

# GOLD RUSH



Iteracja 2, tydzień 2: nie ma jak dobry kilof!

# Jaki jest plan?

Do gry wprowadzamy pierwsze narzędzie: **kilof**.

Dzięki niemu urobek złota będzie większy.

Zróbmy z niego użytek!

Obsługujemy *użycie kilofu podczas zbierania złota*.



Nic nie jest wieczne...

Narzędzia się zużywają, każde ma jakąś wytrzymałość.

Wprowadzamy do gry kowadło.

Porządek!

Trzeba dbać o porządek: w narzędziach i w kodzie.

Refaktoryzujemy i wprowadzamy szopkę na narzędzia.

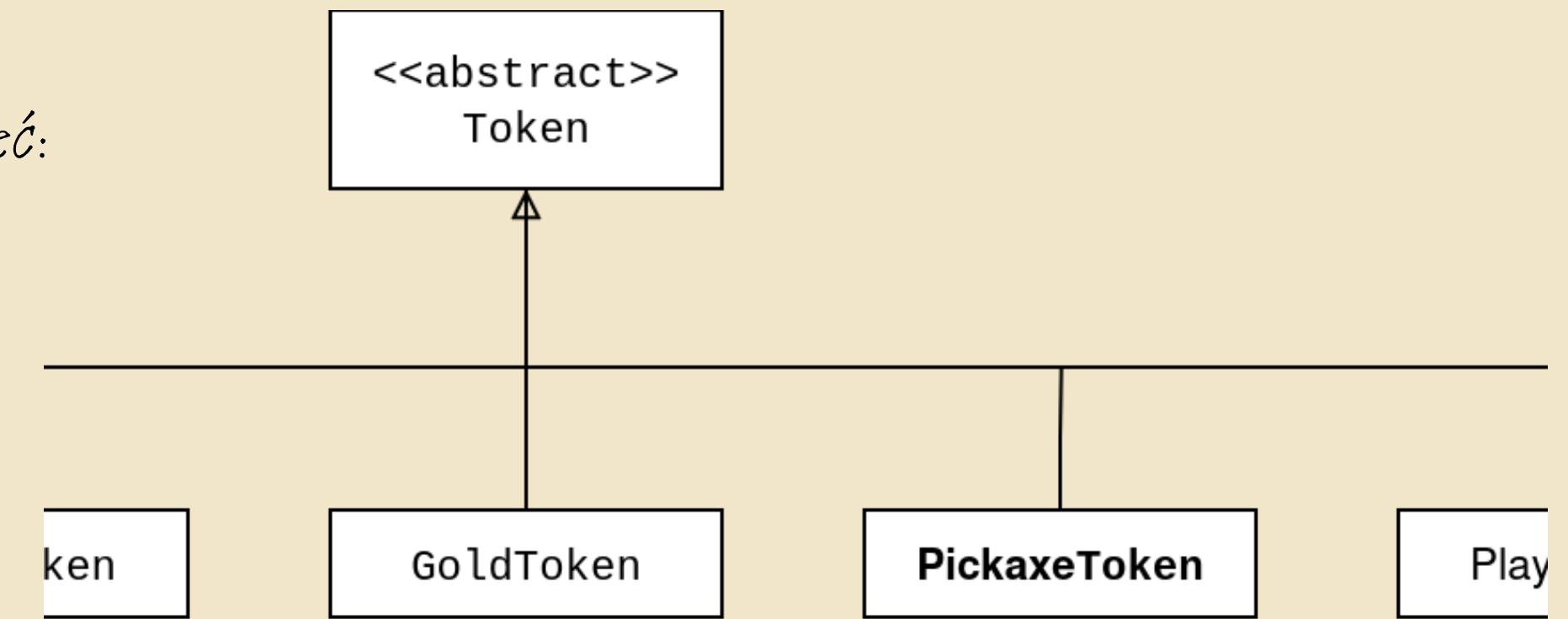
# Kilof

Dzięki przygotowanej wcześniej rodzinie **Token** wprowadzenie nowego żetonu nie jest problemem.

Już nie raz to robiliśmy, ale zawsze warto przypomnieć:

```
public class PickaxeToken extends Token {  
    public PickaxeToken() {  
        super("⛏");  
    }  
}
```

Symbol oczywiście dodajemy do klasy, w której trzymamy etykiety (Label).



Teraz wystarczy zdobyć żeton kilofu i uwzględnić go podczas zbierania złota, czyli przenosimy się do **Player.interactWithToken()**.

Akcja będzie się rozgrywała głównie tu:

```
public void interactWithToken(Token token) {  
    if (token instanceof GoldToken goldToken) {  
        System.out.printf("GOLD!");  
        gainGold(goldToken.amount());  
    }  
}
```

Na razie zbieramy żetony złota.

Zdobywanie kilofu rozwiążemy najprościej, jak się da:

```
if (token instanceof GoldToken goldToken) {  
    ...  
}  
else if (token instanceof PickaxeToken pickaxeToken) {  
    hasPickaxe = true;  
}
```

Player
- ... - hasPickaxe: boolean
... + interactWithToken(Token)

