

GOLD RUSH



Iteracja 2, tydzień 1: zbieranie złota i inne uciechy

Jaki jest plan?

Zbieramy złoto!

Pierwsza implementacja interakcji pionka gracza z żetonem złota.

Gracz i gra

Wprowadzamy postać gracza oraz reprezentację gry.

Czasem znajdujemy więcej, czasem mniej, a czasem wcale

Dajemy możliwość zróżnicowania żetonów złota.

Poza tym testujemy strukturę programu, wprowadzając nowy typ żetonu: piryt, czyli złoto głupców.

Interakcja pionka na poważnie

Wstępna implementacja zbierania złota na dłuższą metę się nie sprawdzi. Trzeba to zrobić z głową.

Plan na kolejny tydzień

Im jesteśmy dalej, tym więcej otwiera się możliwości rozbudowy. Do dzieła!

Zbieramy złoto

Podstawowe pytanie: **gdzie zaimplementować** zbieranie złota?

Co mamy?

- klasę **Board**,
- rodzinę klas **Token**,
- główną klasę z metodą **main()**.

Akcja rozgrywa się na planszy, więc może tu?..

...ale przecież zbieranie złota to
interakcja między pionkiem a żetonem,
więc może tu?

hm...?

Na pewno łatwiej będzie podjąć decyzję, jeżeli spojrzymy na kod i zastanowimy się,
za co odpowiadają poszczególne klasy i metody.

W klasie **Board** mamy jedną kandydatkę: metodę **placeToken()**:

```
public void placeToken(int col, int row, Token token) {  
    grid[col][row] = token;  
}
```

Za co ona odpowiada?

Za umieszczanie żetonów na planszy.

Czy interesuje ją jaki żeton kładzie?

Czy powinno ją to interesować?

Czy umieszczanie tu logiki obsługującej interakcję
między konkretnymi żetonami jest właściwe?

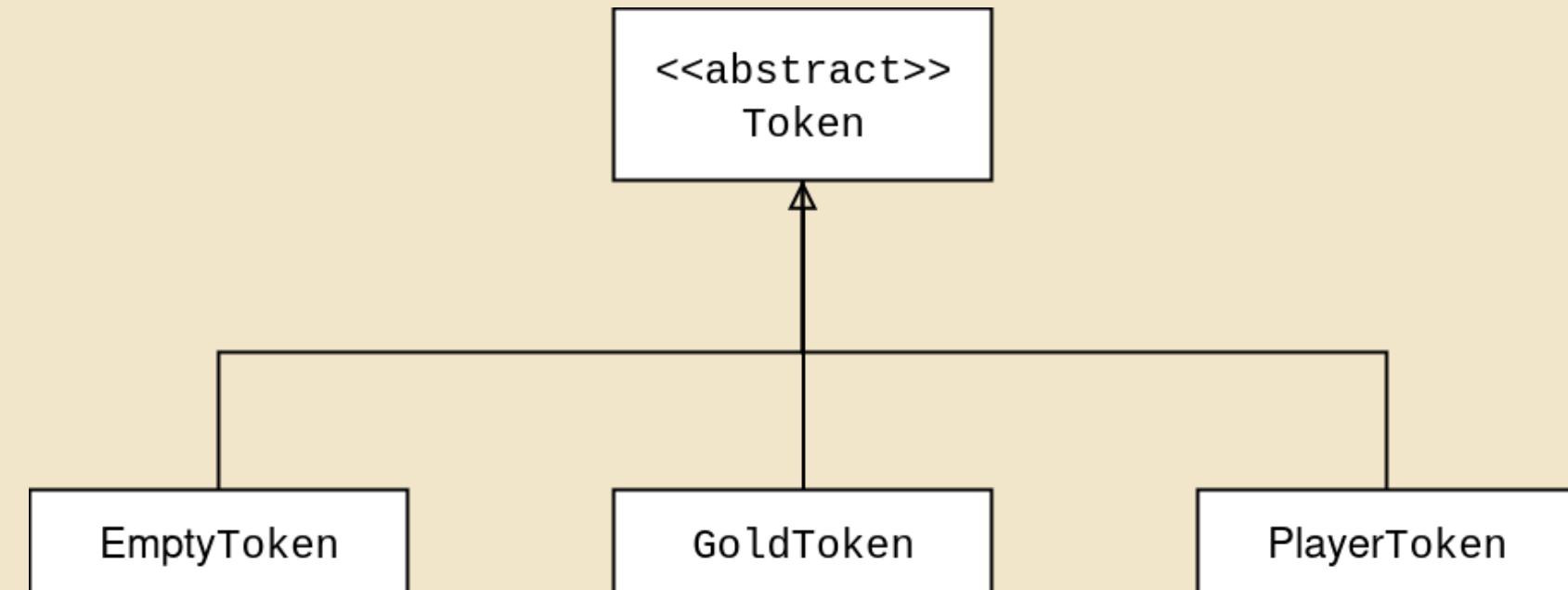
NIE!

