



# Testowanie i jakość oprogramowania

**Code style**  
**dobre praktyki**  
**code review**

Ernest Bieś, PWSZ Tarnów 2020



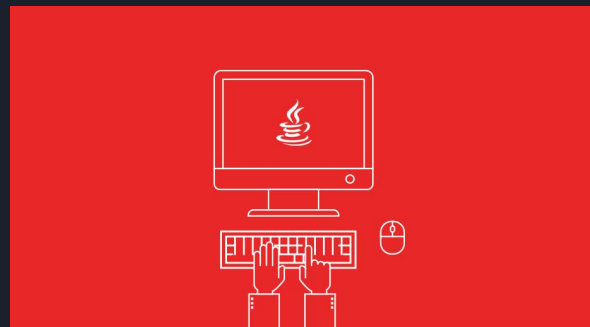
# Java - dobre praktyki

*“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” – Martin Fowler*

```
// Code Readability
if(isCodeReadable()) {
    beHappy();
} else {
    refactor();
}
```

# Java - dobre praktyki

**Dobre praktyki programowania** to klucz do sukcesu. Kod powinien zawsze być **czytelny**. Musimy zawsze mieć na uwadze, że programiści zazwyczaj pracują w zespole, a co za tym idzie nad jednym kodem pracuje wiele osób. Poprawne **nazewnictwo**, **czytelność kodu** to podstawy, które każdy programista powinien znać i rozumieć.



# Komentarze

**Komentarze nie są szminką dla złego kodu!** - to bardzo ważne oraz istotne stwierdzenie. Jeżeli nasz kod jest źle napisany lub nie jest on czytelny nie komentujemy go, tylko poprawiamy. **Czytelny** oraz **prawidłowo napisany** kod często nie wymaga komentarzy.

```
//Sprawdzenie czy jest pełnoletni  
public boolean check(Person p) {  
    if (p.getAge() >= 18) {  
        return true;  
    } else {  
        return false;  
    }  
}
```



```
public boolean checkPersonIsAdult(Person person) {  
    return person.getAge() >= 18;  
}
```

Po zmianie nazwy metody komentarz jest teraz zbędny.  
**Nazwa metody** przekazuje nam informację o tym czego dotyczy.

# Komentarze

```
// Do metody przekazujemy numer przykładu.  
// Metoda wywołuje odpowiednią funkcję (przykład zadania) z klasy ExampleRunner  
public void runExample(int number) {  
    switch (number) {  
        case 0:  
            ExampleRunner.mapExample();  
            break;  
        case 1:  
            ExampleRunner.filterExample();  
            break;  
        case 2:  
            ExampleRunner.streamsExample();  
            break;  
        default:  
            ExampleRunner.codingExample();  
    }  
}
```

Często **komentarze informacyjne** są jednak bardzo przydatne. W powyższym przykładzie objaśniamy działanie naszej metody.

# Komentarze

W niektórych przypadkach komentarze są bardzo przydatne w **wyjaśnianiu zamierzeń**. Możemy nie zgodzić się ze sposobem rozwiązania problemu, jednak wiemy co programista miał na myśli.

```
// Metoda zwraca listę liczb parzystych lub nieparzystych w zależności od przekazanego typu
// type = 0 - metoda zwraca liczby parzyste
// type = 1 - metoda zwraca liczby nieparzyste
public List<Integer> getEvenOddNumbers(ArrayList<Integer> listOfNumbers, int type) {
    if (type == 0) {
        return listOfNumbers.stream().filter(num -> num % 2 == 0).collect(Collectors.toList());
    } else {
        return listOfNumbers.stream().filter(num -> num % 2 != 0).collect(Collectors.toList());
    }
}
```

# Javadoc

**Javadoc** – narzędzie automatycznie generujące dokumentację na podstawie zamieszczonych w kodzie źródłowym znaczników w komentarzach. Javadoc został stworzony specjalnie na potrzeby języka programowania **Java** przez firmę **Sun Microsystems**.





# Javadoc

Komentarz **Javadoc** oddzielony jest znacznikami `/**` i `*/`, które sprawiają, że ich zawartość (czyli to, co znajduje się między nimi), jest ignorowana przez kompilator. Pierwsza jego linia to opis metody lub klasy, która zadeklarowana jest poniżej. Dalej znajduje się pewna liczba opcjonalnych tagów, które z kolei opisują parametry metody (**@param**), wartość zwracaną (**@return**) itp.



# Javadoc - przykład

```
/**
 * Metoda zwraca listę liczb parzystych lub nieparzystych w zależności od przekazanego typu
 * @param listOfNumbers przekazywana lista liczb
 * @param type           przekazywany typ (0 - liczby parzyste, 1 - liczby nieparzyste)
 * @return              zwracana lista liczb parzystych/nieparzystych
 * */
public List<Integer> getEvenOddNumbers(ArrayList<Integer> listOfNumbers, int type) {
    if (type == 0) {
        return listOfNumbers.stream().filter(num -> num %2 == 0).collect(Collectors.toList());
    } else {
        return listOfNumbers.stream().filter(num -> num %2 != 0).collect(Collectors.toList());
    }
}
```



# Nazewnictwo

- 1) Kod piszemy po **angielsku**.
- 2) Nazwy klas to **rzeczowniki** pisane **wielką literą**.
- 3) Nazwy metod to **czasowniki** pisane **małą literą** albo wyrażenia, które zaczynają się od czasownika zapisanego małą literą, a **pierwsza litera** każdego kolejnego słowa jest **duża**.
- 4) Zmienne piszemy **małą literą**.
- 5) Stałe zapisujemy **wielką literą**.
- 6) Nazwy pakietów piszemy **małymi literami**, zwyczajowo nazwa pakietu jest nazwą domeny, ale odwracamy kolejność poszczególnych członów.