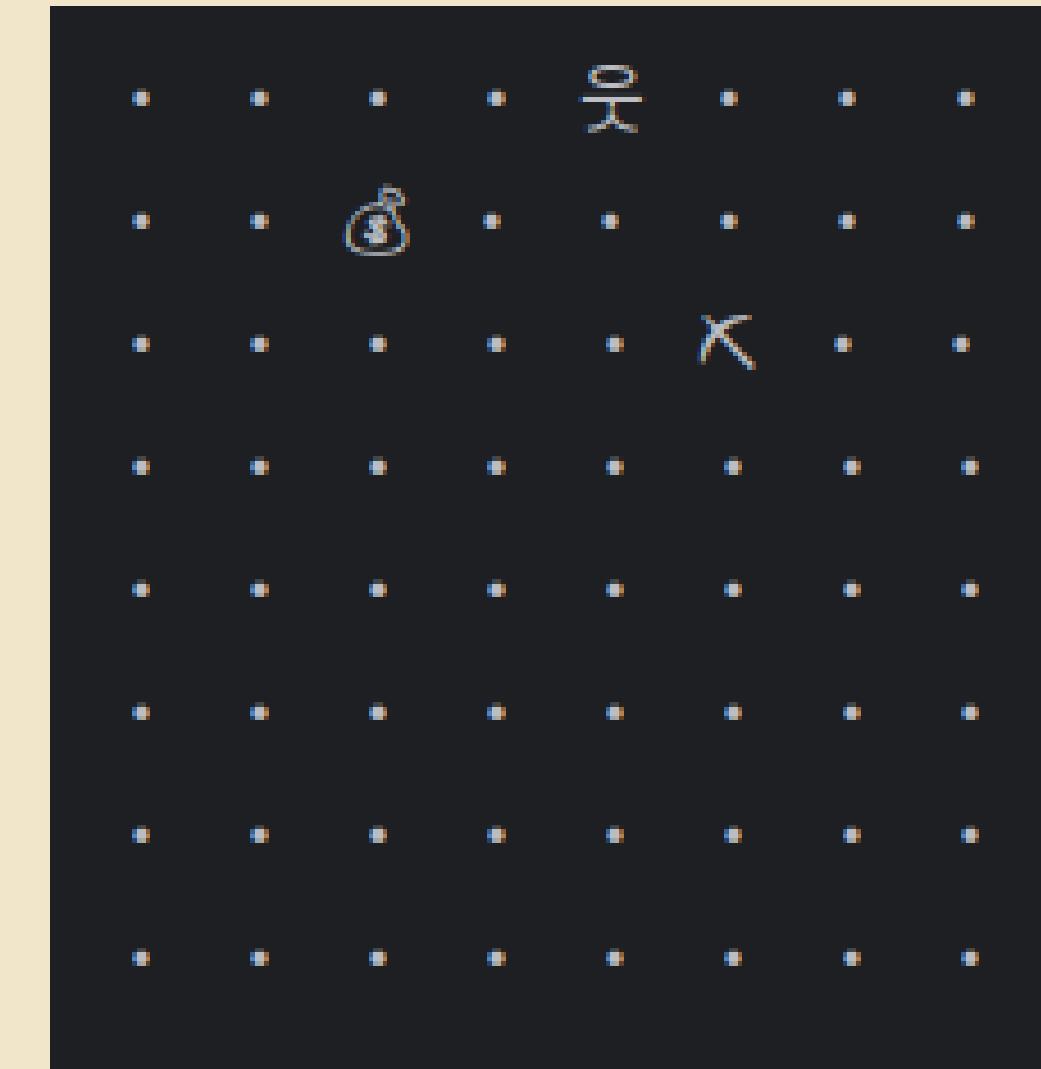
Gracz już wie, czy ma kilof. Możemy go uwzględnić:

```
if (token instanceof GoldToken goldToken) {  
    ...  
    if (hasPickaxe) {  
        amount *= 1.5;  
    }  
    ...  
}
```

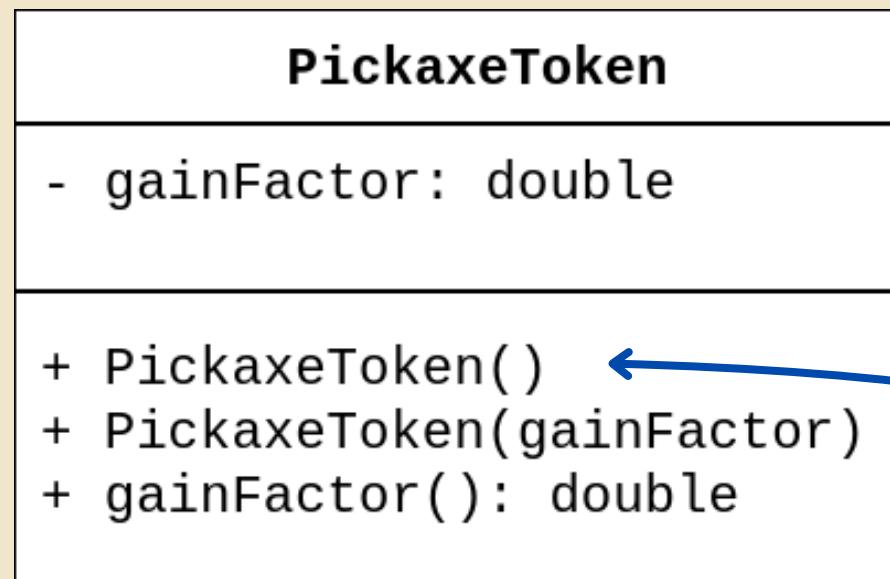
Zaczynamy od stałego współczynnika.

Trzeba to **przetestować**.

Testy jednostkowe swoją drogą,  
zobaczmy **jak to działa!**



Następny krok: chcemy zróżnicować żetony (w końcu są lepsze i gorsze kilofy).



Domyślnie gainFactor = 1.5

Trzeba to uwzględnić w chwili naliczania złota:

```
if (hasPickaxe) {  
    amount *= pickaxeToken.gainFactor();  
}
```

Ale chwila: co to jest **pickaxeToken**?

W klasie **PickaxeToken** mamy tylko **hasPickaxe**.

Jasne: przecież poza odnotowaniem faktu znalezienia kilofu trzeba jeszcze przechować żeton:

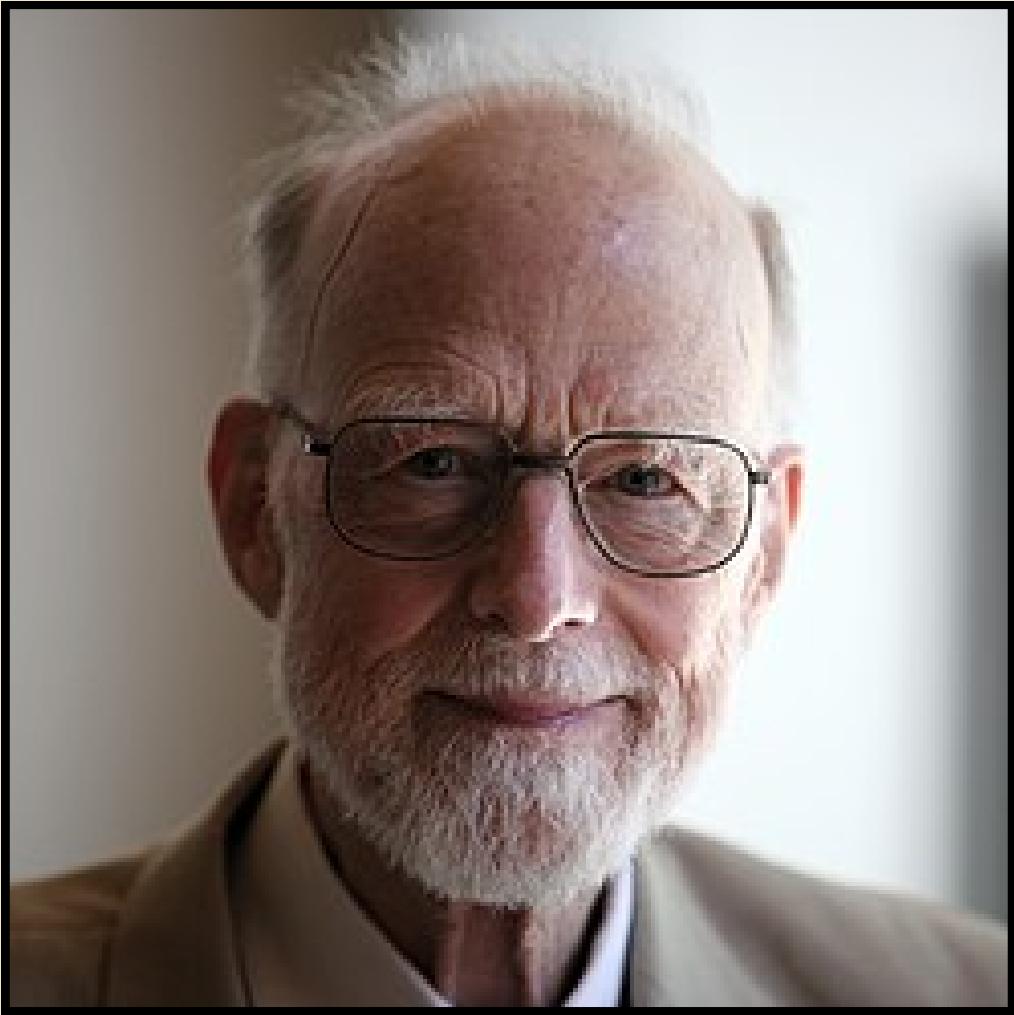
```
...
else if (token instanceof PickaxeToken pickaxeToken) {
    hasPickaxe = true;
    this.pickaxeToken = pickaxeToken;
}
```

A jak zasygnalizować **brak** kilofu? Do tego świetnie nadą się **null**, prawda?

```
class PickaxeToken ...
...
private boolean hasPickaxe = false;
private PickaxeToken pickaxeToken = null;
.....
if (pickaxeToken != null) {
    ... pickaxeToken.gainFactor() ...
}
```



Nadchodzą problemy...



Sir Charles Antony Richard Hoare  
vel **Tony Hoare**

*I call it my billion-dollar mistake. It was the invention of the null reference in 1965.*

[...]

*This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*

Tony Hoare (2009). "[Null References: The Billion Dollar Mistake](#)"

Są sytuacje, w których nie wypada dyskutować. To jedna z nich.

**Nie używamy null-a. Kropka.**

W takim razie jak sygnalizować **brak** kilofu?

Rozwiązanie mamy pod ręką: **EmptyToken**.

```
class PickaxeToken ...  
...  
private PickaxeToken pickaxeToken = null;  
private Token pickaxeToken = new EmptyToken();  
.....  
if (pickaxeToken instanceof PickaxeToken pf) {  
    ... pf.gainFactor() ...  
}
```

Znów „pattern matching”



Kolejna sprawa: **wytrzymałość** kilofu.

Wprowadzamy pole **durability**.

Dodajemy:

- getter,
- kolejny konstruktor,
- metodę **use()**, która powoduje „zużycie” kilofu.

Zakładamy, że kiedy kilof się zużyje, jest usuwany.

Obsługa kilofu może więc wyglądać tak:

```
if (pickaxeToken instanceof PickaxeToken pickaxe) {  
    if (pickaxe.durability() > 0) {  
        amount *= pickaxe.gainFactor();  
        pickaxe.use();  
    }  
    else pickaxeToken = new EmptyToken();  
}
```

PickaxeToken
...
- durability: int
+ PickaxeToken()
+ PickaxeToken(gainFactor)
+ PickaxeToken(gainFactor, durability)
...
+ durability()
+ use()

Jeśli spojrzymy teraz na metodę **interactWithToken()** zauważymy, że większość kodu jest związana z obsługą kilofu.

W tej chwili nie jest to kluczowy problem, ale już niedługo może być: wystarczy dodać kilka narzędzi.

U mnie jest to 5 z 10 linii if-a od zbierania złota plus 2 linie na zdobywanie kilofu.  
Razem 7/10!

Metoda **interactWithToken()** stanie się **odpowiedzialna** za obsługę wszystkich narzędzi. Będzie znała **szczegóły budowy i działania** każdego narzędzia. Straszne!

Zaczniemy od małego kroczku: do **PickaxeToken** dodamy metodę **isBroken()**, która powie nam, czy kilof jest zepsuty (czy nie).

```
public boolean isBroken() {  
    return durability <= 0;  
}
```

W ten sposób schowamy szczegóły działania (durability  $\leq 0$ ).

Dzięki temu kod w **interactWithToken()** może wyglądać choć trochę ładniej.

## Uwaga na marginesie

Przy tej okazji można omówić pewną technikę budowania testów: **AAA**.

```
@Test  
void broken_pickaxe_is_unusable() {  
    // Arrange  
    var player = new Player();  
    var goldToken = new GoldToken(2.0);  
    var pickaxeToken = new PickaxeToken(1.5, 1);  
    player.interactWithToken(pickaxeToken);  
    pickaxeToken.use();  
  
    // Act  
    player.interactWithToken(goldToken);  
  
    // Assert  
    Assertions.assertTrue(pickaxeToken.isBroken());  
    Assertions.assertEquals(2.0, player.gold());  
}
```

„A” jak „arrange”

Aranżujemy, przygotowujemy scenę do występów.

„A” jak „act”

Wszystko przygotowane, więc działamy. Akcja!