Board
- size: int - grid: Token[][]
+ Board() + size() + clean() + placeToken(col, row, Token) + peekToken(col, row): Token + display()

Rzućmy okiem na kolejną możliwość: rodzinę klas **Token**.

Na rodzinę składają się:

- abstrakcyjna klasa bazowa **Token**,
- konkretna klasa **EmptyToken**,
- konkretna klasa **GoldToken**,
- konkretna klasa **PlayerToken**.



PlayerToken wydaje się ciekawy, zwłaszcza, że ma metodę **move()**:

```
public void move(Move dir) {
    ...
    switch (dir) {
        case Move.UP: row -= 1; break;
        ...
    }
    ...
}

board.placeToken(pCol, pRow, new EmptyToken());
board.placeToken(col, row, this);
}
```

Manipulujemy tu żetonami na planszy.
Dobry trop!

To miejsce będzie odpowiednie: po obliczeniu nowych
współrzędnych, ale przed umieszczeniem tam pionka
(i zastąpieniem dotychczasowego żetonu).

Dzięki temu, że poszczególne **typy żetonów są reprezentowane przez różne klasy** (ale z tej samej rodziny), sprawdzanie na co wchodzi pionek możemy rozwiązać lepiej niż sprawdzając, np. etykietkę żetonu:



```
if (token.label().equals("💰")) {  
    ...  
}
```

Etykietka może się zmienić. I co wtedy?
Porównywanie stringów w takich sytuacjach
naprawdę brzydko pachnie...

```
if (token instanceof GoldToken) {
```

}

Na razie po znalezieniu złota wykrzyknijmy (w konsoli): "GOLD!"



A niebawem zobaczymy jeszcze milszą postać tego warunku...

Gracz i gra

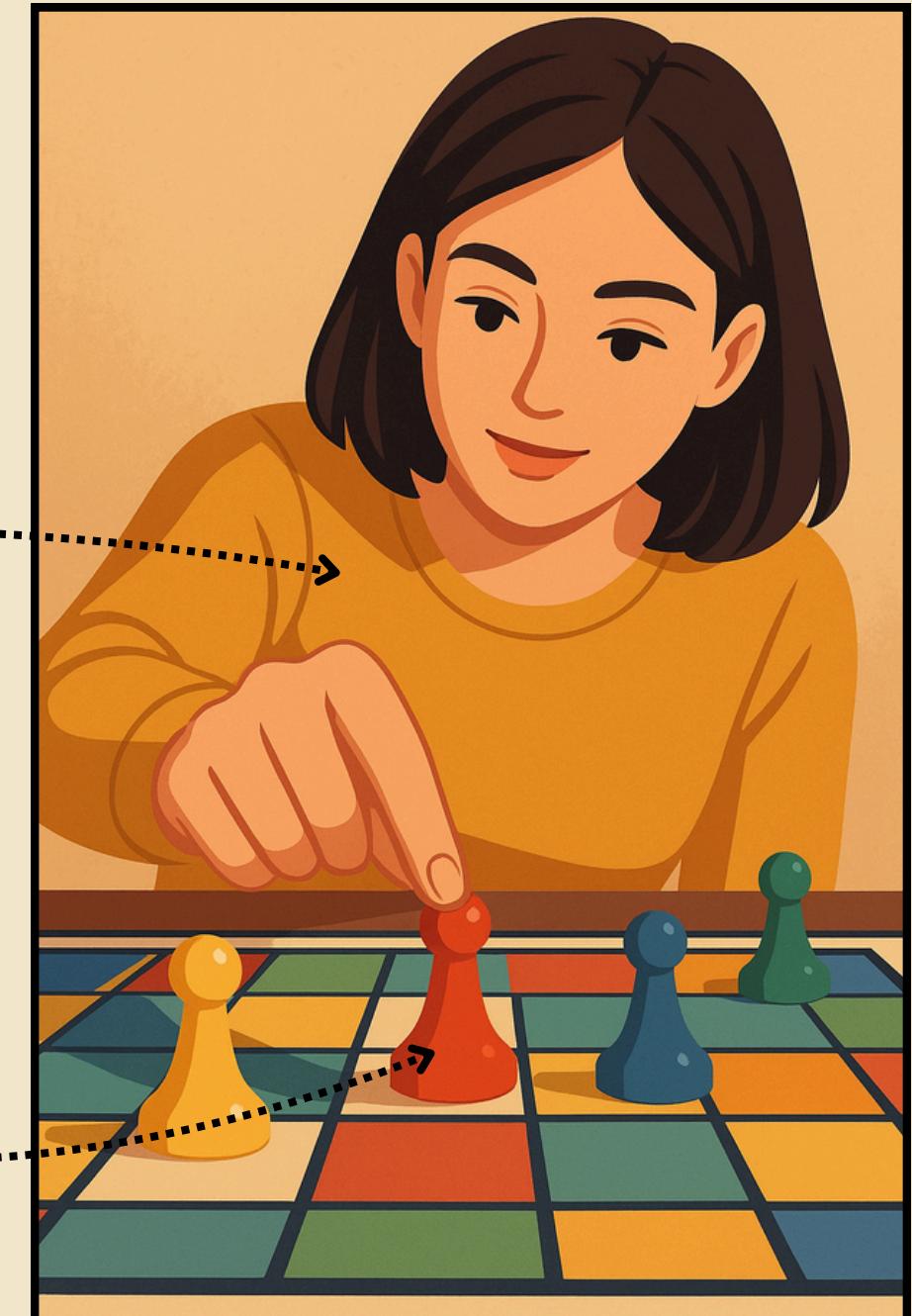
Mamy już zaimplementowane wykrycie wejścia pionka na pole z żetonem złota.

