czasie tury przypadającej stronie reprezentowanej przez daną figurę oraz o tej własności, że po nim król figury nie znajdzie się (nie pozostanie) w stanie szacha. Wszystkie te cechy (prócz trzeciej, sprawdzanej w klasie Main przy pomocy TurnIndicator) są testowane właśnie w klasie Piece.

TurnIndicator – zgodnie z nazwą przede wszystkim wskazuje, która strona z grających jest na ruchu lub, ogólniej, czy jest czyjś ruch, czy też partia jest przeglądana wstecz (tj. nie powinien zostać dopuszczony żaden nowy ruch, dopóki partia nie zostanie przewinięta do ostatniego wykonanego ruchu). Prócz tego klasa dysponuje zestawem referencji do konkretnych figur, pomocnych przy ustalaniu możliwości wykonania roszady w stosownej kolejce.

Tile – klasa reprezentująca konkretne pola szachownicy, posiada pole determinujące kolor, narzędzie służące do podświetlania pól (highlight()), a przede wszystkim – referencję do figury, która stoi na polu (w przypadku braku figury – null) oraz narzędzie pozwalające obiektom z zewnątrz weryfikację tej zajętości, zatem metodę zwracającą typ logiczny boolean: hasPiece().

MoveType – enumerator oznaczający rodzaj wykonywanego ruchu, często uzależniający dalsze wykonanie aplikacji:

a) PAWN_DOUBLE_MOVE – ruch pionkiem o dwa pola
b) CASTLE_QUEENSIDE – roszada długa
c) CASTLE_KINGSIDE – roszada krótka
d) EN_PASSANT – bicie w przelocie
e) KILL_PROMOTION – promocja pionka w wyniku zabicia figury
f) SIMPLE_PROMOTION – promocja pionka bez bicia
g) KILL – inne bicia niż wymienione dotychczas
h) SIMPLE – pozostałe ruchy

Wydawać by się mogło, że niektóre rozróżnienia nie są tutaj konieczne. Każde z nich jednak w istotny sposób różni się od pozostałych – najczęściej chodzi o ilość figur powiązanych z ruchem (również bitych).

PieceColour – enumerator oznaczający kolor bierki bądź kolor innego dychotomicznego zjawiska, np. strony obecnie wykonującej ruch.

a) BLACK – określa właściwości czarnych figur i gracza dysponującego armią czarnych
b) WHITE – określa właściwości białych figur i gracza dysponującego armią białych

PieceKind – enumerator określający rodzaj figury, efektywnie zastępujący alternatywne rozwiązanie, to jest: zastosowanie dziedziczenia.

a) KING – król
b) QUEEN – hetman
c) ROOK – wieża
d) KNIGHT – skoczek
e) BISHOP – goniec
f) PAWN – pionek

Result – opisuje wynik zakończonej partii**.**

a) WHITE – wygrały białe
b) BLACK – wygrały czarne
c) DRAW – został ustalony (bądź wymuszony patem) remis

TimeControl – opisuje tryb rozgrywki, w kategorii pomiaru czasu.

a) NONE – brak pomiaru czasu
b) THREE – gracz ma trzy minuty na wszystkie ruchy
c) FIVE – gracz ma pięć minut na wszystkie ruchy
d) FIFTEEN – gracz ma piętnaście minut na wszystkie ruchy
e) THREE_PLUS_TWO – gracz ma trzy minuty na wszystkie ruchy, dodatkowo otrzymując bonifikatę dwóch sekund za każdy wykonany ruch

Niektóre enumeratory posiadają automatyczną konwersję na napis (toString()), przydatną przy konkretnych zastosowaniach frontendowych).

3. Dziedziczenie

Widniejące w projekcie schematy dziedziczenia należy podzielić na **dwie** kategorie.

Pierwszą z nich stanowią obiekty, które dziedziczą po klasach zaimplementowanych w **standardzie** Javy. Niniejszy fragment dokumentacji wspomina jednakże o tych przykładach, gdyż wybór super-klas (klas nadrzędnych) nie był dziełem przypadku, lecz odpowiadał dokładnym potrzebom określonym w początkowych etapach tworzenia planu.

```
public class Piece extends StackPane {
```

Klasa **Piece** rozszerza klasę javową **StackPane**, z uwagi na jej właściwości w hierarchii widoczności na scenie frameworka JavaFX. Jeszcze bardziej oczywistym wyborem jest klasa, którą rozszerza **Tile.java** – mowa oczywiście o pochodzącej z grupy scene.shape klasie Rectangle – Czytelnik dowiedziawszy się, że plik Tile reprezentuje poszczególne pola na szachownicy zgodzi się z pewnością z Projektem, jeżeli idzie o właśnie omawianą decyzję.

```
public class Tile extends Rectangle {
```

Przykład, należący do odmiennej kategorii, który należy bezwzględnie opisać to relacja zachodząca pomiędzy klasami **Move** oraz **Square**. Pierwsza powstała ta druga, odpowiadająca na wielokrotnie występujące zapotrzebowanie na łączne przesyłanie dwóch współrzędnych (określających koordynaty właściwych pól na szachownicy.