# Nazewnictwo - przykład

```
package com.ernestbies;
```

Nazwa pakietu pisana z małej litery.

```
public class Example {
```

Nazwa klasy pisana z wielkiej litery.

```
    private int number;
```

Nazwa zmiennej pisana z małej litery.

```
    private static final double PI = 3.14;
```

Nazwa stałej pisana z wielkiej litery.

```
    public int getNumber() {
```

Nazwy metod to

```
        return number;
```

czasowniki pisane małą

literą, a pierwsza litera

każdego kolejnego słowa

jest duża.

```
    }
```

```
}
```

# Nazewnictwo - przykłady

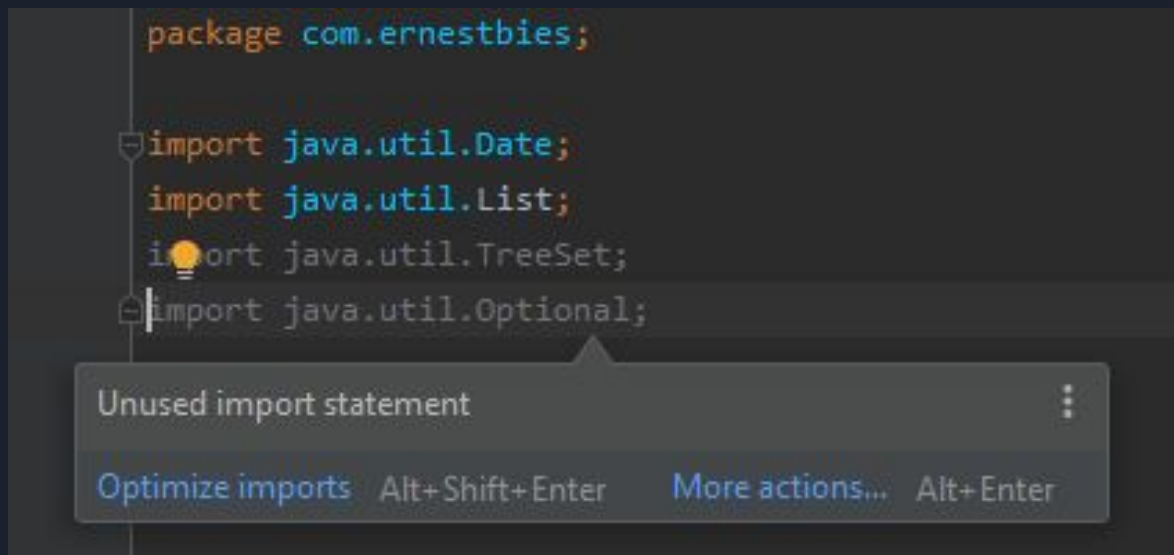
Prose form	Correct	Incorrect
"XML HTTP request"	<code>XmlHttpRequest</code>	<code>XMLHTTPRequest</code>
"new customer ID"	<code>newCustomerId</code>	<code>newCustomerID</code>
"inner stopwatch"	<code>innerStopwatch</code>	<code>innerStopWatch</code>
"supports IPv6 on iOS?"	<code>supportsIpv6OnIos</code>	<code>supportsIPv6OnIOS</code>
"YouTube importer"	<code>YouTubeImporter</code> <code>YoutubeImporter</code> *	

\*Acceptable, but not recommended.

**Note:** Some words are ambiguously hyphenated in the English language: for example "nonempty" and "non-empty" are both correct, so the method names `checkNonempty` and `checkNonEmpty` are likewise both correct.

# Importy

Wszystkie nieużywane **importy** powinny zostać **usunięte**.  
Środowisko **IntelliJ IDEA** samo zgłosi nam taką sytuację. Kombinacja **Alt + Shift + Enter** powoduje usunięcie nieużywanych importów.





# Formatowanie kodu

Nasz kod powinien być **czytelny** oraz dobrze **sformatowany**. Każda, nawet pojedyncza instrukcja powinna być objęta **nawiasami klamrowymi**. Poniższy kod działa, lecz **nie jest** on do końca **poprawnie** sformatowany.