„A” jak „assert”

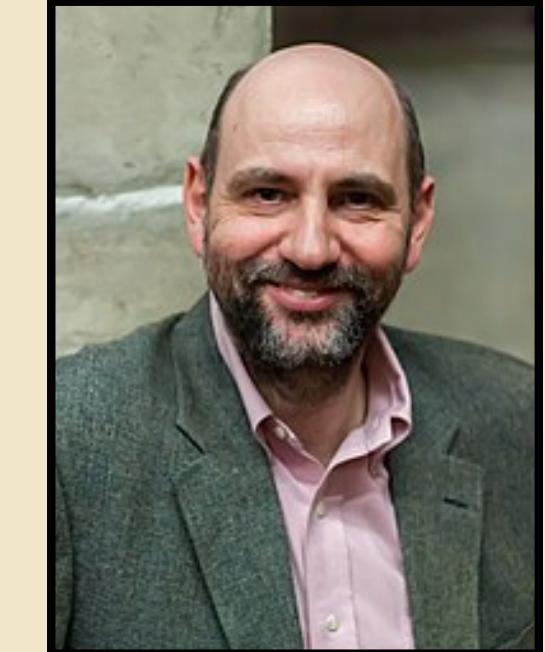
Sprawdzamy, czy dostaliśmy to, czego oczekiwaliśmy. Akceptujemy?

Przejrzyste i intuicyjne – warto stosować tę technikę.

Tak naprawdę jest to okazja do dyskusji o jeszcze jednej koncepcji:  
**fluent interface.**



Chodzi przede wszystkim o *zwiększenie czytelności kodu poprzez wprowadzenie płynności jego czytania*. Istotne jest tu też wyraźne wskazanie kontekstu danego zagadnienia. Termin spopularyzował Martin Fowler.



Martin Fowler

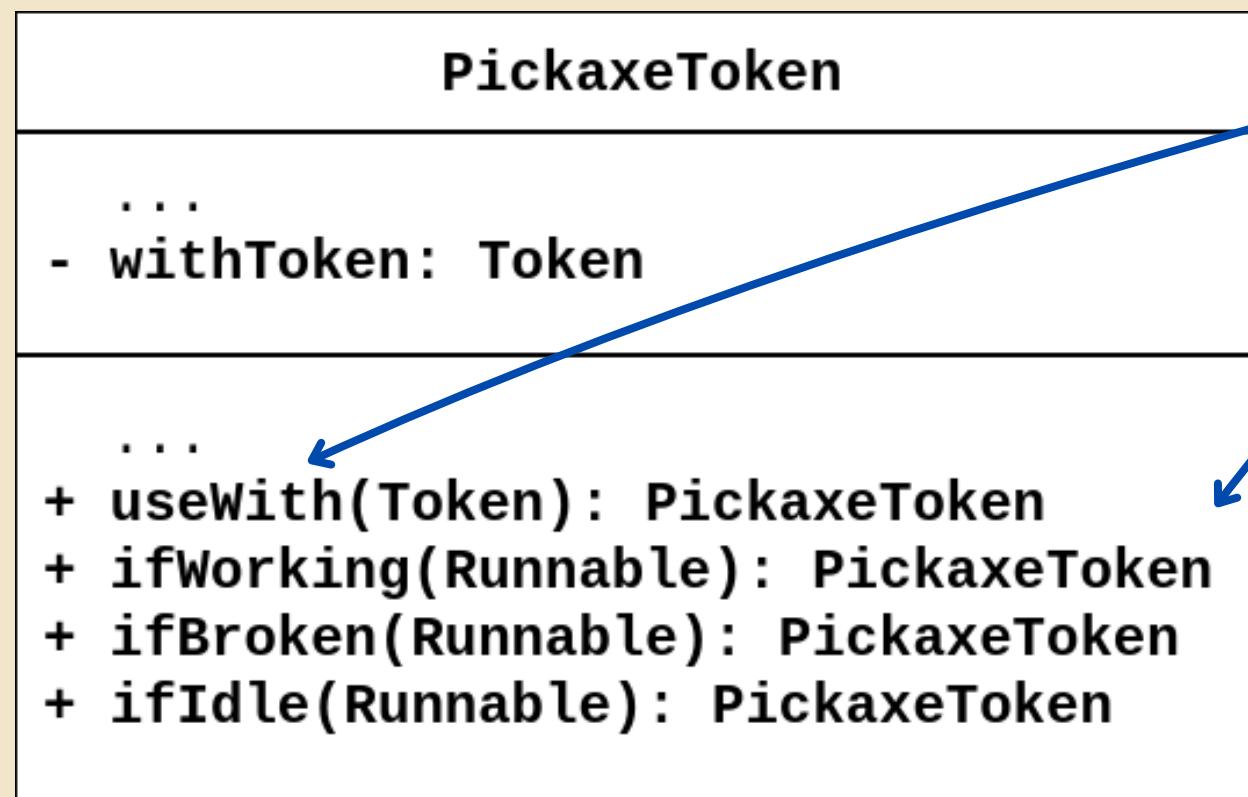
W praktyce: umożliwimy tworzenie łańcuchów wywołań metod (*chaining*).

## PRZED

```
if (...) {  
    pickaxe.use();  
    if (pickaxe.isBroken()) {  
        pickaxeToken = new EmptyToken();  
    }  
    else {  
        amount *= pickaxe.gainFactor();  
    }  
}  
else {  
    gainGold(amount);  
}
```

## PO

```
pickaxe.useWith(goldToken)  
    .ifWorking(() -> {  
        gainGold(amount * pickaxe.gainFactor());  
    })  
    .ifBroken(() -> {  
        gainGold(amount);  
        pickaxeToken = new EmptyToken();  
    })  
    .ifIdle(() -> {  
        gainGold(amount);  
    });
```



Każda metoda zwraca „samego siebie”.  
Dzięki temu możemy tworzyć łańcuchy:

```

pickaxe.useWith(...)
    .ifWorking(...)
    .ifBroken(...)
  
```

W tej metodzie zapamiętujemy obiekt żetonu,  
na którym narzędzie ma być użyte.

A te metody obsługują różne sytuacje.

Budowa metody ***ifCośTam***:

```

public PickaxeToken ifBroken(Runnable action) {
    if (isBroken()) action.run();
    return this;
}
  
```

Jeżeli warunek metody („coś tam”) jest spełniony,  
uruchamiamy akcję.

```

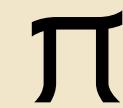
.ifBroken() -> {
    gainGold(amount);
    pickaxeToken = new EmptyToken();
}
  
```

# Kowadło

Możemy już wygodnie korzystać z kilofu.