Co dalej?

Gracz ma zebrać złoto. I tu pojawiają się pytania...

Jeżeli ma zebrać, to musi je gdzieś przechowywać.

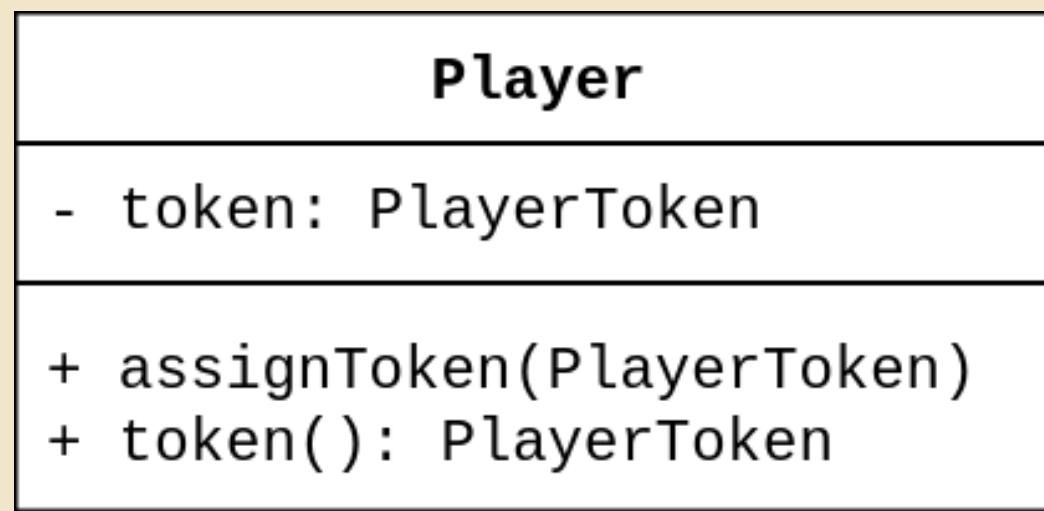
Chwila, a kim jest gracz? Mamy pionek gracza. Czy to to samo?



Wprowadźmy klasę reprezentującą gracza: **Player**.

Co powinien wiedzieć „player”?

Na razie niewiele:



Gracz musi znać swój pionek.

Pionek musimy obsłużyć:

- przypisujemy pionek do gracza
- sprawdzamy: jaki masz pionek?

A gdzie jest złoto? Powoli...

Wprowadziłmy klasę **Player**, która reprezentuje gracza.

Pójdzmy o krok dalej: a co z **grą**?



No właśnie: nie mamy tak naprawdę klasy odpowiedzialnej za przebieg gry.

Klasa **Board**?

Nie: to tylko plansza. Nie wrzucajmy na kawałek twardego, kolorowego papieru **za dużo odpowiedzialności**...

Klasa z metodą **main()**?

Bez żartów: to punkt wejścia do aplikacji.