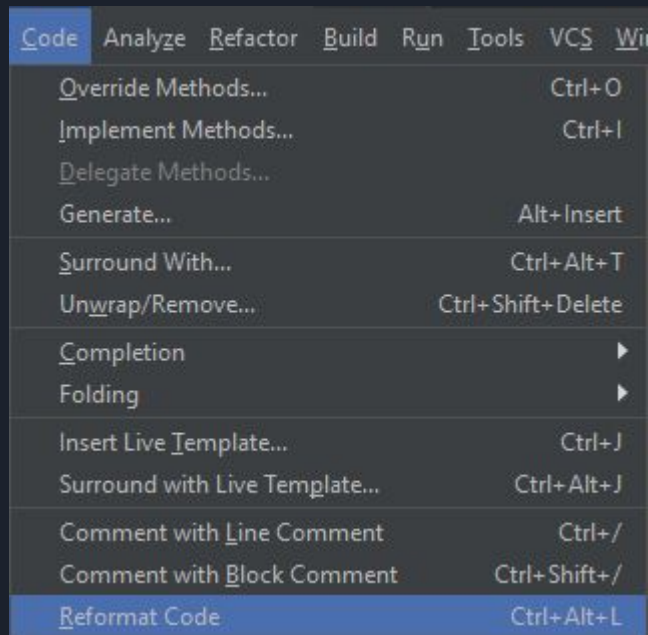
```
public List<Integer> getEvenOddNumbers(ArrayList<Integer> listOfNumbers, int type) {  
    if (type == 0)  
        return listOfNumbers.stream().filter(num -> num %2 == 0).collect(Collectors.toList());  
    else  
        return listOfNumbers.stream().filter(num -> num %2 != 0).collect(Collectors.toList());  
}
```

Każda **pojedyncza instrukcja** powinna być w nawiasach klamrowych:

```
public List<Integer> getEvenOddNumbers(ArrayList<Integer> listOfNumbers, int type) {  
    if (type == 0) {  
        return listOfNumbers.stream().filter(num -> num %2 == 0).collect(Collectors.toList());  
    } else {  
        return listOfNumbers.stream().filter(num -> num %2 != 0).collect(Collectors.toList());  
    }  
}
```

# Formatowanie kodu - środowisko IntelliJ IDEA

Środowisko **IntelliJ IDEA** pomaga nam w formatowaniu naszego kodu. Wystarczy z zakładki **Code** wybrać opcję **Reformat Code** (skrót klawiszowy **Ctrl+Alt+L**). Należy jednak pamiętać, że środowisko nie zrobi wszystkiego za nas i znajomość pewnych zasad obowiązkowa.





# Formatowanie kodu - środowisko IntelliJ IDEA

Przykład kodu **niepoprawnie** sformatowanego.

```
public int getHigherNumber(int number) {  
    if(number % 5 == 0) {return number + 100;}  
    else {return number + 1000;}  
}
```

**Ctrl + Alt + L**



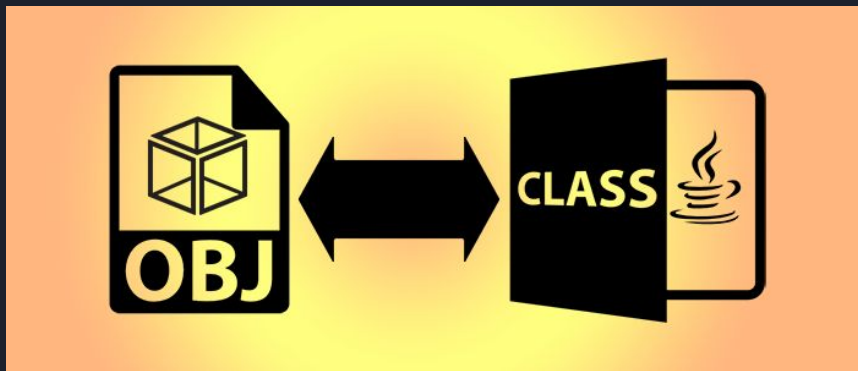
```
public int getHigherNumber(int number) {  
    if (number % 5 == 0) {  
        return number + 100;  
    } else {  
        return number + 1000;  
    }  
}
```

Nasz kod został **poprawnie** sformatowany.



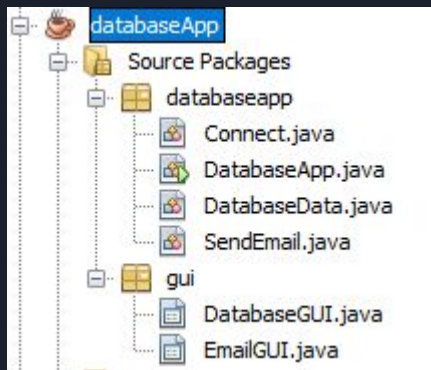
# Klasy powinny być małe!

Dzielenie programu na **klasy** to bardzo ważny element programowania obiektowego. Każda klasa powinna mieć jasno określone zastosowanie, reprezentować określoną strukturę. Dzielenie programu na klasy przynosi ze sobą **wiele korzyści**.



# Warto dzielić program na klasy

Przykład :: Program do łączenia się z bazą danych za pomocą **JDBC**.



```
public class DatabaseData implements Serializable {  
    private String host;  
    private String dbname;  
    private String user;  
    private String port;  
    private String pass;  
    ...  
}
```

Zapisanie danych do połączenia się z bazą w **osobnej klasie** i użycie **Serializacji** zapewnia nam możliwość łatwego zapisania tych danych w pliku i odczytania w razie potrzeby.. (Nie jest to najlepsze rozwiązanie, przykład tylko dla zobrazowania problemu).

