Dokumentacja użytkownika

Rozproszona platforma do tworzenia gier planszowych, przeznaczona na system Android.

*Niniejsze opracowanie powstało w trakcie i jako rezultat za­jęć dy­dak­tycz­nych z przedmiotu wymienionego na stronie tytułowej, pro­wa­dzo­nych w Akademii Górniczo-Hutniczej w Krakowie (AGH) przez oso­bę (oso­by) wy­mie­nioną (wymienione) po słowach "Pro­wa­dzą­cy zajęcia" i nie może być wy­korzystywane w jakikolwiek sposób i do jakichkolwiek ce­lów, w ca­ło­ści lub części, w szczególności pub­li­ko­wa­ne w ja­ki­kol­wiek spo­sób i w jakiejkolwiek formie, bez uzy­ska­nia uprzed­niej, pi­sem­nej zgody tej oso­by (tych osób) lub odpowiednich władz AGH.****Copyright © 2013 Akademia Górniczo-Hutnicza (AGH) w Krakowie***

**Spis treści**

1. Zakres dokumentu 3

2. Przykładowy przebieg tworzenia gry na przykładzie Monopoly 4

2.1. Założenia 4

2.2. Implementowanie logiki gry 5

2.3. Rozpoczynamy pisanie klasy: 7

2.4. Implementacja inicjalizacji gry 9

2.5. Implementacja akcji związanych ze zmianą stanu ze względu na kontekst. 13

2.6. Implementacja szczegółów dotyczących gry 15

2.7. Połączenie modułów w obiekt tworzący grę 16

2.8. Rozpoczęcie gry 17

3. Podsumowanie 19

4. Bibliografia 20

Indeks ważniejszych terminów 21

Spis ilustracji i tabel 21

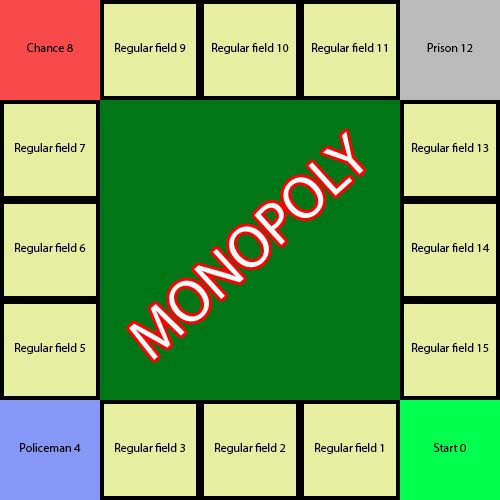
# Zakres dokumentu

Dokument ten ma na celu zaprezentowanie przykładowej implementacji gry Monopoly z wykorzystaniem platformy do gier planszowych.

# Przykładowy przebieg tworzenia gry na przykładzie Monopoly

## Założenia

Nasza implementacja prostej gry monopoly opiera się na następującej planszy:



Plansza składa się z 16 pól , w tym 4 specjalnych. Start jest polem startowym, Policeman powoduje przejście do więzienia – Prison pole 12. Poza tym jest pole szansy na polu ósmym które powoduje obecnie ,że gracz dostanie 100 $. Pola zwykłe kosztują 100$ . Można na nich dokupić domek za 100$ , a gracz który stanie na polu należącym do przeciwnika musi zapłacić 100$. Przy przejściu przez start gracz dostaje 100$. Początkowo gracze mają po 1000$. Gra jest pisana z myślą o rozgrywce dla dwóch osób.

## Implementowanie logiki gry

Stworzenie gry rozpoczynamy od napisania jej logiki. Polega ona na implementacji stanów z których występują przejścia do innych stanów i wykonania w nich odpowiednich akcji. Jest to implementacja wzorca projektowego Stan. Interfejs stanu abstrakcyjnego wygląda tak:

package boardGamePlatform**.**game**;**

**import** java**.**util**.**List**;**

**import** java**.**util**.**Map**;**

**import** boardGamePlatform**.**gameResources**.**Board**;**

public abstract class State **{**

protected Board board**;**

protected Player currentPlayer**;**

protected List**<**Player**>** allPlayers**;**

protected StateChangable stateChangable**;**

protected Map**<**String**,** String**>** attributes**;**

public State**(**Board board**,**Player currentPlayer**,**List**<**Player**>** allPlayers**,**StateChangable stateChangable**)** **{**