Kilof starcza nam tylko na kilka grudek złota. Po zużyciu znika z ekwipunku gracza.

Dodajmy żeton, który będzie służył do naprawiania narzędzi: **kowadło**.



Zebranie tego żetonu ma spowodować odnowienie kilofu, czyli przywrócenie początkowej wartości **durability**. W **PickaxeToken** przyda się metoda **repair()**:

```
public void repair() {  
    ...  
}
```

Naprawianie zrealizujemy oczywiście w metodzie **interactWithToken()**:

```
public void interactWithToken(Token token) {  
    ...  
    else if (token instanceof AnvilToken) {  
        if (pickaxeToken instanceof PickaxeToken pt) {  
            pt.repair();  
        }  
    }  
    ...  
}
```

Musimy to sprawdzić, dlatego że pole  
pickaxeToken jest typu Token.

Wygląda na to, że **zaimplementowaliśmy kilof i kowadło**. Super, brawo!

Zostało jedno zadanie: **szopa na narzędzia**.

Jednak najpierw – ponieważ sporo się ostatnio działało w kodzie – dobrze będzie na niego spojrzeć krytycznym okiem i sprawdzić, czy nie należałoby czegoś **posprzątać**.

# Porządkи

W klasie **Player** zajmujemy się dwiema sprawami:

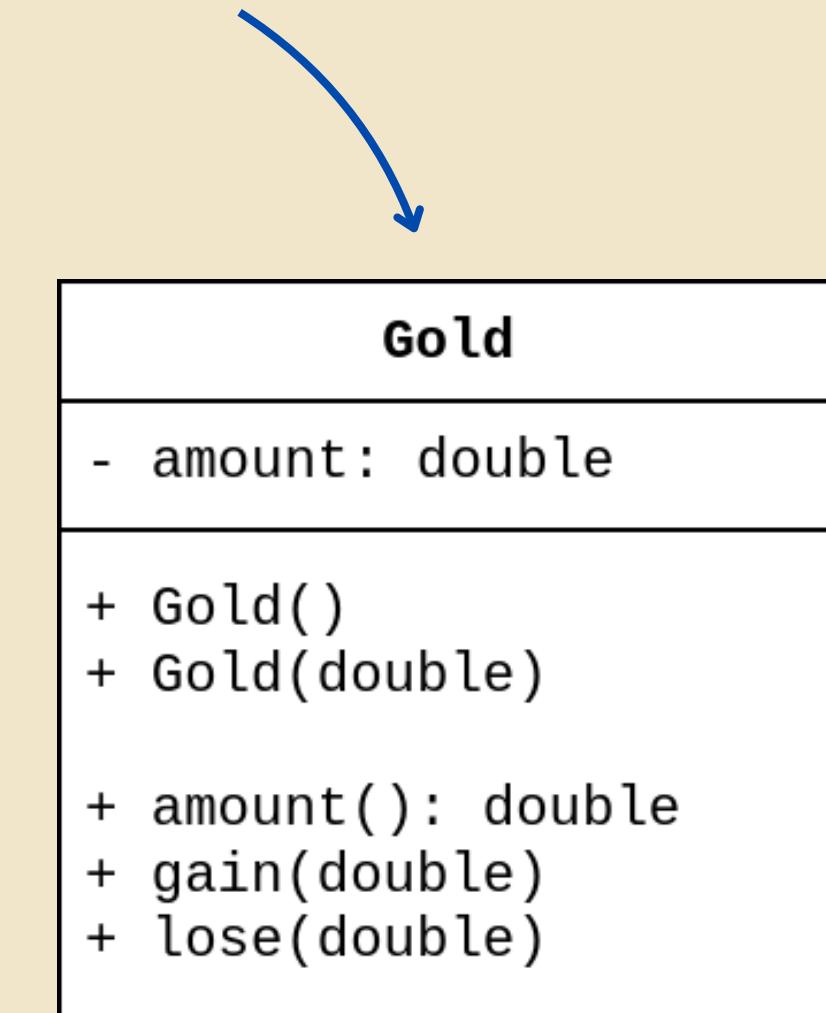
- interakcją między graczem i żetonami,
- zarządzaniem zebranym złotem.

Możemy „wyciągnąć” sprawy złota i przenieść je do osobnej klasy: **Gold**.

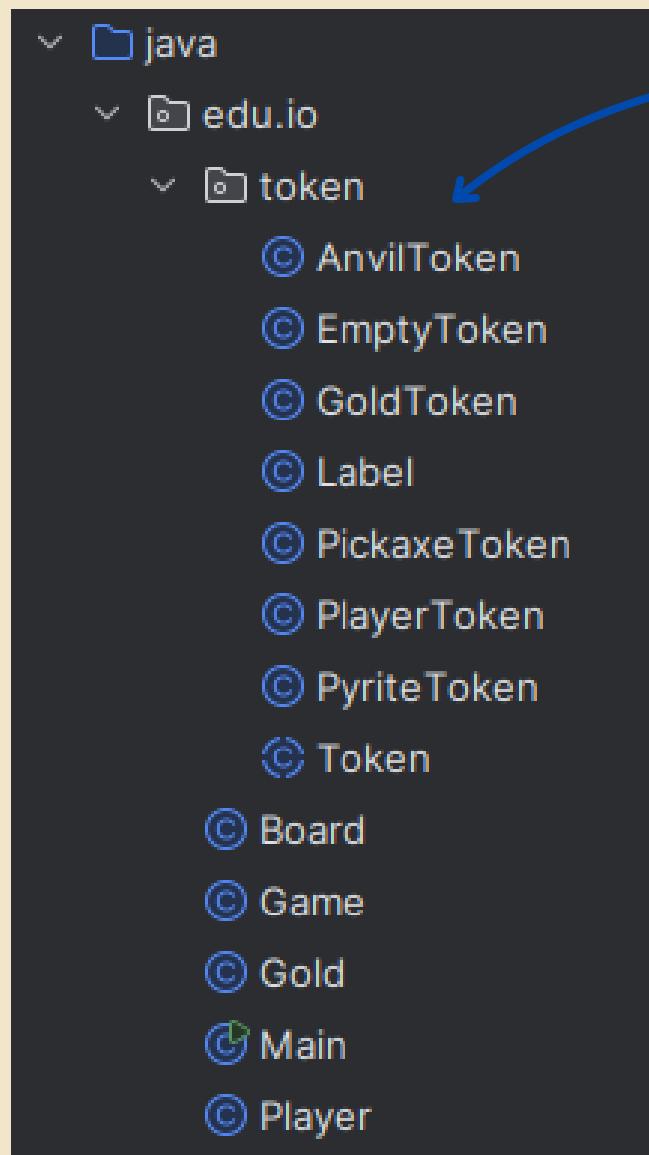
W klasie **Player** dodamy pole i utworzymy jej obiekt.

```
public class Player...  
...  
public final Gold gold = new Gold();  
...
```

W ten sposób udało nam się trochę odchudzić klasę gracza. Co dalej?