# Enkapsulacja danych

**Enkapsulacja** (inaczej hermetyzacja) to **ukrywanie** widoczności pól danej klasy dla innych klas, co w ten sposób chroni dane przechowywane w tych polach przed niepowołanym, lub co najmniej nieuzasadnionym dostępem. Dostęp do danych powinien być realizowany za pomocą **getterów** i **setterów**.

```
public class Person {  
    private String firstName;  
    private String lastName;  
    private int age;  
}
```

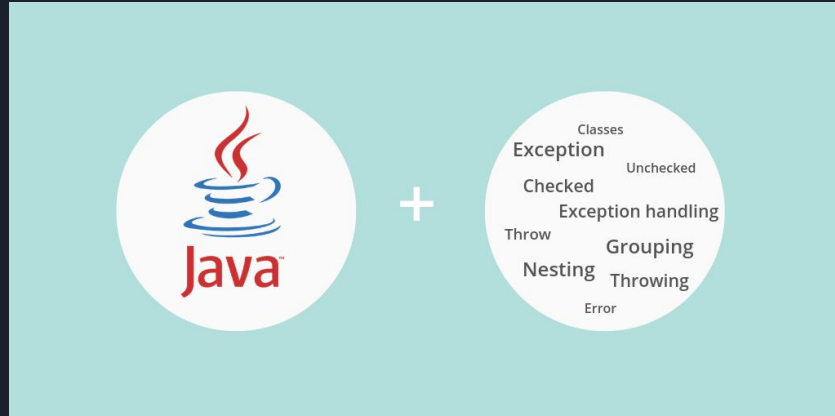
```
public class HiddenData {  
    private HiddenData() {  
    }  
}
```

```
HiddenData hiddenData = new HiddenData();
```

Możemy również stworzyć **prywatny konstruktor**, jeżeli z jakiś przyczyn chcemy zabezpieczyć naszą klasę przed utworzeniem jej instancji. (Domyślnie tworzony jest konstruktor publiczny).

# Obsługa wyjątków

**Wyjątek** jest specjalną klasą. Jest ona specyficzna ponieważ w swoim łańcuchu dziedziczenia ma klasę **java.lang.Throwable**. Instancje, które w swojej hierarchii dziedziczenia mają tę klasę mogą zostać „rzucone” przerywając standardowe wykonanie programu.





# Rzucanie wyjątków

**Przykład ::** Jeśli metoda zostanie wywołana z argumentem **mniejszym od 0** możemy uznać to za nieprawidłowe wywołanie i zasygnalizować taką sytuację **rzucając wyjątek**.

```
public int getNumberOfSeconds(int hour) {  
    if (hour < 0) {  
        throw new IllegalArgumentException("Hour must be >= 0: " + hour);  
    }  
    return hour * 60 * 60;  
}
```



# Łap wyjątki - nie ignoruj!

Wyjątki musimy **obsłużyć**. Mówimy, że wyjątek jest obsługiwany, jeśli reagujemy na jego wystąpienie i próbujemy “naprawić” program w trakcie jego działania. Do obsługi wyjątków służy **try / catch**.

```
int hours = -3;
int numberOfSeconds = 0;
try {
    numberOfSeconds = instance.getNumberOfSeconds(hours);
}
catch (IllegalArgumentException exception) {
    numberOfSeconds = instance.getNumberOfSeconds(hours * -1);
}
```

Jeśli kod wewnątrz nawiasów **{}** rzuci **wyjątek** i blok **catch** będzie obsługiwał ten typ wyjątku wówczas zostanie wywołany kod w bloku **catch** i wyjątek **nie przerwie** działania programu.

**Mechanizm obsługi wyjątków** w Javie jest bardzo ważny i powinniśmy z niego korzystać.

# Refaktoryzacja

**Refaktoryzacja** – proces **wprowadzania zmian** w projekcie/programie, w wyniku których zasadniczo nie zmienia się funkcjonalność. Celem refaktoryzacji jest więc nie wytwarzanie nowej funkcjonalności, ale utrzymywanie **odpowiedniej, wysokiej jakości organizacji systemu**.

```
public static long getNumbersCount(String numbers) throws EmptyDataException {  
    if (numbers.isEmpty() || numbers == null) {  
        throw new EmptyDataException();  
    } else {  
        return Arrays.stream(numbers.split(",")).count();  
    }  
}
```

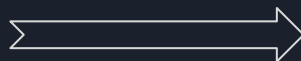




# Kolejność warunków

Kompilator kompiluje program od “**lewej do prawej strony**”. Jest to bardzo ważna o której musimy pamiętać, szczególnie przy ustalaniu warunków.