**…**

**}**

public State**(**Board board**,**Player currentPlayer**,**List**<**Player**>** allPlayers**,**StateChangable stateChangable**,** Map**<**String**,** String**>** attributes**)** **{**

**…**

**}**

public Player getCurrentPlayer**()** **{**

**return** currentPlayer**;**

**}**

public List**<**Player**>** getAllPlayers**()** **{**

**return** allPlayers**;**

**}**

public abstract Context getContext**();**

public abstract void makeTurn**(**TurnResult currTurnResult**,**Map**<**Player**,**TurnResult**>** otherTurnResults**);**

public abstract boolean gameEnded**();**

**}**

Z tego wynika ,że klasa dziedzicząca po stanie musi:

* Posiadać klasę przyjmujący planszę, obiekt logiki którego stan się zmienia po stanie i ewentualnie dodatkowe atrybuty przyjmowane przy tworzeniu stanu.
* Zaimplementować funkcję zwracającą obecny kontekst(getContext) który zostanie przekazany do obecnego gracza i do pozostałych graczy.
* Zaimplementować funkcję dokonującą tury w stanie ( makeTurn) na podstawie rezultatów tur gracza obecnego i pozostałych graczy biorących udział w rozgrywce.
* Zaimplementować funkcję sprawdzającą czy gra dobiegła końca( gameEnded).

Początkowym stanem gry będzie StartGame. W nim poinformujemy wszystkich graczy ,że gra się rozpoczęła i przejdziemy do stanu rozpoczynającego turę gracza.

Importujemy potrzebne klasy:

**import** java**.**util**.**List**;**

**import** java**.**util**.**Map**;**

**import** boardGamePlatform**.**game**.**Context**;**

**import** boardGamePlatform**.**game**.**Player**;**

**import** boardGamePlatform**.**game**.**State**;**

**import** boardGamePlatform**.**game**.**StateChangable**;**

**import** boardGamePlatform**.**game**.**TurnResult**;**

**import** boardGamePlatform**.**gameResources**.**Board**;**

Tworzymy nowy stan poprzez nadpisanie klasy bazowej Start i robimy konstruktor który przekaże parametry do konstruktora klasy bazowej:

public class StartGame **extends** State**{**

public StartGame**(**Board board**,** Player currentPlayer**,**

List**<**Player**>** allPlayers**,** StateChangable stateChangable**)** **{**

**super(**board**,** currentPlayer**,** allPlayers**,** stateChangable**);**

**}**

Nadpisujemy funkcję zwracającą kontekst:

public Context getContext**()** **{**

**return** **new** Context**(**"StartGame"**,** board**,** currentPlayer**,** allPlayers**);**

**}**

W kontekście przekazujemy nazwę stanu , planszę , obecnego gracza i wszystkich graczy, te informacje wystarczą nam u graczy by odbyć na ich podstawie w tym stanie odpowiednie instrukcje.

Następną rzeczą jest implementacja logiki tury który odbędzie się na podstawie działań graczy w turze.

public void makeTurn**(**TurnResult currTurnResult**,**

Map**<**Player**,** TurnResult**>** otherTurnResults**)** **{**

stateChangable**.**changeState**(new** PlayerMove**(**board**,** currentPlayer**,** allPlayers**,** stateChangable**));**

**}**

Tura na początku gry nie jest zbyt skomplikowana, obejmuje tylko przejście do nowego stanu w którym obecny gracz będzie dokonywał ruchu . Jest to realizowane poprzez wykonanie na obiekcie logiki gry stateChangable funkcji zmiany stanu changeState do odpowiedniego stanu.