Z tego miejsca powinniśmy uruchomić **prawdziwą aplikację**.

Czyli czas na klasę **Game**.

Gra rozgrywa się na planszy,
więc musimy tę planszę utworzyć *.

Do gry dodajemy gracza.

Musimy więc go przechować **.

Całą mechanikę gry
przenosimy do metody start().

Game
- board: Board - player: Player
+ Game() + join(Player) + start()

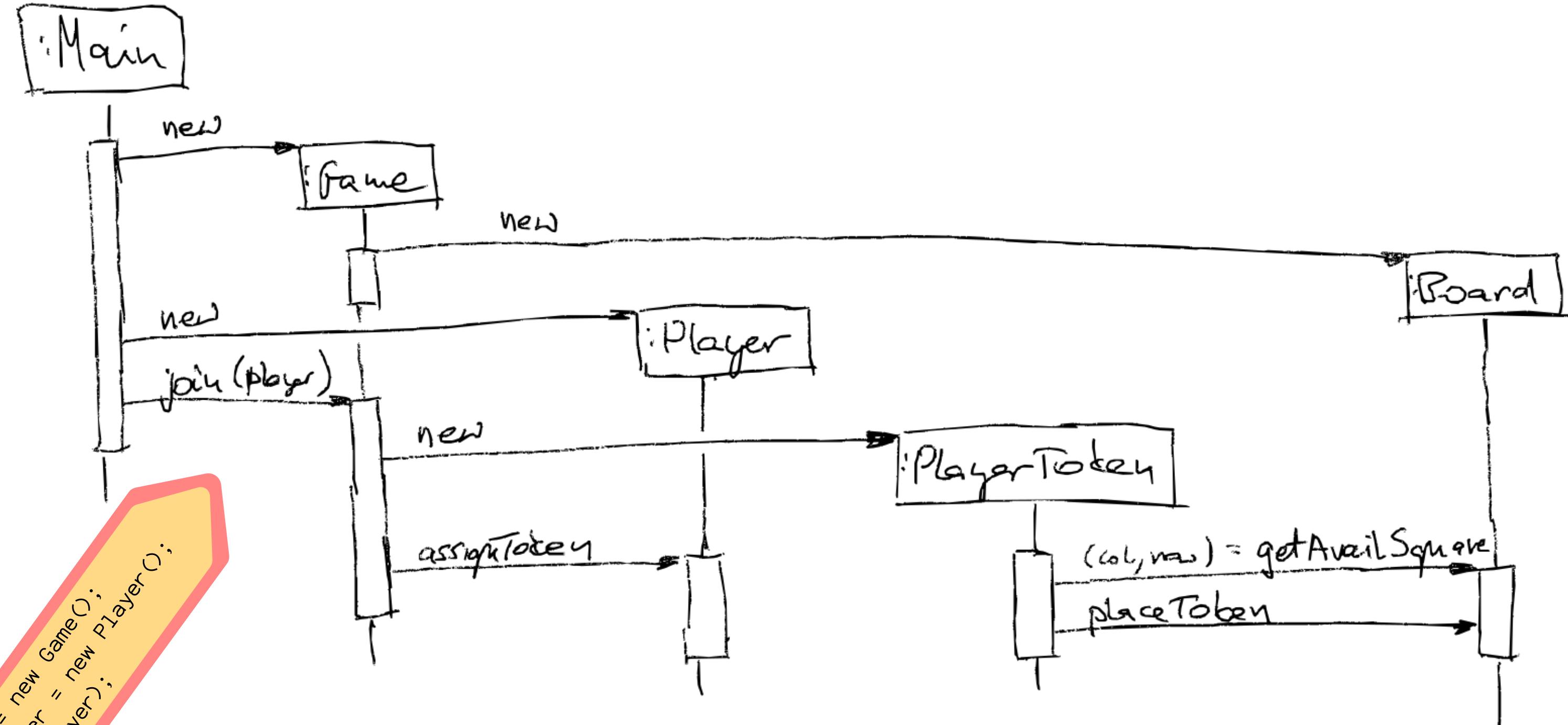
Dzięki zmianom metoda **main()** wygląda tak:

```
public static void main(String[] args) {  
    Game game = new Game();  
    Player player = new Player();  
    game.join(player);  
    game.start();  
}
```