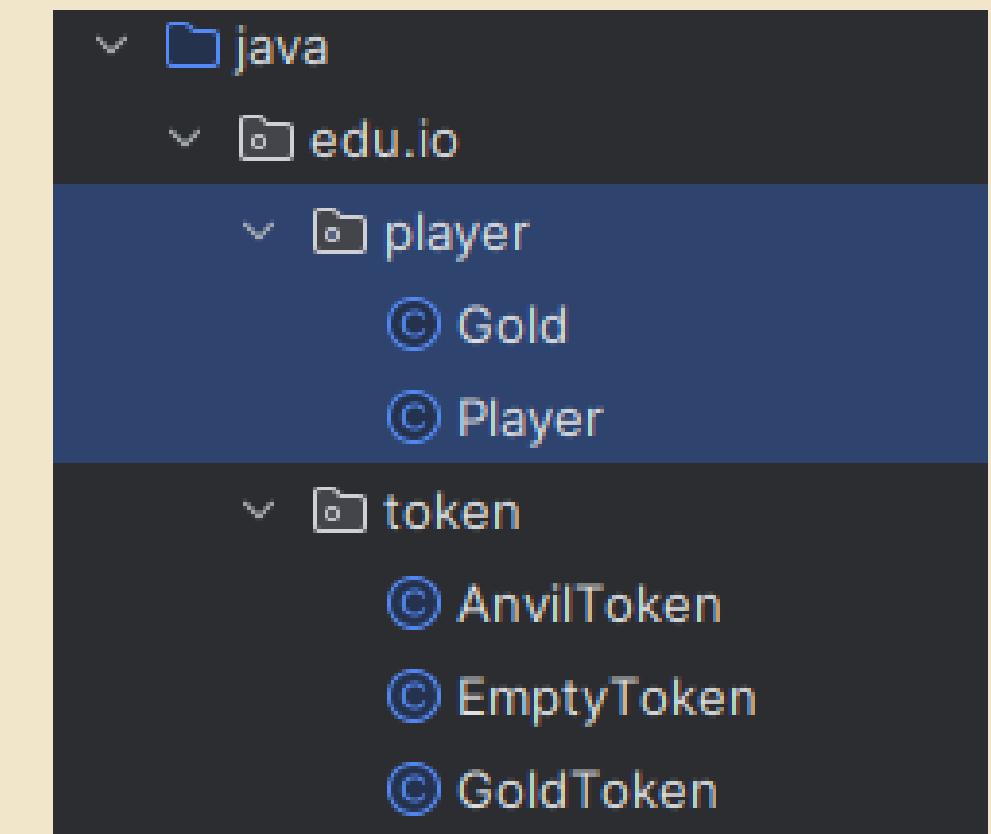


W projekcie mamy już trzynaście klas. Sporo...



Osiem z nich zgrupowaliśmy w pakiecie token.

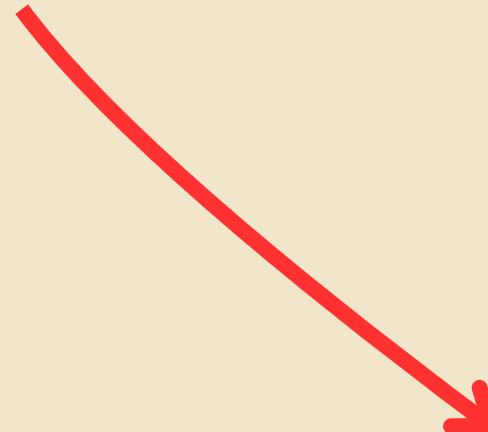
Wśród pozostałych są dwie, które są ze sobą bliżej związane:  
Player i Gold.  
Może dla nich też utworzyć pakiet?  
(zwłaszcza, że nie powiedzieliśmy jeszcze ostatniego słowa:  
w kolejce czeka przecież szopa na narzędzia)



Zajrzyjmy jeszcze do metody **interactWithToken()**:

```
if (token instanceof GoldToken goldToken) { ... }  
else if (token instanceof PickaxeToken pickaxeToken) { ... }  
else if (token instanceof AnvilToken) { ... }
```

To też *pattern matching*, ale dla instrukcji *switch*.  
Jeżeli mamy więcej możliwości, to jest bardziej czytelne.



```
switch (token) {  
    case GoldToken goldToken -> { ... }  
    case PickaxeToken pickaxeToken -> { ... }  
    case AnvilToken anvilToken -> { ... }  
    default -> { ... }  
}
```

Wprowadzenie tej zmiany nie jest obowiązkowe, ale gorąco do tego zachęcam.  
Jest to część zadania dodatkowego **i02w02\_ex1** (szczegóły na końcu prezentacji).

# Shed: szopka na narzędzia



Zamiast trzymać narzędzia (na razie mamy tylko kilof, ale z czasem pojawią się kolejne) bezpośrednio w klasie **Player**, wprowadźmy klasę pełniącą funkcję szopki na narzędzia.

Przyjmijmy założenie, że gracz zawsze **korzysta z ostatnio zdobytego narzędzia**.

A to dlatego, że szopkę ma tak zawalona narzędziami, że dostęp jest tylko do ostatniego.

Dobre wyjaśnienie?