Kończymy stan StartGame implementując funkcję zwracającą czy gra została zakończona i domykamy nawias klamrowy z początku klasy.

public boolean gameEnded**()** **{**

**return** **false;**

**}**

**}**

Nasz pierwszy stan gry został ukończony. Kontynuacją jest napisanie stanu PlayerMove implementującego pobranie ruchu od gracza i dokonanie odpowiedniej akcji w zależności od wyrzutu kostką. Importujemy potrzebne klasy:

**import** java**.**util**.**List**;**

**import** java**.**util**.**Map**;**

**import** boardGamePlatform**.**game**.**Context**;**

**import** boardGamePlatform**.**game**.**Player**;**

**import** boardGamePlatform**.**game**.**State**;**

**import** boardGamePlatform**.**game**.**StateChangable**;**

**import** boardGamePlatform**.**game**.**TurnResult**;**

**import** boardGamePlatform**.**gameResources**.**Board**;**

**import** boardGamePlatform**.**gameResources**.**Field**;**

**import** boardGamePlatform**.**platformExceptions**.**FieldNotHasNeihgboursException**;**

**import** boardGamePlatform**.**platformExceptions**.**NotAvalaibleMoveException**;**

**import** boardGamePlatform**.**platformExceptions**.**NotPawnFoundException**;**

**import** boardGamePlatform**.**platformExceptions**.**PlayerNotHasPawnsException**;**

**import** boardGamePlatform**.**platformExceptions**.**UnsupportedMoveNumberException**;**

**import** boardGamePlatform**.**utilities**.**BoardHelper**;**

## Rozpoczynamy pisanie klasy:

public class PlayerMove **extends** State **{**

public PlayerMove**(**Board board**,** Player currentPlayer**,**

List**<**Player**>** allPlayers**,** StateChangable stateChangable**)** **{**

**super(**board**,** currentPlayer**,** allPlayers**,** stateChangable**);**

**}**

Implementujemy funkcję pobierającą kontekst:

public Context getContext**()** **{**

**return** **new** Context**(**"PlayerMove"**,** board**,** currentPlayer**,** allPlayers**);**

**}**

I rozpoczynamy pisanie logiki pobierania tur w stanie:

public void makeTurn**(**TurnResult currTurnResult**,**

Map**<**Player**,** TurnResult**>** otherTurnResults**)** **{**

int moveLen **=** Integer**.**parseInt**(**currTurnResult**.**getProperty**(**"Move"**));**

BoardHelper boardH **=** **new** BoardHelper**(**board**);**

W naszym rozwiązaniu efektem realizacji tury u obecnego gracza będzie w TurnResult w atrybucie Move ilości wyrzuconych oczek. Wydobywamy z currTurnResult atrybut Move , przypisujemy do zmiennej moveLen . Inicjalizujemy BoardHelper który zawiera funkcję które pomogą nam przenosić pionki po planszy i rozpoczynamy działanie tury:

**try** **{**

Field startField **=** boardH**.**getPawnsField**(**currentPlayer**.**getPlayerFirstPawn**());**

Field endField **=** boardH**.**movePlayerFirstPawnByNum**(**currentPlayer**,** moveLen**);**

Do pola startField przypisujemy początkowe pole gracza, a do endField pole na którym powinien się znaleźć. W obu przypadkach używamy odpowiednich funkcji z boardHelpera( po dalsze informację na ich temat polecam przejrzenie dokumentacji JavaDoc).

**if(**startField**.**getID**()** **>** endField**.**getID**())** **{**

stateChangable**.**changeState**(new** CrossingStartField**(**board**,** currentPlayer**,** allPlayers**,** stateChangable **));**

**return;**

**}**

Sprawdzamy czy pionek gracza przeszedł przez pole startowe, jeżeli tak to zmieniamy odpowiedni stan i wychodzimy z funkcji.

**switch(**endField**.**getID**())** **{**

**case** 0 **:** stateChangable**.**changeState**(new** StartField**(**board**,** currentPlayer**,** allPlayers**,** stateChangable**));**

**return;**

**case** 4 **:** stateChangable**.**changeState**(new** Policeman**(**board**,** currentPlayer**,** allPlayers**,** stateChangable**));**

**return;**

**case** 8 **:** stateChangable**.**changeState**(new** ChanceField**(**board**,** currentPlayer**,** allPlayers**,** stateChangable**));**

**return;**

**case** 12 **:** stateChangable**.**changeState**(new** Prison**(**board**,** currentPlayer**,** allPlayers**,** stateChangable**));**

**return;**

**}**

Sprawdzamy na jakie pole stanął gracz i zależności od jego ID zmieniamy stan na odpowiadający jemu jeżeli stanął na jedno z pól specjalnych ( pole start, pole policjant, pole szansy, pole więzienie). Jeżeli to nie jest pole specjalne :

int ownerID **=** Integer**.**parseInt**(**endField**.**getAttribute**(**"Owner"**));**