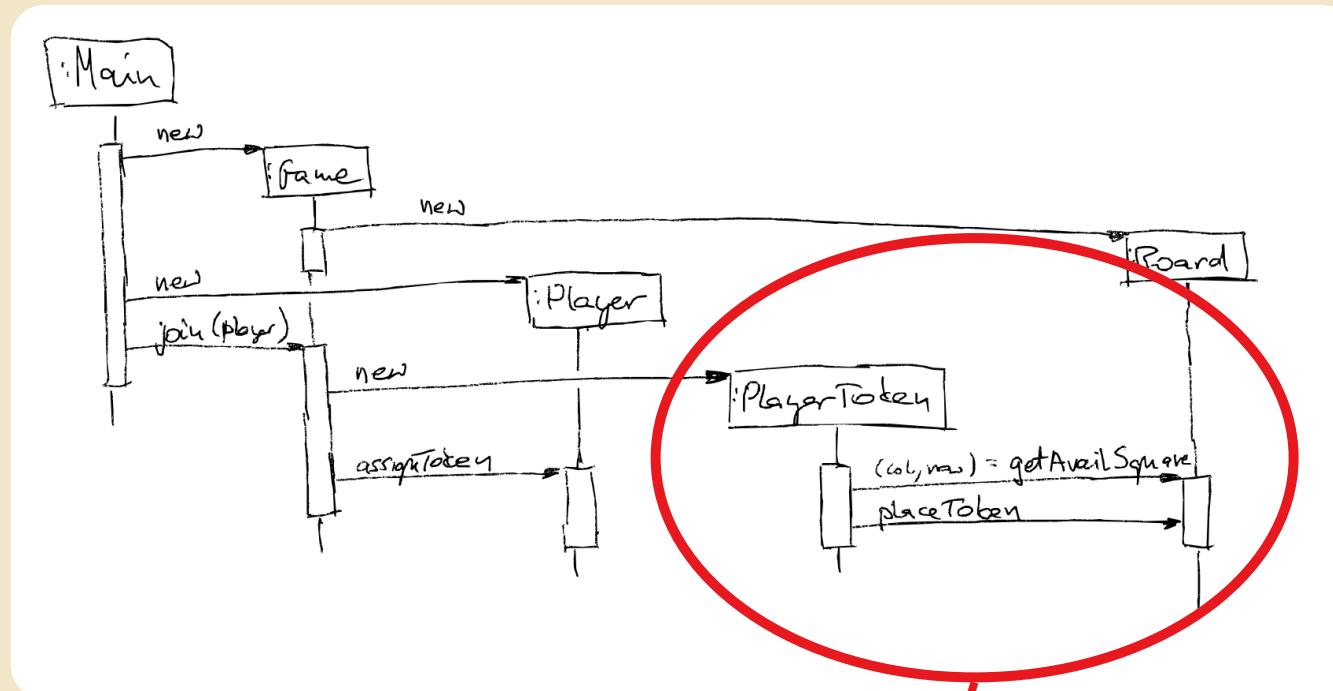
* Obiekt planszy można przekazać przez konstruktor (DI), ale w tej chwili nie jest to potrzebne.

