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.

2. Opis zadań najważniejszych klas

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.

Niejednokrotnie 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

5. Wykorzystane wzorce projektowe