**if(** ownerID **==** currentPlayer**.**getPlayerID**())** **{**

stateChangable**.**changeState**(new** AtOwnerField**(**board**,** currentPlayer**,** allPlayers**,** stateChangable**));**

**return;**

**}** **else** **if(**ownerID **==** 0**)** **{**

stateChangable**.**changeState**(new** AtNobodysField**(**board**,** currentPlayer**,** allPlayers**,** stateChangable**));**

**return;**

**}** **else** **{**

stateChangable**.**changeState**(new** AtEnemyField**(**board**,** currentPlayer**,** allPlayers**,** stateChangable**));**

**return;**

**}**

**}** **catch** **(**Exception e**)** **{**

e**.**printStackTrace**();**

**return;**

**}**

**}**

Początkowo sprawdzamy właściciela pola( 0 – bank, 1 – gracz pierwszy, 2 – gracz drugi). W zależności od przypadku zmieniamy stan na odpowiadający.

public boolean gameEnded**()** **{**

**return** **false;**

**}**

**}**

Piszemy funkcję zwracającą czy gra już się skończyła i kończymy klasę. Piszemy analogicznie resztę klas wchodzących w skład logiki gry( cały kod źródłowy jest dołączony do projektu w pakiecie monopolyConsolePrototype).

## Implementacja inicjalizacji gry

Implementacja inicjalizacji rozgrywki polega na nadpisaniu odpowiednich klas i dokonaniu w niej odpowiednich akcji. Rozpoczniemy od implementacji interfejsu inicjalizacji przedmiotów. Wygląda on w ten sposób:

package boardGamePlatform**.**gameInitializers**;**

**import** java**.**util**.**Map**;**

**import** boardGamePlatform**.**gameResources**.**Item**;**

public interface ItemsInitializer **{**

public Map**<**String**,** Item**>** initializeItems**(**int playersCount**);**

**}**

Implementacja initializatora przedmiotów polega na zwróceniu mapy wszystkich przedmiotów używanych w trakcie rozgrywki dla odpowiedniej liczby graczy podanej w parametrze. A oto jego implementacja:

class ItemsInitializerImpl1 **implements** ItemsInitializer**{**

public Map**<**String**,** Item**>** initializeItems**(**int playersCount**)** **{**

**return** **null;**

**}**

**}**

W naszym przykładzie zdecydowaliśmy się na implementację tej funkcjonalności razem z tworzeniem pól i graczy, ale przy większych projektach implementacja tej funkcjonalności wprowadza porządek.

Następną rzeczą którą należy implementować jest inicjalizacja pionków:

package boardGamePlatform**.**gameInitializers**;**

**import** java**.**util**.**Map**;**

**import** boardGamePlatform**.**gameResources**.**Item**;**

**import** boardGamePlatform**.**gameResources**.**Pawn**;**

public interface PawnsInitializer **{**

public Map**<**String**,** Pawn**>** initializePawns**(**int playerCount**,** Map**<**String**,** Item**>** items**);**

**}**

Funkcja implementowana ma zwrócić mapę nazw na pionki i przyjmuję liczbę graczy i przedmioty wytworzone w etapie pierwszym inicjalizacji.

class PawnsInitializerImpl1 **implements** PawnsInitializer**{**

public Map**<**String**,** Pawn**>** initializePawns**(**int playerCount**,**

Map**<**String**,** Item**>** items**)** **{**

Map**<**String**,** Pawn**>** pawns **=** **new** HashMap**<**String**,** Pawn**>();**

pawns**.**put**(**"Player1Pawn"**,** **new** Pawn**(**1**));**

pawns**.**put**(**"Player2Pawn"**,** **new** Pawn**(**2**));**

**return** pawns**;**

**}**

**}**

W naszym przypadku z założenia robimy grę tylko dla dwóch osób więc bez względu na parametr playerCount tworzymy dwa pionki,przypisujemy je do mapy i zwracamy.