Nierzadko wszelako owe współrzędne powinny być przekazywane pomiędzy obiektami wraz z informacją dynamiczną będącą deskrypcją typu ruchu pewnej figury (przez typ rozumiemy tu bicie, bicie w przelocie, którąś z roszad – wszelkie typy specyfikowane w enumeratorze MoveType), który to typ determinował działanie obiektu lub metody utrzymującej informacje o współrzędnych. Ponadto często wywoływane *getery* i *setery* dotyczące iksów i igreków z klasy Square przydałyby się i w drugiej klasie, której powstanie niniejszym się uzasadnia. Naturalną odpowiedzią na zaistniałą sytuację stało się zaprojektowanie klasy Move, która dziedziczyła po klasie Square wszystkie jej funkcjonalności, samemu dodając te, dotyczące dodatkowych informacji związanych właśnie z ruchem. Dowodem na optymalność przyjętego rozwiązania jest ilość miejsc w kodzie, w których znajdują się i między którymi są przesyłane instancje wyżej wymienionej klasy.

4. Zaimplementowane algorytmy

Uwadze Czytelnika Projekt poleca algorytm generowania poprawnej i losowej pozycji szachów Fischera, zaimplementowany w konstruktorze klasy ExCoordinates. Algorytm z równym prawdopodobieństwem zwraca każdą z poprawnych pozycji startowych, czyli takich, gdzie (patrząc z perspektywy jednej z grających stron, gdyż druga otrzymuje bliźniacze ustawienie):

a) gońce są różnopolowe (stoją na polach różnych kolorów)
b) król znajduje się gdzieś pomiędzy dwoma wieżami

Algorytm został zaproponowany przez p. Ingo Althöfera (1998). Wykorzystuje jedną kostkę do gry (sześcienną); Projekt prezentuje równoważną wersję, korzystającą z możliwości otrzymywania liczb (pseudo)losowych z dowolnego zakresu.

1. Wylosuj liczbę od 1 do 4 – na tym białym polu, licząc od lewej, postaw gońca białopolowego
2. Wylosuj liczbę od 1 do 4 – na tym białym polu, licząc od lewej, postaw gońca czarnopolowego
3. Wylosuj liczbę od 1 do 6 – na tym polu, licząc od lewej (spośród pól dotychczas wolnych), postaw hetmana
4. Wylosuj liczbę od 1 do 5 – na tym polu, licząc od lewej (spośród pól dotychczas wolnych), postaw pierwszego skoczka
5. Wylosuj liczbę od 1 do 5 – na tym polu, licząc od lewej (spośród pól dotychczas wolnych), postaw drugiego skoczka
6. Na pierwszym polu wolnym od lewej postaw pierwszą z wież
7. Na pierwszym polu wolnym od prawej postaw drugą z wież
8. Na ostatnim wolnym polu umieść króla

Poprawność otrzymanej pozycji, jak i jej losowość, stanowi natychmiastową obserwację.

5. Wykorzystane wzorce projektowe

Stan – instancje klasy Piece zmieniają swoje zachowanie w zależności od stanu, w którym się znajdują, jako że pionek w momencie promocji zmienia jeden ze swoich wewnętrznych parametrów. Od tego czasu metoda findPossibleMoves() działa w istotny sposób inaczej niż uprzednio, co jest wynikiem zmiany stanu konkretnego obiektu. Listing prezentuje sposób zmiany stanu, determinującej rodzaj wywołań wymienionej powyżej funkcji.

```
public void promote(PieceKind kind) {
    this.kind = kind;
    String src = getImageSource(colour, kind,  mode: 0);
    back.setFill(new ImagePattern(new Image(src)));
    kindAfterPromotion = kind;
}
```

Obserwator – metoda doPromotionButtonAction() kończy sekwencję metod wykonujących promocję pionka; jako ostatnia w kolejności, powiadamia ona obiekt klasy TurnIndicator (w tym przypadku o nazwie onMove), że należy zmienić turę rozgrywki i pozwolić następnemu graczowi na dokonanie ruchu.

```
public void doPromotionButtonAction(PieceKind buttonKind, Piece movingPiece, Stage stage, Pair<Note, Note> annotation, int pieceDuplication, Move result)
    right.getChildren().remove(promotionBox);
    movingPiece.setKindAfterPromotion(buttonKind);
    gameSupervisor.add(annotation, pieceDuplication, result);
    movingPiece.promote(buttonKind);
    movingPiece.repaint();
    onMove.switchMode();
    postMoveAction(stage);
}
```

Fasada – zgrupowanie wielu figur w jednej instancji klasy Army umożliwia sprawne gospodarowanie nimi, pomijające na tym poziomie abstrakcji liczne zawiłości i różnice występujące między nimi. Kluczowe znaczenie ma zastosowanie kontenera ArrayList jako pola prywatnego i mechanizmu iterowania się po elementach tak utworzonej kolekcji. Więcej o tym wzorcu w opisie klasy Army

```
public void makeThemReady(int enPassantXPossibility) {
    for (Piece piece : pieces) {
        piece.findPossibleMoves(enPassantXPossibility, checkMode: false, checkFlag: false);
    }
}

public boolean doTheyHaveAnyMoves() {
    for (Piece piece : pieces) {
        if (piece.getPossibleMoves().size() > 0) {
            return true;
        }
    }
    return false;
}

public boolean lookForChecks(int enPassantXPossibility, Square enemyKingCoords) {
    for (Piece piece : pieces) {
        piece.findPossibleMoves(enPassantXPossibility, checkMode: true, checkFlag: false);
    }

    for (Piece piece : pieces) {
        if (!piece.getIsTargeted()) {
            for (Move move : piece.getPossibleMoves()) {
                if (move.areTheyTheSame(enemyKingCoords)) {
                    return true;
                }
            }
        }
    }
    return false;
```