** Na razie mamy jednego gracza. W przyszłości będzie ich wielu.

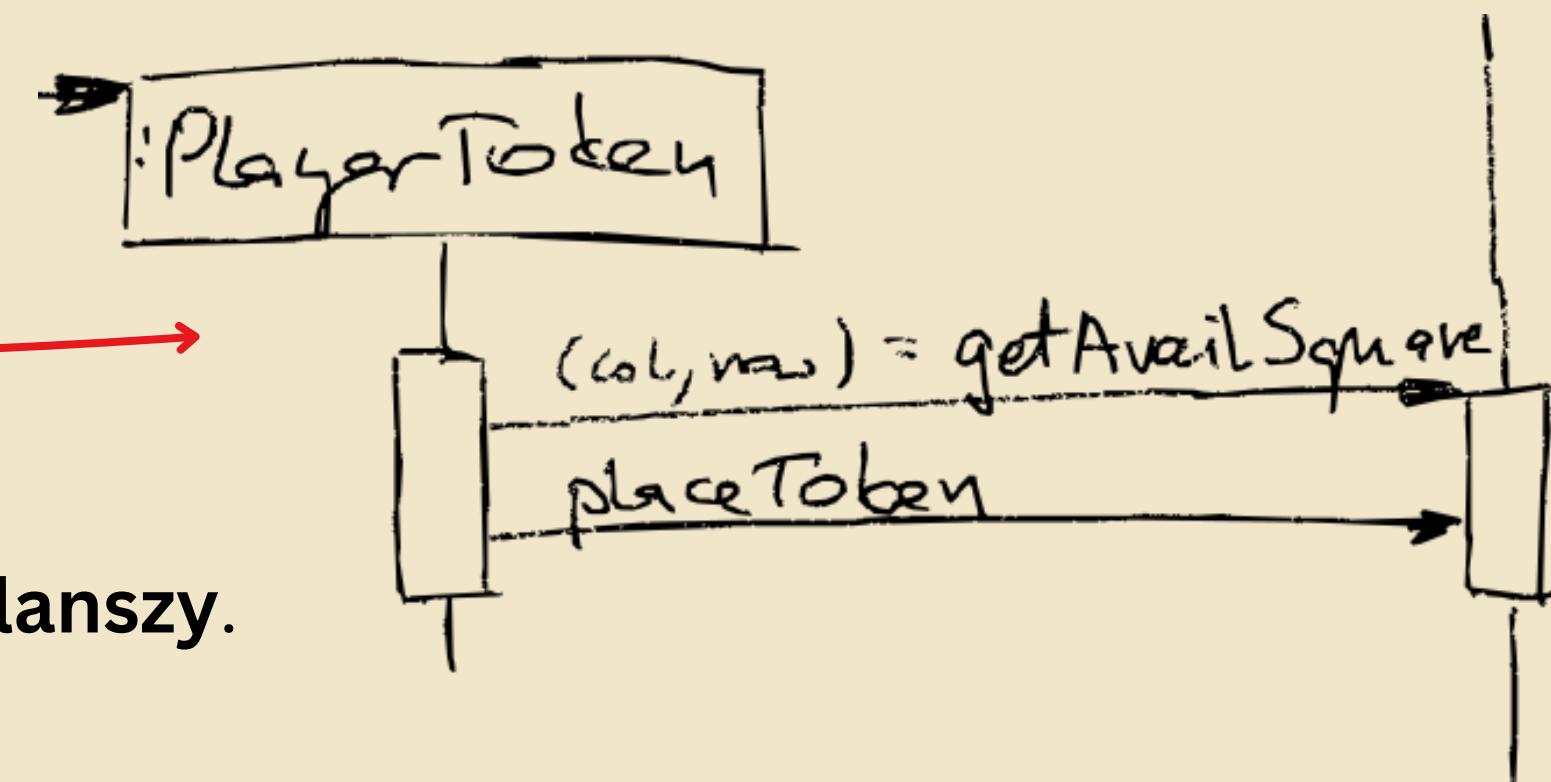
Ale jak to działa? Wszystko pięknie widać na **diagramie sekwencji**:



Większość wygląda tu znajomo: te klasy i ich metody poznaliśmy wcześniej.



Jest jeden wyjątek: metoda **getAvailableSquare()** w klasie Board – jej jeszcze nie było.



Jej zadaniem jest **wskazanie wolnego miejsca na planszy**.

W pierwszej implementacji niech to będzie po prostu kolejne wolne miejsce.

Z metody korzystamy w konstruktorze **PlayerToken**,
by nowo utworzony pionek od razu położyć na planszy.

Uwaga: na diagamie jest "getAvailSquare",
bo pełna nazwa była za długa i się nie mieściła ;)
Trzymamy się pełnej nazwy `getAvailableSquare`.

Kilka slajdów temu, w czasie definiowania klasy **Player**, padło pytanie: „A gdzie jest złoto?”

Co oczywiste, gracz musi **przechowywać** złoto, które zebrał.

Prosta sprawa: w klasie Player musimy dodać **pole „gold”**. Jakiego typu?

Wiele zależy od tego, jak przechowujemy złoto.

A właściwie w jakich jednostkach.

gramy

ziarna (grains)

uncje (oz)

pennyweight (dwt)

int

double

Integer

Double

BigDecimal

Przyjmujemy, że złoto reprezentujemy w **uncjach trojańskich**.

Musimy uwzględnić **części ułamkowe**.



Jednostka ustalona, typ też: **double**. Ma wady, ale jest najprostszy w zastosowaniu.

Kolejne wątpliwości: jak w klasie Player obsłużyć złoto?

Moglibyśmy po prostu dodać *getter* i *setter*. Prosta sprawa, klasyczne rozwiązanie.

Ale Player nie jest żadnym
POJO ani JavaBean!

Getter może być, ale skoro reprezentujemy gracza, może lepiej będzie wyposażyć go w **metody zwiększające i zmniejszające ilość posiadanego złota**?

Player	
- token:	PlayerToken
- gold:	double
+ assignToken(PlayerToken)	
+ token():	PlayerToken
+ gold():	double
+ gainGold(double)	
+ loseGold(double)	

Do pełni szczęścia (związanego ze zbieraniem złota) brakuje nam jeszcze właściwości opisującej ilość złota w żetonie **GoldToken**.