LEFT



RIGHT



# Kolejność warunków - przykład

Na początku sprawdzamy czy String number jest pusty, następnie sprawdzamy null -a. Wywołujemy metodę: **getNumbersCount(null)**.

```
// Funkcja zwraca ilość liczb zapisanych jako łańcuch znakowy rozdzielony przecinkiem
public static long getNumbersCount(String numbers) throws EmptyDataException {
    if (numbers.isEmpty() || numbers == null) {
        throw new EmptyDataException();
    } else {
        return Arrays.stream(numbers.split(regex: ",")).count();
    }
}
```

```
Exception in thread "main" java.lang.NullPointerException
    at com.ernestbies.Main.getNumbersCount(Main.java:72)
    at com.ernestbies.Main.main(Main.java:18)
```

Process finished with exit code 1

```
// Funkcja zwraca ilość liczb zapisanych jako łańcuch znakowy rozdzielony przecinkiem
public static long getNumbersCount(String numbers) throws EmptyDataException {
    if (numbers == null || numbers.isEmpty()) {
        throw new EmptyDataException();
    } else {
        return Arrays.stream(numbers.split(regex: ",")).count();
    }
}
```

```
Exception in thread "main" com.ernestbies.EmptyDataException
    at com.ernestbies.Main.getNumbersCount(Main.java:73)
    at com.ernestbies.Main.main(Main.java:18)
```

Process finished with exit code 1

# Kolejność warunków - przykład

Kompilator sam powiadamia nas, że warunek **nigdy** nie zostanie spełniony, dlatego musi **odwrócić** kolejność.

```
// Funkcja zwraca ilość liczb zapisanych jako łańcuch znakowy rozdzielony przecinkiem
public static long getNumbersCount(String numbers) throws EmptyDataException {
    if (numbers.isEmpty() || numbers == null) {
        throw new EmptyDataException();
    }
}
```

Condition 'numbers == null' is always 'false' when reached

Simplify 'numbers == null' to false Alt+Shift+Enter More actions... Alt+Enter



# Programowanie funkcyjne oraz strumienie

**Programowanie funkcyjne** - ma ono swoje podstawy już w latach trzydziestych XX wieku gdy to Alonzo Church opracował **rachunek lambda**. W programowaniu funkcyjnym, w odróżnieniu od programowania obiektowego, najważniejszym i zarazem jedynym narzędziem są **funkcje**. Dzięki temu możemy tworzyć **zwięzły**, **czysty** i **wydajny kod**. **Strumienie** pozwalają w łatwy sposób zrównoleglić pracę na danych. Dzięki temu przetwarzanie dużych zbiorów danych może być **dużo szybsze**.

# Strumienie

```
// Funkcja zwraca listę wszystkich słów zawierających 'A'
public static List<String> getWordsWithA(String words) {
    String[] listOfWords = words.split( regex: ",");
    List<String> newList = new ArrayList<>();
    for(String word: listOfWords){
        if(word.contains("A")){
            newList.add(word);
        }
    }
    return newList;
}
```

**Przykład ::** Metoda zwraca listę wszystkich słów zawierających literę **'A'**. Funkcja została napisana bez użycia strumieni.

Po użyciu **strumieni** naszą metodę możemy zapisać w jednej linijce. Kod jest **zwięzły**, **czytelny** oraz **łatwy do analizy**.

```
// Funkcja zwraca listę wszystkich słów zawierających 'A'
public static List<String> getWordsWithA(String words) {
    return Arrays.stream(words.split( regex: ",")).filter(word -> word.contains("A")).collect(Collectors.toList());
}
```



# Strumienie

Przykłady prostych algorytmów oraz wykorzystania strumieni w języku **Java**.

Link do repozytorium na **Bitbucket**:

<https://bitbucket.org/ernestbies/tijo-good-practices/>



# Nie rzucaj nullami!

Od Javy 8 mamy możliwość korzystania z klasy `Optional`, która pozwala nam nie robić tradycyjnego sprawdzenia: **if (object != null)**.

Klasyczne podejście:

```
private String saveTrim(final String input) {  
    return input == null ? "" : input.trim();  
}
```

**Optional** powstał po to żebyśmy nie musieli umieszczać w kodzie warunków sprawdzających czy **referencja** ma wartość **null**.

```
private String saveTrim(final Optional<String> input) {  
    return optional.ofNullable(input)  
        .map(String::trim)  
        .orElse("");  
}
```



# Adnotacje

Dawniej, kiedy w Javie nie było adnotacji, stosowane było specjalne nazewnictwo (np. prefixy), aby zaznaczyć, że dany element potrzebuje specjalnego traktowania (np. przez framework).

**Adnotacje** rozwiązują wszystkie te problemy i to je właśnie powinniśmy używać. Jedną z najważniejszych i najpopularniejszych adnotacji w standardowej bibliotece **Javy** to **@Override**. Używana jest na metodach i deklaruje, że dana metoda nadpisuje metodę w nadklasie.

```
@Override
public String toString() {
    return firstName + " " + lastName;
}
```



# done, done, done

To określenie pochodzi ze Scruma a poszczególne 'done' oznaczają:

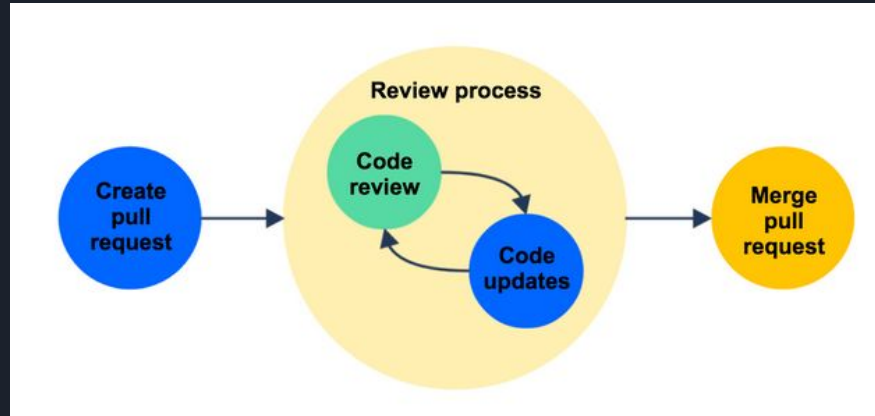
- **kod napisany**, czyli taki, który działa lokalnie ("u mnie działa")
- **kod przetestowany**, za pomocą testów jednostkowych, a także integracyjnych
- **kod zatwierdzony**, czyli taki, który został zatwierdzony przez Product Ownera, jako realizujący konkretną potrzebę biznesową wynikającą z zadania. Taki kod jest zmerdżowany do mastera i stanowi część naszego produktu.