Następnie implementujemy interfejs inicjalizacji graczy, wygląda on w ten sposób:

package boardGamePlatform**.**gameInitializers**;**

**import** java**.**util**.**List**;**

**import** java**.**util**.**Map**;**

**import** boardGamePlatform**.**game**.**Player**;**

**import** boardGamePlatform**.**gameResources**.**Item**;**

**import** boardGamePlatform**.**gameResources**.**Pawn**;**

public interface PlayersInitializer **{**

public List**<**Player**>** initializePlayers**(**int playersCount**,** Map**<**String**,** Item**>** items**,** Map**<**String**,** Pawn**>** pawns**);**

**}**

Należy w nim zaimplementować funkcję initializPlayers która przyjmuję liczbę graczy , przedmioty i pionki wytworzone na etapie wcześniejszym i zwraca listę graczy. A oto implementacja:

class PlayersInitializerImpl1 **implements** PlayersInitializer**{**

@Override

public List**<**Player**>** initializePlayers**(**int playersCount**,**

Map**<**String**,** Item**>** items**,** Map**<**String**,** Pawn**>** pawns**)** **{**

List**<**Player**>** players **=** **new** LinkedList**<**Player**>();**

**if(**playersCount **==** 2**){**

Player p1 **=** **new** Player**(**1**);**

Player p2 **=** **new** Player**(**2**);**

p1**.**addPawn**(**pawns**.**get**(**"Player1Pawn"**));**

p2**.**addPawn**(**pawns**.**get**(**"Player2Pawn"**));**

p1**.**setItem**(**"Money"**,** **new** Item**(**0**,** "Amount"**,** "1000"**));**

p2**.**setItem**(**"Money"**,** **new** Item**(**0**,** "Amount"**,** "1000"**));**

players**.**add**(**p1**);**

players**.**add**(**p2**);**

**return** players**;**

**}**

**return** **null;**

**}**

**}**

Tworzymy dwóch graczy przypisując im w konstruktorze ID ( odpowiednio 1,2). Dodajemy graczom pionki i przypisujemy im przedmiot Money. Dodajemy graczy do listy i zwracamy.

Następnie implementujemy interfejs BoardInitializer

package boardGamePlatform**.**gameInitializers**;**

**import** java**.**util**.**Map**;**

**import** boardGamePlatform**.**gameResources**.**Board**;**

**import** boardGamePlatform**.**gameResources**.**Item**;**

**import** boardGamePlatform**.**gameResources**.**Pawn**;**

public interface BoardInitializer **{**

public Board initializeBoard**(**int playersCount**,** Map**<**String**,** Item**>** items**,** Map**<**String**,** Pawn**>** pawns**);**

**}**

Należy w nim zaimplentować initializeboard przyjmującą liczbę graczy, mapę nazw przedmiotów na przedmioty i mapę pionków ( obydwa wytworzone na etapie wcześniejszym).

class BoardInitializerImpl1 **implements** BoardInitializer**{**

@Override

public Board initializeBoard**(**int playersCount**,** Map**<**String**,** Item**>** items**,**

Map**<**String**,** Pawn**>** pawns**)** **{**

Board board **=** **new** Board**();**

BoardBuilder boardBuilder **=** **new** BoardBuilder**(** board **);**

Rozpoczynamy inicjalizator planszy. Tworzymy nową planszę i BoardBuilder którego celem jest pomoc w tworzeniu podstawowego rodzaju plansz.

**try** **{**

boardBuilder**.**makeSimpleLinearBoard**(**16**);**

**}** **catch** **(**UnsupportedBoardSizeException e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

Przy pomocy funkcji makeSimpleLinearBoard tworzymy najprostszy rodzaj planszy z 16 polami gdzie ostatnie pole planszy jest połączone z pierwszym.

**for** **(** Field f **:** board**.**getFields**()** **)** **{**

f**.**addAttribute**(**"Owner"**,** "0"**);**

f**.**addAttribute**(**"Price"**,** "100"**);**

f**.**addItem**(** "House" **,** **new** Item**(**0**,**"Amount"**,**"0"**)** **);**

**}**

Ustawiamy na wszystkich polach planszy cenę i właściciela początkowego( Owner =0 => bank). Dodajmy też możliwość wykupienia na każdym domku.