Robi się!

GoldToken
- amount : double
+ GoldToken()
+ GoldToken(amount)
+ amount() : double



Wszystko przygotowane, możemy zaimplementować zbieranie złota **z uwzględnieniem jego ilości** w złożu (żetonie).

Szczerbaty Joe i Wąski Ed prezentują swój dzienny urobek
okolice Jamestown, lipiec 1853

Dla przypomnienia: zbieranie złota mamy wstępnie zaimplementowane w metodzie **PlayerToken.move()**:

```
var token = board.peekToken(col, row);  
if (token instanceof GoldToken) {  
    System.out.println("GOLD!");  
}  
}
```

Wyświetlanie może zostać, ale przede wszystkim musimy zebrać złoto.

Sprawa wydaje się prosta – metodzie **gainGold()** musimy przekazać ilość złota, która znajduje się w żetonie:

```
player.gainGold( token.amount() );
```

```
player.gainGold(token.amount());
```

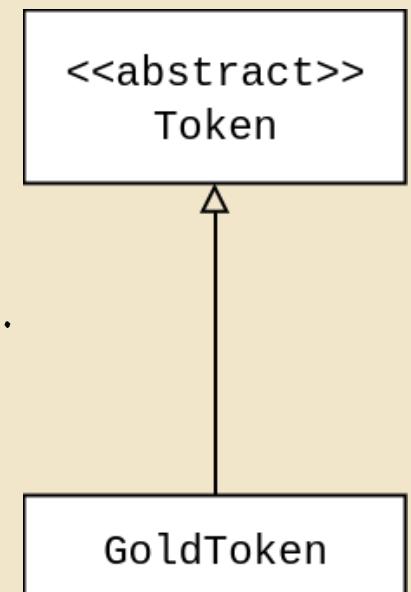
Cannot resolve method 'amount' in 'Token'

Cast qualifier to 'edu.io.token.GoldToken' Alt+Shift+Enter

No candidates found for method call token.amount().



Problem wynika stąd, że w klasie **Token** nie ma metody **amount()**.



Ale przecież nas interesuje właśnie **GoldToken**!

Tak, tylko że obiekt dostajemy z metody **peekToken()**, której typ zwracany to **Token** (a nie **GoldToken**).

Dodaliśmy ją w klasie **GoldToken**.

```
var token = board.peekToken(col, row);
```

Musimy więc **rzutować**, tak żeby z ogólnego żetonu „zrobić” żeton złota:

```
player.gainGold( ((GoldToken) token).amount() );
```

Na szczęście możemy to załatwić **prościej i bezpieczniej**:

```
var token = board.peekToken(col, row);
if (token instanceof GoldToken gold) {
    ...
    player.gainGold(gold.amount());
}
```

PATTERN MATCHING

Pięknie! Udało nam się pożądnie zaimplementować zbieranie złota.

Na pewno? Spójrzmy na metodę **PlayerToken.move()**:

```
public void move(Move dir) {  
    ...  
    switch (dir) {  
        case Move.UP: row -= 1; break;  
        ...  
    }  
  
    var token = board.peekToken(col, row);  
    if (token instanceof GoldToken gold) {  
        ...  
    }  
  
    board.placeToken(pCol, pRow, new EmptyToken());  
    board.placeToken(col, row, this);  
}
```

Annotations with arrows pointing to specific code snippets:

- An arrow points from the `case Move.UP: row -= 1; break;` line to the handwritten note *Obsługa pouszania się pionka*.
- An arrow points from the `if (token instanceof GoldToken gold) {` line to the handwritten note *Obsługa zbierania złota*.
- An arrow points from the `board.placeToken(pCol, pRow, new EmptyToken());` line to the handwritten note *A co, kiedy pojawią się narzędzia, bonusy i inne żetony?*.
- An arrow points from the `board.placeToken(col, row, this);` line to the handwritten note *Aktualizacja planszy*.

Czy nie za dużo się tu dzieje?

PROPOZYCJA

Zostawmy w pionku sprawy związane z **poruszaniem się po planszy**.

Sprawy **interakcji gracza z żetonami** przenieśmy do klasy **Player**.

Klasa Player zyskuje metodę o wdzięcznej nazwie **interactWithToken(Token)** .