# Co szopka powinna nam oferować?

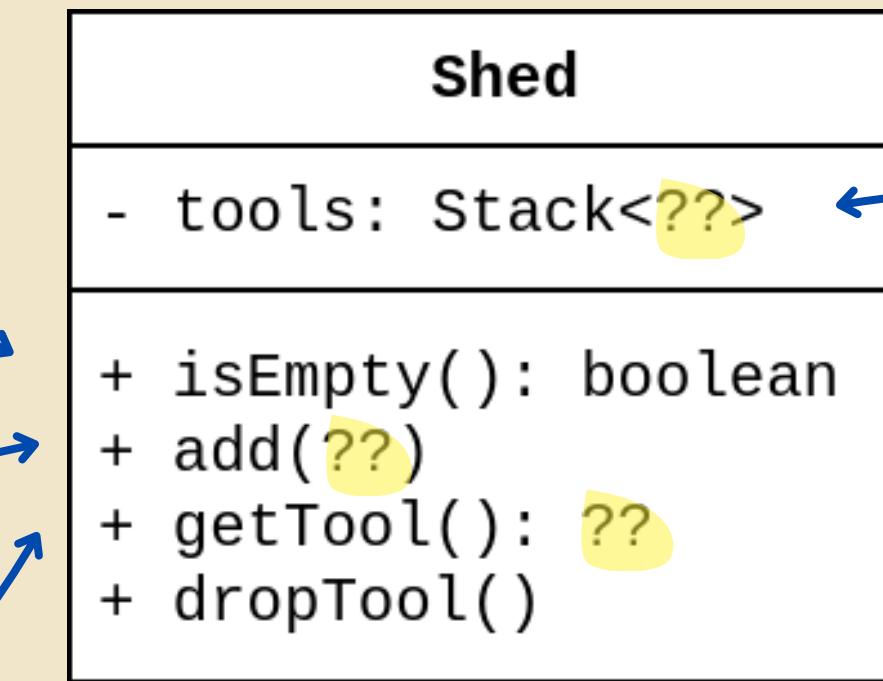
Proste pytanie: czy szopka jest pusta?

Umieść narzędzie w szopce.

Weź ostatnie narzędzie.

(uwaga: narzędzie nie jest tak naprawdę  
wyjmowane z szopki)

Wyrzuć narzędzie.



Narzędzia trzymamy na stosie.

LIFO dobrze oddaje przyjęty sposób  
funkcjonowania szopki.

O co chodzi z tym **??** w miejscu, gdzie powinien być typ?

Biorąc pod uwagę dostępne klasy, daliśmy tam **Token**.

Ale czy w szopce chcemy przechowywać **żetony**?

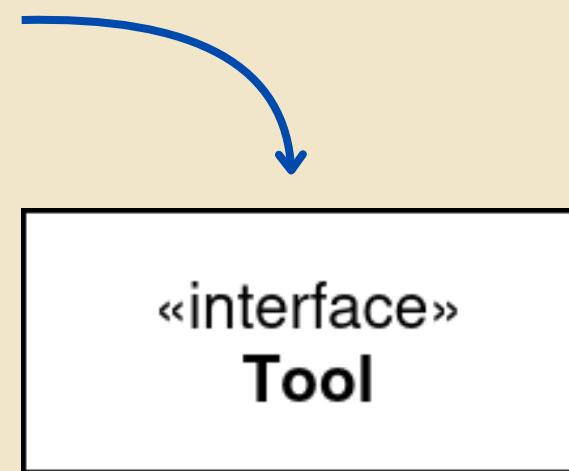
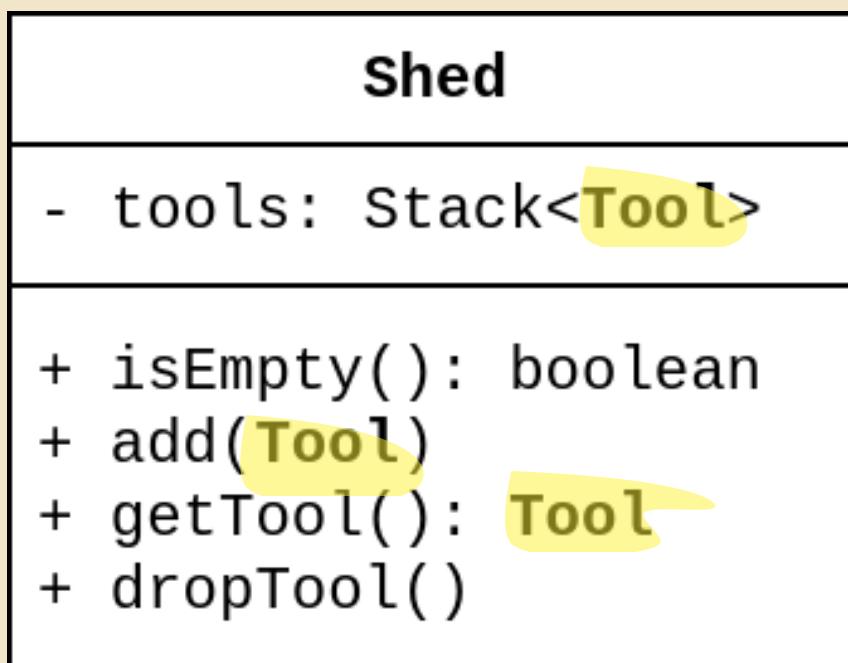
Nie: chcemy tam **narzędzi!**

Nie będziemy tworzyć osobnych klas dla narzędzi:

## ~~PickaxeToken – Pickaxe~~

W tej sytuacji byłaby to zdecydowanie przesada.

Wprowadzimy typ **Tool**, ale nie jako klasę, tylko **interfejs**.



Na razie interfejs jest pusty.  
Za chwilę wypełnimy go metodami, które świadczą o tym, że coś jest narzędziem.

Wiemy, że **kilof jest narzędziem**.

Klasa PickaxeToken musi więc **implementować** interfejs Tool:

```
public class PickaxeToken extends Token implements Tool
```

W ten sposób będziemy mogli oznaczać klasy, które reprezentują narzędzia.