Warto pamiętać o tej zasadzie podczas implementacji naszego kodu.



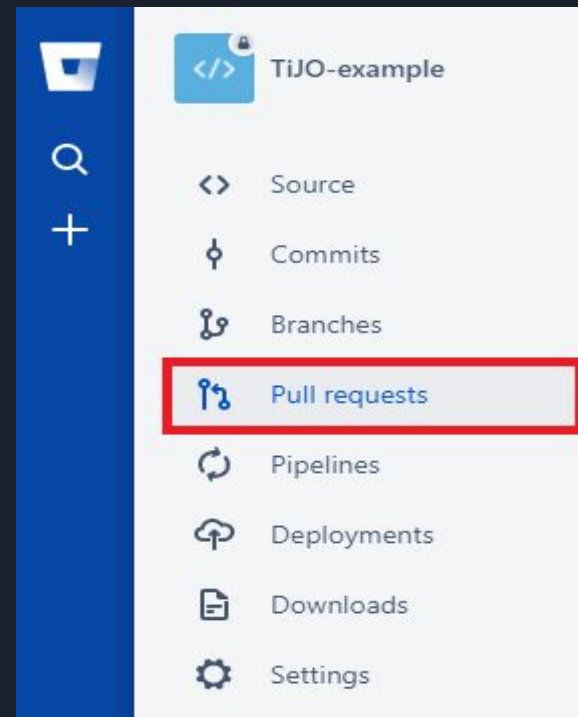
# Code review

**Pull Request** to metoda scalania gałęzi w gicie. Na tej zasadzie rozwijane są różne projekty w tym projekty typu **open source**. Programista tworzy własnego brancha gdzie wprowadza swoje zmiany i wysyła je do repozytorium – robi tzw. **PR (z ang. pull request)**. Zmiany te są testowane przez mechanizm **CI (z ang. continuous integration)** a następnie trafiają przed scaleniem do głównego brancha do weryfikacji (z ang. *review*) przez administratora repozytorium dostarczając również informacje o potencjalnych konfliktach. Na końcu procesu zmiany są scalane z główną gałęzią projektu.



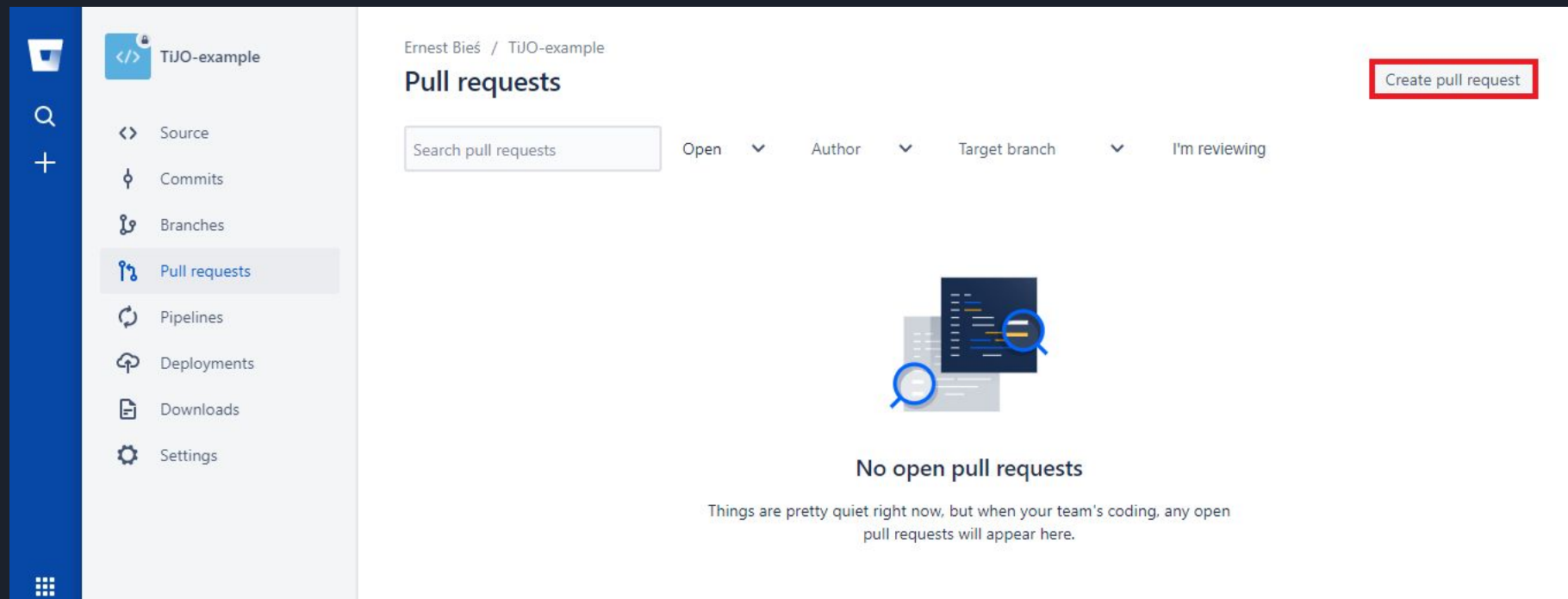
# Mechanizm Pull Requests na przykładzie serwisu Bitbucket