BoardHelper boardHelper **=** **new** BoardHelper**(**board**);**

**try** **{**

boardHelper**.**placePawnOnField**(**pawns**.**get**(**"Player1Pawn"**),** 0**);**

**}** **catch** **(**NotFieldFoundException e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

**try** **{**

boardHelper**.**placePawnOnField**(**pawns**.**get**(**"Player2Pawn"**),** 0**);**

**}** **catch** **(**NotFieldFoundException e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

**return** board**;**

**}**

**}**

Ustawiamy pionki na planszy i zwracamy wytworzoną planszę.

Po wytworzeniu inicjalizatorów pora na implementację reakcji po stronie graczy na odpowiednie konteksty

## Implementacja akcji związanych ze zmianą stanu ze względu na kontekst.

Implementacja reakcji ze względu na zmianę kontekstu polega na przypisaniu w TurnHandlerze nazw kontekstów na obiekty TurnMakeable które reagują na odpowiednie nazwy kontekstów.Interfejs TurnMakeable wygląda tak:

package boardGamePlatform**.**game**;**

**import** java**.**io**.**Serializable**;**

public interface TurnMakeable **extends** Serializable**{**

public TurnResult makeTurn**(**Context context**);**

**}**

Implementacja odpowiedniej akcji na kontekst polega na implementacji funkcji makeTurn do której jest przekazany obecny kontekst od stanu i zwróceniu wyniku wykonania tury przez gracza w postaci obiektu TurnResult.

Możliwe jest także implementacja w której jest domyślny obiekt implementujący TurnMakeable który zajmuje się zrobieniem akcji dla wszystkich kontekstów które rozróżnia na podstawie ich nazw. Do inicjalizacji turnHandlera z użyciem domyślnego obiektu TurnMakeable należy użyć konstruktora:

public TurnHandler**(**TurnMakeable defaultTurnMaker**)** **{**

…

**}**

Lub przypisać go później używając funkcji z TurnHandler:

public void setTurnMaker**(**String name**,**TurnMakeable turnMakeable**)** **{**

…

**}**

W naszej implementacji monopoly domyślny reaktor na zdarzenia będzie w pełni odpowiedni. Nasz domyślny reaktor na zdarzenia obecnego gracza rozpoczniemy od przypisania do zmiennej :

TurnMakeable currTurnMaker **=** **new** TurnMakeable**()** **{**

@Override

public TurnResult makeTurn**(**Context context**){**

System**.**out**.**println**(**context**.**getCurrPlayer**().**getNick**());**

W akcji na zmianę kontekstu początkowo wypisujemy nick obecnego gracza.

**if(**context**.**getName**().**equals**(**"BuyingItem"**)){**

System**.**out**.**println**(**context**.**getBoard**().**toString**());**

**for(**Player p **:** context**.**getAllPlayers**())**

System**.**out**.**println**(**p**.**toString**());**

System**.**out**.**println**(**"Decision (Yes/No): "**);**

Scanner scanner **=** **new** Scanner**(**System**.**in**);**

String decision **=** scanner**.**next**();**

**return** **new** TurnResult**(**"Decision"**,** decision**);**

**}**

Jeżeli kontekstem jest Buyingitem wypisujemy planszę, wszystkich graczy, pobieramy decyzję gracza co do kupna przedmiotu i zwracamy jego decyzję poprzez TurnResult.

**else** **if(**context**.**getName**().**equals**(**"ChanceField"**)){**

System**.**out**.**println**(**context**.**getBoard**().**toString**());**

**for(**Player p **:** context**.**getAllPlayers**())**

System**.**out**.**println**(**p**.**toString**());**

System**.**out**.**println**(**"Youre standing on chance field. 100 $ get."**);**

Scanner scanner **=** **new** Scanner**(**System**.**in**);**

String moveVal **=** scanner**.**next**();**

**return** **new** TurnResult**(**"Chance"**,**"TheChance"**);**

**}**

W wypadku gdy obecny gracz stanął na polu szansy, informujemy go o tym i zwracamy wylosowaną szansę. Analogicznie wypełniamy pozostałe rodzaje kontekstów:

**else** **if(**context**.**getName**().**equals**(**"PlayerMove"**))** **{**

System**.**out**.**println**(**context**.**getBoard**().**toString**());**

**for(**Player p **:** context**.**getAllPlayers**())**

System**.**out**.**println**(**p**.**toString**());**

System**.**out**.**println**(**"Move:"**);**

Scanner scanner **=** **new** Scanner**(**System**.**in**);**

String moveVal **=** scanner**.**next**();**

**return** **new** TurnResult**(**"Move"**,** moveVal**);**

**}** **else** **{**

System**.**out**.**println**(**context**.**getName**());**

System**.**out**.**println**(**context**.**getBoard**().**toString**());**

**for(**Player p **:** context**.**getAllPlayers**())**

System**.**out**.**println**(**p**.**toString**());**

System**.**out**.**println**(**"Write sthing"**);**

Scanner scanner **=** **new** Scanner**(**System**.**in**);**

String moveVal **=** scanner**.**next**();**

**return** **new** TurnResult**();**

**}**

**}**

**};**

W wypadku ruchu gracza pobieramy o jaką liczbę oczek się ruszył i zwracamy do logiki gry. W pozostałym wypadku tylko wypisujemy na ekranie obecny stan i zwracamy w pełni utworzony uchwyt na zdarzenia z domyślnym obiektem Turnmakeable.

Następnie tworzymy uchwyt na zdarzenia gdy obecnym graczem jest inny gracz:

TurnMakeable otherTurnMaker **=** **new** TurnMakeable**()** **{**

@Override

public TurnResult makeTurn**(**Context context**){**

System**.**out**.**println**(**context**.**getName**());**

System**.**out**.**println**(**context**.**getBoard**().**toString**());**

**for(**Player p **:** context**.**getAllPlayers**())**

System**.**out**.**println**(**p**.**toString**());**

**return** **new** TurnResult**();**

**}**

**};**

W tym wypadku tak samo zastosujemy domyślny obiekt TurnMakeable który przy zdarzeniach innego gracza tylko wypisuje przekazane informację i akcję podjętę przez obecnego gracza.

## Implementacja szczegółów dotyczących gry

W wielu przypadkach rzeczą przydatną mogłoby być przechowywanie dodatkowych atrybutów gry takich jak nazwa czy możliwa liczba graczy na która gra jest pisana. Jest to osiągane na platformie poprzez implementację interfejsu GameDetails który wygląda następująco:

package boardGamePlatform**.**gameInitializers**;**

public interface GameDetails **{**

public boolean ifPlayersCountPossible**(** int playersCount**);**

public String getGameName**();**

**}**

W interfejsie znajdują się dwie metody :

* getGamename do zwrócenia nazwy wytworzonej gry
* ifPlayersCountPossible zwracająca czy gra została stworzona z myśla o podanej w parametrze liczbie graczy.

Nasza implementacja wygląda w ten sposób:

class MonopolyDetails1 **implements** GameDetails**{**

public boolean ifPlayersCountPossible**(**int playersCount**)** **{**

**if(**playersCount **==** 2**)**

**return** **true;**

**return** **false;**

**}**

public String getGameName**()** **{**

**return** "Monopoly"**;**

**}**

**}**

## Połączenie modułów w obiekt tworzący grę

Z punktu widzenia platformy cała logika gry , inicjalizacja i reakcji użytkowników są widziane jako obiekt GameFactory. Jest to sztampowy przykład wykorzystania wzorca fabryka abstrakcyjna a wygląda on następująco:

package boardGamePlatform**.**gameInitializers**;**

**import** java**.**util**.**List**;**

**import** boardGamePlatform**.**game**.**Player**;**

**import** boardGamePlatform**.**game**.**State**;**

**import** boardGamePlatform**.**game**.**TurnHandler**;**

**import** boardGamePlatform**.**game**.**TurnLogic**;**

**import** boardGamePlatform**.**gameResources**.**Board**;**

public interface GameFactory **{**

public ItemsInitializer createItemsInitializer**();**

public PawnsInitializer createPawnsInitializer**();**

public BoardInitializer createBoardInitializer**();**

public PlayersInitializer createPlayersInitializer**();**

public TurnHandler createCurrentTurnHandler**();**

public TurnHandler createOtherTurnHandler**();**

public GameDetails createGameDetails**();**

public State createInitialState**(**Board board**,** List**<**Player**>** allPlayers**,** TurnLogic turnLogic**);**

**}**

Zawiera on funkcję zwracającą obiekty potrzebna do prawidłowego działania gry o których mowa była wcześniej. Jedyną funkcją której przeznaczenia może być nie do końca jasne jest createInitialState. Słuzy ona do zwrócenia początkowego stanu od którego rozpocznie się gra.