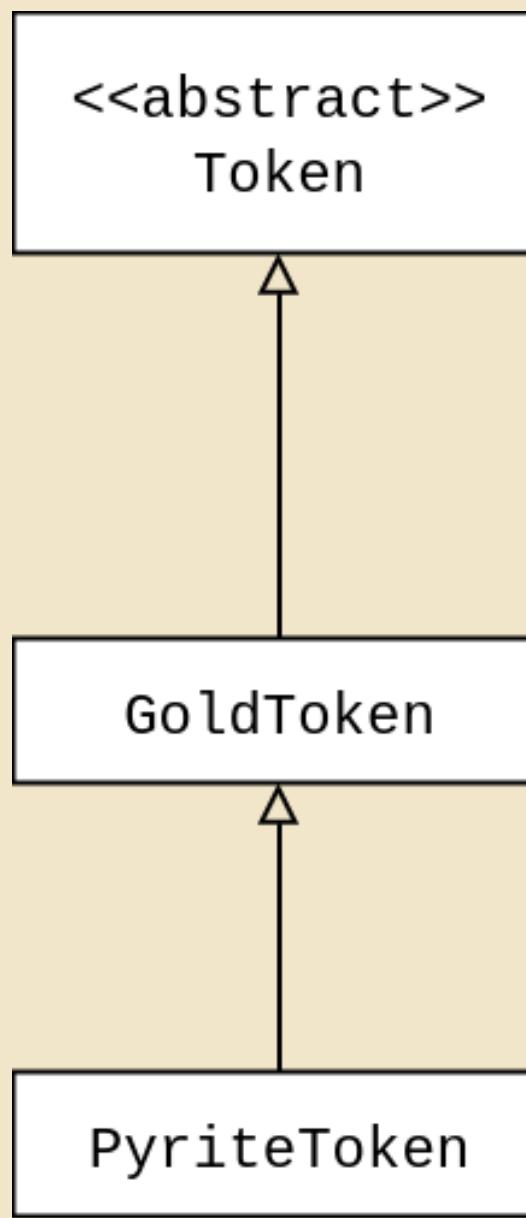
To w niej ma się znaleźć obsługa zbierania złota.

W metodzie **move()** tylko ją wywołamy:

```
player.interactWithToken(board.peekToken(col, row));
```

Player
- token: PlayerToken - gold: double
+ assignToken(PlayerToken) + token(): PlayerToken + gold(): double + gainGold(double) + loseGold(double) + interactWithToken(Token)

Na koniec jeden smaczek: słyszeliście o „złocie głupców”? Chodzi o **piryt**, który łudząco przypomina złoto.



Wprowadźmy do gry żeton reprezentujący piryt.
Założenia: ma **wyglądać** jak złoto, ale mieć **zerową wartość**.

Ponieważ mamy hierarchię klas, wystarczy by klasa **PyriteToken** dziedziczyła po **GoldToken**.

Dzięki temu **piryt jest złotem, ale trochę innym...**

```
public class PyriteToken extends GoldToken {  
    public PyriteToken() {  
        ???  
    }  
}
```

To umiecie zrobić sami!

Podsumowanie zadań

W ramach zadań trzeba **obowiązkowo** zrealizować to, co zostało opisane na slajdach.

Kluczowe jest oczywiście **pozytywne przejście testów i02w01**.

Są też dwa nieobowiązkowe, **dodatkowe zadania**. ← Dla tych, którzy celują w najwyższą ocenę.

Zadanie i02w01ex1

Narysuj diagram sekwencji przedstawiający interakcję gracza z żetonem złota.

Technika rysunku dowolna: może być odręczny (zdjęcie), można skorzystać z jakiegoś narzędzia.

Zadanie i02w01ex2

Metoda **getAvailableSquare()** ma wskazywać wolne pole za pomocą **wybranej strategii**.

Algorytm wybierania miejsca ma być wstrzykiwany z zewnątrz za pomocą metody

Board.setPlacementStrategy(PlacementStrategy). Domyślnie jest ustawiana (wewnętrznie) implementacja wybierająca kolejne wolne miejsce (początkowy algorytm). Zaimportuj strategię **przydzielania losowego pola**. W przypadku zapełnienia planszy ma być rzucany wyjątek. Napisz testy sprawdzające możliwość ustawienia strategii, jej działanie i zachowanie po zapełnieniu planszy.

Plan kolejnej iteracji

- Wprowadzamy przykładowy żeton narzędzia: **kilof**.
- Graczowi organizujemy **skrzynkę na narzędzia**.
- Narzędzia **są używane** podczas gry.
- Narzędzia się **zużywają**.
- Narzędzia można naprawiać: żeton **kowadła**.