W serwisie **Bitbucket** w lewym górnym rogu przechodzimy do repozytorium a następnie klikamy opcję **Pull requests**.



# Mechanizm Pull Requests na przykładzie serwisu Bitbucket


Po przejściu do zakładki **Pull requests** wybieramy opcję **Create pull request**.




The screenshot shows the Bitbucket interface for a repository named 'TiJO-example'. On the left is a blue sidebar with navigation icons and labels: Source, Commits, Branches, Pull requests (highlighted), Pipelines, Deployments, Downloads, and Settings. The main content area is titled 'Ernest Bieś / TiJO-example' and 'Pull requests'. It features a search bar labeled 'Search pull requests' and a filter bar with 'Open', 'Author', 'Target branch', and 'I'm reviewing'. A red box highlights the 'Create pull request' button in the top right corner. Below the filter bar is an illustration of a document with a magnifying glass. The text 'No open pull requests' is displayed, followed by a message: 'Things are pretty quiet right now, but when your team's coding, any open pull requests will appear here.'

# Mechanizm Pull Requests na przykładzie serwisu Bitbucket

Create a pull request


 ernestbies / TiJO-example  
Created 27 minutes ago, updated 9 minutes ago  
feature/my-feature

→

 ernestbies/tijo-example  
develop

Title <sup>\*</sup> TiJO - Pull-request example

Description

Reviewers  Konrad Czechowski x

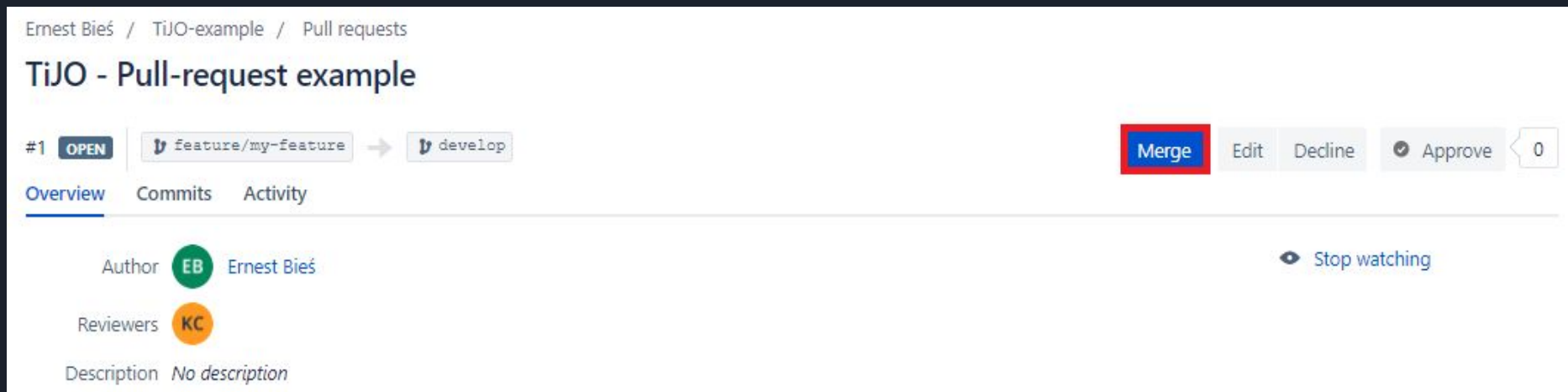
[Add all commit authors as reviewers](#)

Close branch ☐ Close **feature/my-feature** after the pull request is merged

Create pull request

Następnie wybieramy nasz **branch** (w tym wypadku **feature/my-feature**) oraz **branch docelowy**, z którym chcemy się zmerge'ować (w tym wypadku **develop**). Możemy również ustawić tytuł, opis oraz recenzentów z naszego zespołu. Następnie naciskamy **Create pull request**.

# Mechanizm Pull Requests na przykładzie serwisu Bitbucket



The screenshot shows a Bitbucket Pull Request interface. At the top, the breadcrumb navigation reads 'Ernest Bieś / TiJO-example / Pull requests'. The title of the pull request is 'TiJO - Pull-request example'. Below the title, it shows '#1' followed by a blue 'OPEN' button. The source branch is 'feature/my-feature' and the target branch is 'develop', connected by a right-pointing arrow. On the right side, there are four buttons: 'Merge' (highlighted with a red border), 'Edit', 'Decline', and 'Approve' (which has a checkmark icon). To the right of the 'Approve' button is a dropdown menu showing the number '0'. Below the buttons, there are three tabs: 'Overview' (selected with a blue underline), 'Commits', and 'Activity'. Under the 'Overview' tab, the 'Author' is listed as 'Ernest Bieś' with a green circular avatar containing 'EB'. The 'Reviewers' section shows 'KC' with an orange circular avatar. The 'Description' field is empty and labeled 'No description'. In the top right corner of the pull request area, there is a link that says 'Stop watching' with an eye icon.