Narzędzia można trzymać w szopce...

```
case PickaxeToken pickaxeToken -> {
    shed.add(pickaxeToken);
}
```

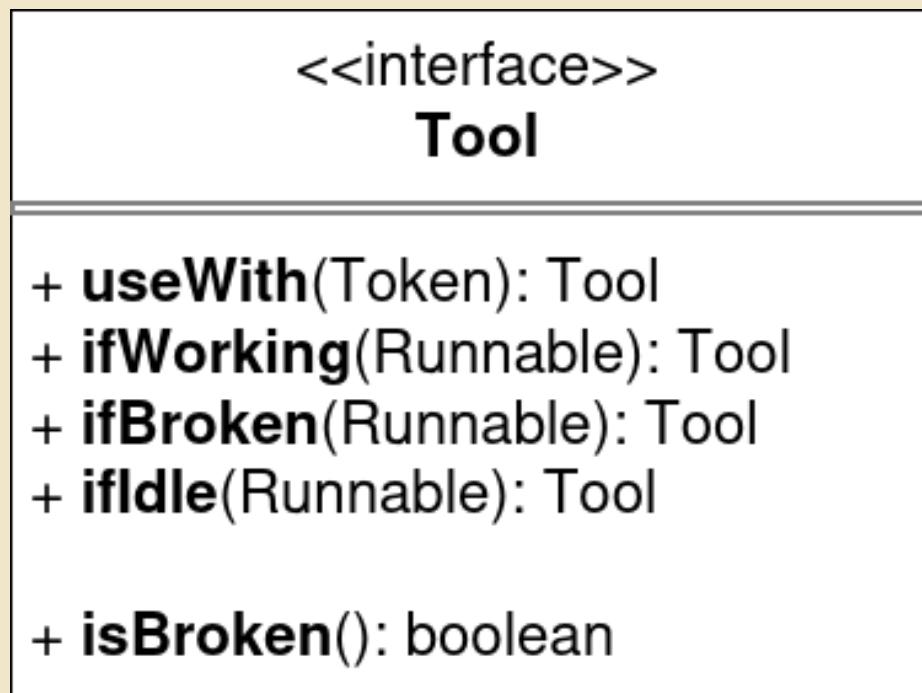
...i można ich używać przy zbieraniu złota:

```
case GoldToken goldToken -> {
    Tool tool = shed.getTool();
    if (tool instanceof PickaxeToken...
```

Interfejs na razie jest pusty.

Wypełnijmy go metodami.

W ten sposób „powiemy”, **co muszą umieć** klasy implementujące ten interfejs.



W naszej obecnej sytuacji nic się nie zmieni, dlatego że klasa PickaxeToken miała te metody. Za to przy dodawaniu kolejnego narzędzia zostaniemy **zmuszeni do zaimplementowania** metod z interfejsu. To dobrze: mamy pewność, że narzędzie będzie pełnowartościowe.

Skorzystajmy z nowo zdefiniowanego interfejsu, dodając *narzędzie – nie-narzędzie*: **NoTool**. O co chodzi?

Zadajmy sobie pytanie: **co ma zwrócić metoda getTool() jeżeli szopka jest pusta?**

No właśnie! Potrzebujemy „pustego” narzędzia. Takiego, które jest narzędziem, ale nie ma żadnej funkcjonalności.

```
public Tool getTool() {  
    if (tools.isEmpty()) {  
        return new NoTool();  
    }  
    ...  
}
```

A najlepiej utworzyć ten obiekt raz, zapisać go w prywatnym polu i potem wykorzystywać tę jedną instancję.

Takie rozwiązanie jest nazywane wzorcem „Null Object”.

```
public class NoTool implements Tool {  
    @Override  
    public Tool useWith(Token withToken) {  
        return this;  
    }  
  
    @Override  
    public Tool ifWorking(Runnable action) {  
        return this;  
    }  
  
    @Override  
    public Tool ifBroken(Runnable action) {  
        return this;  
    }  
  
    @Override  
    public Tool ifIdle(Runnable action) {  
        return this;  
    }  
  
    @Override  
    public boolean isBroken() {  
        return false;  
    }  
}
```

W interfejsie **Tool** zabrakło chyba jednej metody: **repair()**?

Dodawaliśmy ją na etapie wprowadzania kowadła.

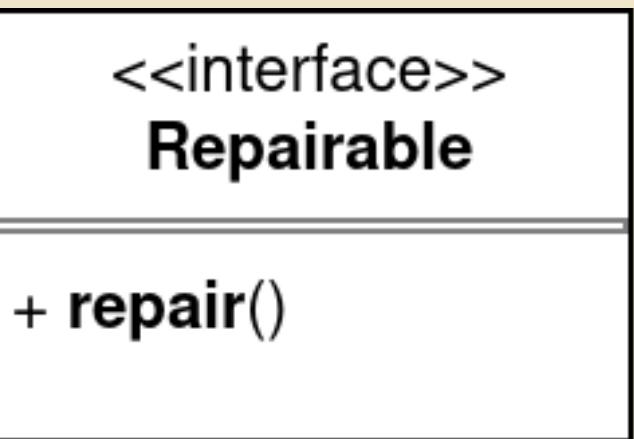
Celowo nie umieściłem jej w interfejsie **Tool**. Założyłem, że pewne narzędzia da się naprawiać, a pewnych się nie da. W związku z tym wprowadzam kolejny interfejs: **Repairable**.

Jeżeli chcę, żeby narzędzie dało się naprawiać, dopisuję:

```
public class PickaxeToken extends Token implements Tool, Repairable
```

W ten sposób w obsłudze kowadła nie sprawdzam, czy gracz ma jakieś konkretne narzędzie (np. kilof), tylko czy narzędzie jest „naprawialne”. Jeżeli tak, to je naprawiam:

```
case AnvilToken anvilToken -> {
    if (shed.getTool() instanceof Repairable tool) {
        tool.repair();
    }
}
```



# Zadania dodatkowe

## Zadanie i02w02\_ex1

Zrefaktoryzuj metodę **interactWithToken()**:

- zastosuj „**pattern matching**” dla instrukcji **switch**,
- wydziel metodę obsługującą interakcję kilofu na złocie (**private usePickaxeOnGold()**).

Dotychczasowe testy muszą przechodzić pozytywnie.

## Zadanie i02w02\_ex2

Dodaj rynnę do płukania złota: **SluiceboxToken** (przykładowa etykietka: ).

Rynna daje domyślne wzmocnienie zbierania złota 1.2 i wytrzymałość 5.

Wmocnienie **zmniejsza się** wraz z zużyciem: 0.04 na każde użycie.

Rynny **nie da się naprawić** kowadłem (odpowiednio użyj interfejsów!).

**Napisz testy** sprawdzające działanie nowego narzędzia.

# Plan kolejnej iteracji

- właściwości gracza, np. poziom nawodnienia, siła itp.
- żetony wpływające na właściwości gracza
- może popracujemy nad interfejsem użytkownika?