Nasza implementacja wygląda następująco:

public class MonopolyGameFactory1 **implements** GameFactory **{**

public ItemsInitializer createItemsInitializer**()** **{**

**return** **new** ItemsInitializerImpl1**();**

**}**

public PawnsInitializer createPawnsInitializer**()** **{**

**return** **new** PawnsInitializerImpl1**();**

**}**

public BoardInitializer createBoardInitializer**()** **{**

**return** **new** BoardInitializerImpl1**();**

**}**

public PlayersInitializer createPlayersInitializer**()** **{**

**return** **new** PlayersInitializerImpl1**()** **;**

**}**

public TurnHandler createCurrentTurnHandler**()** **{**

TurnMakeable currTurnMaker **=** **new** TurnMakeable**()** **{**

…. // tak jak w sekcji do tworzenia uchwytu

**}**

public TurnHandler createOtherTurnHandler**()** **{**

TurnMakeable otherTurnMaker **=** **new** TurnMakeable**()** **{**

… // jak w sekcji do tworzenia uchwytu

**}**

public GameDetails createGameDetails**()** **{**

**return** **new** MonopolyDetails1**();**

**}**

public State createInitialState**(**Board board**,** List**<**Player**>** allPlayers**,**

TurnLogic turnLogic**)** **{**

**return** **new** StartGame**(**board**,**allPlayers**.**get**(**0**)** **,** allPlayers**,**turnLogic**);**

**}**

**}**

Jak widać implementacją fabryki jest prostym zwracaniem obiektów wytworzonych wcześniej w dokumencie w odpowiednich metodach fabryki.

## Rozpoczęcie gry

Najprostszy scenariusz w którym jeden z graczy tworzy pokój , czeka na drugiego gracza i w razie jego pojawienia się rozpoczyna rozgrywkę jest bardzo prosta. Najpierw należy zainicjalizować serwer:

public static void main**(**String**[]** args**)** **{**

boardGamePlatform**.**network**.**Server srv **=** **new** boardGamePlatform**.**network**.**Server**(**"127.0.0.1"**,** 1099**,** **new** MonopolyGameFactory1**());**

**}**

Na odpowiednim ip i porcie zostanie zainicjalizowany serwer.

Po stronie gracza tworzącego pokój należy wykonać:

public static void main**(**String**[]** args**)** **{**

Client clientAdmin **=** **new** Client**(**"127.0.0.1"**,** 1099**,** "Ciochon"**,** **new** MonopolyGameFactory1**());**

clientAdmin**.**makeRoom**(**"PokojCiochonia"**);**

**}**

W naszym przykładzie łączymy się z serwerem działającym na localhost z portem jak w serwerze, nazwą gracza i odpowiednią fabryką gry. Następnie tworzymy pokój.

Po stronie gracza przyłączającego się do rozgrywki:

public static void main**(**String**[]** args**)** **{**

Client clientPlayer **=** **new** Client**(**"127.0.0.1"**,** 1099**,** "Listkiewicz"**,** **new** MonopolyGameFactory1**());**

clientPlayer**.**joinFirstAvailableRoom**();**

**}**

Podłączamy się do serwera analogicznie jak w admininstratorze pokoju i łączymy się z pierwszym dostępnym pokojem i to wszystko.

# Podsumowanie

Platforma zapewnia mechanizmy do tworzenia gry sprowadzające całą złożoność i wymaganą kreatywność przy jej implementacji do szeregu rutynowych kroków które twórca gry musi wykonać. Do stworzenia sieciowej gry Monopoly z grafiką konsolową nie jest wymagane od twórcy gry zgłębianie się w grafikę czy programowanie sieciowe .

# Bibliografia

1. Brett D. McLaughlin, Gary Police & David West: *Object Oriented Analysis and Design*, O’Reilly, 2007

Indeks ważniejszych terminów

**Nie można odnaleźć żadnych haseł indeksu.**

1. Spis ilustracji i tabel

Ilustracje

**Nie można odnaleźć pozycji dla spisu ilustracji.**

Tabele