Nasz **pull-request** jest otwarty (posiada status **Open**). W tym momencie inna osoba z zespołu oceni nasze zmiany. Po **zaakceptowaniu** zmian przez drugą osobę możemy wykonać **Merge**.


# Mechanizm Pull Requests na przykładzie serwisu Bitbucket

**TiJO - Pull-request example**

#1 **OPEN** `feature/my-feature` → `develop` **Merge** Edit Decline **Approve** < 1


Overview Commits Activity


Author  Ernest Bieś [Stop watching](#)

Reviewers 

Description *No description*

**Comments (1)**


 Konrad Czechowski  
Looks great to me!  
Reply • Like • Delete • Create task • 41 seconds ago


 What would you like to say?

Po **zaakceptowaniu** zmian przez członka naszego zespołu możemy przejść do **Merge**.

# Mechanizm Pull Requests na przykładzie serwisu Bitbucket

**Merge pull request**

Source  feature/my-feature

Destination  develop

Commit message 

Merged in feature/my-feature (pull request #1)

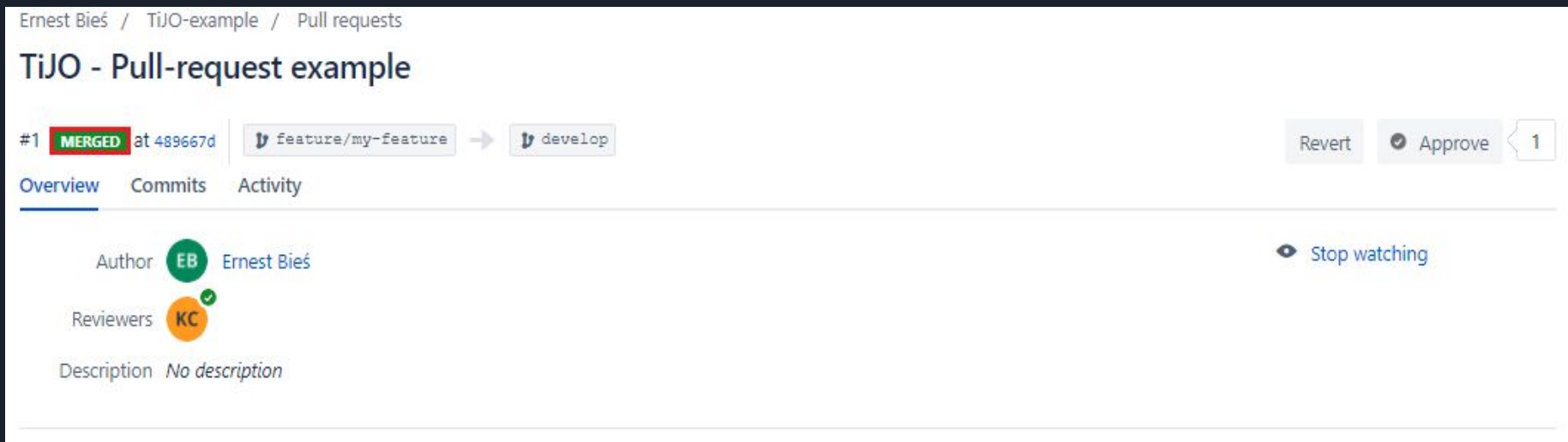
TiJO - Pull-request example

Merge strategy Merge commit ▼

☐ Close source branch

**Merge** Cancel

# Mechanizm Pull Requests na przykładzie serwisu Bitbucket





Ernest Bieś / TiJO-example / Pull requests

## TiJO - Pull-request example

#1 **MERGED** at 489667d | `feature/my-feature` → `develop` Revert Approve 1

[Overview](#) [Commits](#) [Activity](#)

Author  Ernest Bieś Stop watching

Reviewers 

Description *No description*

Po wykonaniu merge'a status **Pull requesta** zmienia się na **Merged**.  
Zmiany zostały zatwierdzone oraz wysłane.





# Pull Requests w środowisku IntelliJ IDEA

Dzięki środowisku **IntelliJ IDEA** mamy również możliwość tworzenia **Pull requests**.

Link do tutoriala:

<https://www.jetbrains.com/help/idea/contribute-to-projects.html>

Pełny tutorial tworzenia Pull requestów w serwisie **Bitbucket**:

<https://confluence.atlassian.com/bitbucket/create-a-pull-request-to-merge-your-change-774243413.html>



# Dziękuję za uwagę!

Źródła:

- 1) <https://google.github.io/styleguide/javaguide.html>
- 2) <https://kobietydokodu.pl/niezbednik-juniora-dobre-praktyki-dla-poczatkujacego-programisty/>
- 3) <https://howtodoinjava.com/java-best-practices/>
- 4) **Czysty kod. Podręcznik dobrego programisty - “Robert C.Martin”**